Quantifying quantum walk speed-ups

MSc Thesis *(Afstudeerscriptie)*

written by

Martijn Brehm

under the supervision of **Jordi Weggemans** and **Harry Buhrman**, and submitted to the Examinations Board in partial fulfillment of the requirements for the degree of

MSc in Logic

at the *Universiteit van Amsterdam.*

Date of the public defense:	**Members of the Thesis Committee:**
July 5th, 2023	dr. Benno van den Berg (chair)
	prof. dr. Harry Buhrman
	prof. dr. Michael Walter
	dr. Nicolas Resch
	Jordi Weggemans, MSc

Institute for Logic, Language and Computation

Contents

Abstract

Given specific instances of a computational problem like 3SAT, can a quantum computer solve these instances more efficiently than a classical computer? By expressing the complexity of our algorithms asymptotically we have inhibited ourselves from rigorously answering this question analytically. Since (large fault-tolerant) quantum computers don't exist (yet), we also can't answer this question empirically.

In this work, we bridge this gap. We derive exact expressions for the query complexity of several quantum walk search algorithms by parametrising in several non-trivial, yet still efficiently classically computable, input features. Specifically, we do so for Belovs' detection algorithm [Bel13], a search algorithm for trees using Belovs' detection algorithm, and a general search algorithm due to Piddock [Pid19]. We also optimally configure Belovs' detection algorithm and experimentally show a speed-up of a factor 10 compared to both Belovs' and Montanaro's proposed configurations in certain settings, slightly optimise Piddock's algorithms, and derive upper-bounds on the inaccuracy of two subroutines of Piddock's algorithm.

We then apply the search algorithms in a quantum backtracking algorithm for constraint satisfaction problems, promising a quadratic speed-up over the corresponding classical backtracking algorithm [Mon16]. Specifically we study uniformly randomly generated 3SAT instances in the critical regime, and compare to the classical DPLL algorithm, using our expressions to compute the exact quantum query complexity.

For the detection-based binary search algorithm, we observe a polynomial quantum speed-up of order between 1.36 to 1.56 in the query complexity (depending on the algorithm's configuration), which manifests in 3SAT instances in roughly 330 to 545 variables and on. For Piddock's search algorithm, we find a polynomial speed-up of order at most 1.58, which starts to occur in 3SAT instances in least 430 variables, where we note that Piddock's algorithm assumes access to (an upper-bound on) the size of DPLL's backtracking tree, the computation of which hasn't been accounted for in these numbers. Surprisingly, then, Piddock's algorithm doesn't seem to perform better than the binary search algorithm, despite a clear asymptotic advantage.

Since these observed speed-ups are in the query complexity model, they ignore the significant constant overhead suffered by actually implementing quantum operations and memory [BMT+22]. The above results should therefore be interpreted as as showing the *possibility* of a quantum speed-up for solving uniformly random 3SAT instances in the critical regime: if a speed-up wasn't found here, it certainly won't exist after taking further overhead into account.

Chapter 1

Introduction

The study of computational problems, algorithms and their complexity is of great philosophical interest: by delineating the extent of efficient computation, we also delineate the possible extent of human knowledge. However, at the end of the day, such study mostly informs the application of algorithms to real-life problems. It is somewhat paradoxical, then, that prevailing methods to express the complexity of algorithms inhibit our ability to make meaningful guarantees about an algorithm's complexity on real-life problems.

To see this, say we have an algorithm for a problem and wish to understand its time-complexity. Clearly, for every problem instance, there is a specific amount of time that the algorithm needs to solve it[1], so that the only 'complete' description of an algorithms' complexity is a list which contains, for each problem instance, the amount of time the algorithm requires to solve it. However, an infinite list is impossible to create, let alone comprehend.

To overcome this, it has become standard procedure to aggregate all of this information into a single expression. This relies on the fact that (time) complexity is a function of the structure of the problem instance and the algorithm. A main parameter in this function is the *size* of the problem instance. For instance, if an algorithm performs some quadratic search over an input of size n, roughly n^2 time steps are needed [2].

However, parametrising only in the input size n generally doesn't give an exact expression of the complexity: this can happen only if all the instances of a given size n have equal complexity, which virtually never happens. One might therefore try to parametrise in other features of the input to get to an exact expression. But this is generally too difficult, or would give an expression so complicated that it lacks explanatory power (or both).

Lacking an exact expression, the question becomes what kind of *in*exact expression we want. How should we aggregate the complexity of each of the instances for some assignment to the parameters — say, all the instances that have size 16 — into a single expression? Perhaps the most obvious idea is to take an average. However, taking an average requires a distribution

[1] Assuming some model of computation, that is, a formal definition of what an algorithm is. We don't give such a definition in this thesis, as our notion of complexity will not be time, but rather the number of queries an algorithm makes to a function, and this is not dependent on the specifics of the underlying model of computation.

[2] Parametrising complexity in terms of the input size pre-supposes that the algorithm is invariant under input size, i.e. we don't have a completely different algorithm for instances of differing sizes. Though formally this is always the case (non-uniform models of computation like boolean circuits aside), in practice algorithm can behave quite differently on different input sizes. For example, primality testing for odd numbers can be done in constant time, while for even numbers it is much more difficult. The complexity for some input sizes may then be n, while for others it might be n^2. To re-obtain a single-expression, it is usually assumed we aggregate by choosing the worst-case.

over the input instances, making this approach highly context specific, and therefore unpopular. Two universal approaches are to *upper-bound* or *lower-bound* the instances for an arbitrary assignment to the parameters. When using randomised algorithms, there is an additional dimension over which to aggregate: each problem instance might now have multiple complexities for each sequence of "coin tosses". The probability distribution over these sequences is known — it is part of the algorithm — allowing one to compute the average (usually called expected) complexity.

Even for simple algorithms, these bounding expression can become relatively complicated. It has therefore become common to express these bounds *asymptotically*, meaning the expression is equated with its value as the input size tends to infinity, modulo constants. In short: the bound is identified with its fastest growing term: $5/3n^3 + 16(\log\log n)^2 + n + 4\sqrt{n}$ simply becomes *of the order* n^3. Such asymptotic upper and lower-bounds on the (expected) complexity are the prevailing method to express complexity.

In this process we have lost much information regarding the exact complexity of each problem instance. It might be that each instances' complexity is relatively close the derived asymptotic bounds, but in practice we know this often isn't so. For some algorithms, we know that many instances perform far better than our asymptotic upper-bounds suggest, a fact underlying efficient modern heuristics for problems with, say, an exponential asymptotic upper-bound. Conversely, for some algorithms, upper-bounds may harbour large constants and/or significant additive terms, so that instances can perform worse than expected.

One area where this uncertainty is of pressing concern is quantum computing, where quantum algorithms promise impressive speed-ups of classical counterparts. These speed-ups are generally based on asymptotic upper-bounds, so that the real speed-up for a specific problem instance is subject to much uncertainty. While for classical algorithms, the 'gap' between real and asymptotic complexity is easily investigated empirically, this is not feasible for quantum algorithms, as we don't have large fault-tolerant quantum computers (yet).

This motivates the main question we consider in this work: *to what extent can we rigorously estimate the quantum speed-up on specific problem instances, without access to a quantum computer?* To tackle the question, we will "reverse" the above story, and instead of giving asymptotic upper-bounds, *try to derive tightened upper-bounds (and ideally exact expressions) of the complexity of quantum algorithms by parametrising in classically computable input features.* For problem instances of interest, we can then compute these features of the input, yielding a very good estimate — or even exact statement — of the quantum complexity. This will often involve a type of 'mock simulation' of the quantum algorithm, where some subroutines are replaced by classical version, possibly making the approach infeasible for quantum algorithm that promise an exponential speed-up.

1.1 Previous work

The idea of deriving tightened upper-bounds by parametrising in non-trivial features of the input was first proposed and applies in in [CFNW22a, CFNW22b], after an idea of Harry Buhrman. These authors proposed to use the expected query count as their measure of complexity, a measure originally proposed in [BBC$^+$01]. This makes the complexity measure relatively implementation independent. This is important because the precise implementation details of future quantum devices are unknown. In addition, depending on the algorithm, queries can serve as a good proxy for time-complexity.

This choice does mean that the significant complexity overhead suffered by actually implementing quantum operations and memory (mostly due to error-correction) is not taken into account, and this could be multiple orders of magnitude of constant overhead per quantum operation [BMT+22]. Nonetheless, the query model is a useful baseline: if no quantum speed-up occurs here, then it certainly wont occur *after* this overhead is taken into account. Conversely, if a speed-up does occur here and one believes that the constant overhead will come down significantly, this implies that a quantum speed-up may be possible.

The authors [CFNW22a, CFNW22b] derive tightened upper-bounds on the expected query complexity of Grover's search algorithm, a quantum algorithm which can search an unstructured space of size n in $O(\sqrt{n})$ (see Section 2.1.4). They then study two quantum algorithms that use Grover's algorithm as a subroutine, including a Hill climber heuristic for 3SAT [CFNW22a]. The complexity measure is then the number of queries to the 3SAT formula. Comparing this quantum algorithm with a classical version where Grover search was replaced with classical unstructured search, a polynomial quantum speed-up of the order 1.45-1.72 in the number of queries was found, as opposed to the quadratic speed-up in the idealised asymptotic setting. The author note that this was only after extensive optimisation, raising the question of whether achieving quantum speed-ups is perhaps harder than expected.

1.2 Our contributions

We continue this line of work by studying quantum walk search algorithms: a generalisation of Grover search from unstructured space to graphs. We derive exact expressions for the query complexity of several quantum walk search algorithms. Specifically, we do so for Belovs' detection algorithm (the first quantum walk detection algorithm able to start from arbitrary starting distributions), a search algorithm for trees using Belovs' detection algorithm [Bel13], and a general search algorithm due to Piddock [Pid19] (the first general quantum walk search algorithm — a title shared with the independent and concurrent work [AGJ21]). These expressions allow us to compute the exact query complexity of these quantum algorithms on specific problem instances. We also optimally configure Belovs' detection algorithm and experimentally show a speed-up of a factor 10 compared to both Belovs' and Montanaro's proposed configurations in certain settings, slightly optimise Piddock's algorithms, and derive upper-bounds on the inaccuracy of two subroutines of Piddock's algorithm.

We then apply the search algorithms in a quantum backtracking algorithm for constraint satisfaction problems, promising a quadratic speed-up over the corresponding classical backtracking algorithm [Mon16]. Specifically we study uniformly randomly generated 3SAT instances in the critical regime and compare to the classical DPLL algorithm, using our expressions to compute the exact quantum query complexity.

For the detection-based binary search algorithm, we observe a polynomial quantum speed-up of order between 1.36 to 1.56 in the query complexity (depending on the algorithm's configuration), which manifests in 3SAT instances in roughly 330 to 545 variables and on. For Piddock's search algorithm, we find a polynomial speed-up of order at most 1.58, which starts to occur in 3SAT instances in least 430 variables, where we note that Piddock's algorithm assumes access to (an upper-bound on) the size of DPLL's backtracking tree, the computation of which hasn't been accounted for in these numbers. Surprisingly, then, Piddock's algorithm doesn't seem to perform better than the binary search algorithm, despite a clear asymptotic advantage.

As noted above, since these observed speed-ups are in the query complexity model, they

ignore the significant constant overhead suffered by actually implementing quantum operations and memory [BMT+22]. The above results should therefore be interpreted as as showing the *possibility* of a quantum speed-up for solving uniformly random 3SAT instances in the critical regime: if a speed-up wasn't found here, it certainly won't exist after taking further overhead into account.

1.3 Outline of the thesis

We start in Chapter 2 by reviewing preliminary knowledge on quantum computing (Section 2.1), classical random walks, quantum walks and search algorithms (Section 2.2), and (quantum) backtracking (Section 2.3). We note that Section 2.1.4 on Grover's search algorithm and Section 2.2 on walks are not necessary to understand the rest of the thesis, and serve to sketch the intellectual background of the quantum walk search algorithms we study. The uninterested reader can safely skip these roughly 10 pages.

In Chapter 3 we study Belovs' detection algorithm [Bel13]: the first quantum walk search algorithm able to start from an arbitrary distribution over the graph. We motivate Belovs' definitions, break down the proof of correctness of his algorithm while filling in some details, show how to configure Belovs' algorithm optimally, and then give a classically computable and exact expression for its query complexity.

In Chapter 4 we study two search algorithms. The first algorithm applies only to trees, performing binary search while using Belovs' detection algorithm. We give a classically computable and exact expression for its query complexity. The second algorithm is a general search algorithm due to Piddock. This was the first quantum walk search algorithm for general graphs. We motivate the definitions of Piddock's algorithm, filling in several details left by Piddock, give a classically computable exact expression for its expected query complexity, and provide two upper-bounds on inaccuracy of subroutines of the algorithm.

In Chapter 5, we compute the complexity of various configurations of these algorithms on a set of uniformly random 3SAT instances in the critical regime. In Chapter 6 we conclude and sketch future research directions.

Chapter 2

Preliminaries

In Section 2.1 we review preliminary knowledge on quantum computing. In Section 2.2 we review classical random walks, quantum walks and search algorithms based on these walks. Finally, in Section 2.3 we review (quantum) backtracking, including the DPLL algorithm that we will study in Chapter 5.

2.1 Quantum computing

Quantum computing attempts to harness quantum mechanical effects to define novel models of computation which solve certain problems faster than 'classical' models. We give the necessary background to this field in this section. In section

2.1.1 we introduce the quantum mechanical basis for quantum computing, formalised in terms of linear algebra;

2.1.2 we introduce the basic operations of quantum computing;

2.1.3 we introduce our model of quantum computing, and introduce our notion of complexity: query complexity;

2.1.4 we outline Grover's influential search algorithm;

2.1.5 we introduce quantum phase estimation, a quantum algorithm that forms the backbone of the quantum walk search algorithms that we study in this thesis;

2.1.6 we introduce quantum amplitude estimation, which forms an important part of Piddock's quantum walk search algorithm.

2.1.1 Quantum states

Consider some physical system that can be in m states. For example a transistor that can be in two states: 0 or 1. Classically, our description of this system is limited to a probability distribution over the states, the limiting case being a single definite state. A quantum state, by contrast, assigns each state a complex amplitude (a 'superposition') such that the squared norms of these amplitudes induce a probability distribution.

You can do two things with a quantum state. You can observe it, 'collapsing' the superposition to one definitive state according to the induced probability distribution, and you can apply

linear transformations to it, manipulating the complex amplitudes (*not* the induced probability distribution). Since the amplitudes can be negative and/or imaginary, their manipulation can lead to interference effects. This is the novelty that quantum computing provides over classical computing (as probabilities cannot be negative), and the key to its speed-ups over classical computing.

To denote such quantum states formally, we follow the Hilbert space formulation of quantum physics. That is, given some quantum state over m states, we can view the m states as an orthonormal basis for a Hilbert space, so that quantum state are vectors in this space with a Euclidean norm of 1, or equivalently, an inner product of 1.

Definition 2.1.1. Let $\mathcal{H}$ be a complex Hilbert space.

- A *pure quantum state* $|\varphi\rangle$ is a vector in $\mathcal{H}$ with inner product 1.
- The complex conjugate of $|\varphi\rangle$ is denoted by $\langle\varphi|$.
- The inner product of $\langle\varphi|$ and $|\psi\rangle$ is denoted by $\langle\varphi|\psi\rangle$.
- If $|\varphi\rangle$ is a quantum state with components ψ_i, then *observing* (or *measuring*) $|\varphi\rangle$ collapses the quantum states to basis vector i with probability $|\psi_i|^2$.
- We write $|\varphi\psi\rangle$ as a shorthand for $|\varphi\rangle \otimes |\psi\rangle$. We also write $|\varphi\rangle^{\otimes m}$ as a shorthand for the m-fold tensor product $|\varphi\rangle \otimes ... \otimes |\varphi\rangle$.

The notation in the final item is of use as we will generally deal with many quantum states at the same time, and taking their tensor product allows us to write the state of this whole system in a single vector. We will concern ourselves with particular instances of quantum states, namely qubits.

Definition 2.1.2. A *qubit* is a pure quantum state over a two state physical system, that is, a Euclidean norm 1 vector in the Hilbert space spanned by orthonormal basis $\{|0\rangle, |1\rangle\}$. An *n-qubit register* is an n-partite quantum state constructed as the tensor product of n qubits.

Without loss of generality we take $|0\rangle = \begin{bmatrix}1 & 0\end{bmatrix}^T$ and $|1\rangle = \begin{bmatrix}0 & 1\end{bmatrix}^T$. Since we will often work with registers of some variable size n, we won't be able to denote the basis states directly (as we would need to write a variable number of 0s and 1s for this). As a solution, we allow ourselves to denote that 2^n basis states with $|0\rangle, |1\rangle, ..., |2^n - 1\rangle$. Let us consider an example to tie all this together.

Since quantum states are individual entities, one might expect their behaviour to be independent. This turns our not to be the case. Consider the so-called EPR-pair $\frac{1}{\sqrt{2}}|00\rangle + \frac{1}{\sqrt{2}}|11\rangle$. It is not hard to see that measuring either of the states automatically settles the fate of the other: the quantum states have becomes *entangled*. This is a strikingly counter-intuitive feature of quantum mechanics, and a surprisingly useful tool in the hands of the quantum programmer.

2.1.2 Quantum operations

As mentioned above, only linear operations can be applied to transform quantum states into quantum states. However, not all matrices suffice as some might change the Euclidean norm of our quantum state to something unequal to 1. Valid quantum operations are thus matrices A which preserve the Euclidean norm. Recall that a matrix A is called *unitary* if $A^{-1} = A^*$. It is not hard to show that a matrix A preserves norm iff $A^{-1} = A^*$. Quantum operations thus

correspond precisely to unitary matrices. Interestingly, since unitary matrices always have an inverse, operations on quantum states are always reversible: we can undo every operation, and so no information is lost (or created). This contrasts strongly with measurements, which are completely non-reversible. We will call unitary matrices that act on a small number of qubits *gates*, these will be the building blocks of our quantum algorithms. Some often used 1, 2 and 3-qubit gates are:

The bitflit gate X.

Swap $|0\rangle$ and $|1\rangle$.

$X|0\rangle = |1\rangle\,.$
$X|1\rangle = |0\rangle\,.$

$$X = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}.$$

Phaseflip gate Z.

Put a $-$ in front of $|1\rangle$.

$Z|0\rangle = |0\rangle\,.$
$Z|1\rangle = -|1\rangle\,.$

$$Z = \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix}.$$

Phase gate R_φ.

Rotate phase of $|1\rangle$ by φ.

$R_\varphi|0\rangle = |0\rangle\,.$
$R_\varphi|1\rangle = e^{i\varphi}|1\rangle\,.$

$$R_\varphi = \begin{bmatrix} 1 & 0 \\ 0 & e^{i\varphi} \end{bmatrix}.$$

Hadamard gate H.

Set up uniform superposition.

$H|0\rangle = \frac{1}{\sqrt{2}}|0\rangle + \frac{1}{\sqrt{2}}|1\rangle\,.$
$H|1\rangle = \frac{1}{\sqrt{2}}|0\rangle - \frac{1}{\sqrt{2}}|1\rangle\,.$

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

Controlled-not gate $CNOT$.

Negate bit 2 if bit 1 is set.

$CNOT|0\rangle|b\rangle = |0\rangle|b\rangle\,.$
$CNOT|1\rangle|b\rangle = |1\rangle|1-b\rangle\,.$

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

Tofolli gate T (or $CCNOT$).

Negate bit 3 if bits 1 and 2 are set.

$T|1\rangle|1\rangle|b\rangle = |1\rangle|1\rangle|1-b\rangle\,.$
All bits stay the same otherwise.

$$T = \begin{bmatrix} I & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 \end{bmatrix},$$

I is the 4×4 identity matrix.

Note that we can use the Hadamard gate to set up a uniform superposition. As we will often do this for registers of more than 1 qubit, we define the shorthand $H^{\otimes n}$ for the tensor product of n simple 2×2 H gates. This is thus a $2^n \times 2^n$ unitary which maps $|0\rangle^{\otimes n}$ to a uniform superposition

$$\frac{1}{\sqrt{2^n}} \sum_{j \in \{0,1\}^n} |j\rangle\,.$$

It is not hard to see that this matrix also maps $|i\rangle$ to

$$\frac{1}{\sqrt{2^n}} \sum_{j \in \{0,1\}^n} (-1)^{i \cdot j} |j\rangle\,,$$

which will turn out to be very useful.

2.1.3 Quantum circuits

The above naturally lead to a model of quantum computation: given some input n qubit register in state $|\varphi\rangle$, apply any number of quantum gates to (part of) the qubits, with any number of intermediate measurement of (part of) the qubits, and return a measurement of the final state $|\psi\rangle$ of the n qubit register. Such a configuration of gates and measurement is called a *quantum circuit*, and so this model is called the *quantum circuit model* [1]. This model is then relative

[1] Though many other (equivalent) models of quantum computation exist, such as the the quantum Turing machine [Deu85], the cluster-state model [BBD+09] or the adiabatic model [AL18].

to the set of quantum gates provided, begging the question which quantum gates we should implement.

It is not hard to see that no small number of gates suffices to implement every unitary. This may seem surprising, since classically we can implement any boolean function using just two small boolean gates: negation and disjunction (or various other combinations of two). But quantum gates contain complex numbers, and these require in general infinite precision to implement. As such, we require at least infinitely many quantum gates to express every unitary. For example, the set of all infinitely many 1-qubit gates and the CNOT gate can express any unitary. However, this is clearly unreasonable as engineers can't implement infinitely many gates. Our best bet, then, becomes finding a small set of gates which can *approximate* all unitaries well enough. There are two well-known options. First, the CNOT, Hadamard, and phase gate $R_{\pi/4}$ can approximate any 1 or 2-qubit gate arbitrarily well. In fact, the Solovay-Kitaev theorem tells us that exponentially small errors in this approximation require only a polynomial overhead in gates. Second, the Hadamard and Tofolli gate can approximate any unitary with only real entries arbitrarily well [dW22].

As noted in the introduction, our complexity measure will be queries, allowing us to largely disregard the specific model of computation as irrelevant: this serves the specific time or space complexity far more than the query complexity. As such, we don't bother defining quantum circuits more formally than we have already done, in particular ignoring the choice of basic quantum gates. We recall the asymptotic notation used to express complexity.

Definition 2.1.3 (Bachmann–Landau notation)**.** Let $f, g : \mathbb{N} \to \mathbb{N}$ be functions. Then:

- $f \in O(g) : \iff \exists c \exists n_0 \forall n (n > n_0 \to f(n) \leq cg(n))$
- $f \in \Theta(g) : \iff f \in O(g) \wedge f \in \Omega$
- $f \in \Omega(g) : \iff \exists c \exists n_0 \forall n (n > n_0 \to c \cdot g(n) \leq f(n)) \iff g \in O(f)$

where $c \in \mathbb{R}$ and $n, n_0 \in \mathbb{N}$.

We should however briefly dwell on how to implement quantum queries. It is perhaps not obvious how this can be implemented using unitaries. A 'classical' implementation of a query to function f might simply map $|i\rangle \to |f(i)\rangle$. This is clearly not a unitary, as it is not in general reversible (i.e. when f is not bijective). Quantum queries are usually implemented by leaving the index i untouched, and writing the outcomes of the query to a separate qubit (register).

Definition 2.1.4. Let $f : \{0,1\}^n \to \{0,1\}$ be a function. A quantum oracle to f is a unitary O_f such that $O_f : |i, b\rangle \mapsto |i, b \oplus x_i\rangle$ for each $x \in \{0,1\}^n$.

Note that quantum queries can be done in superposition. Often, the target bit b is set to $H|1\rangle = \frac{1}{\sqrt{2}}(|0\rangle - |1\rangle)$. The effect of this is that our register is unchanged if the outcome of the

query is 0, while our register is negated otherwise:

$$\begin{aligned} O_x(|i\rangle \frac{1}{\sqrt{2}}(|0\rangle - |1\rangle)) &= |i\rangle \frac{1}{\sqrt{2}}((|0\rangle \oplus x_i) - (|1\rangle \oplus x_i) \\ &= |i\rangle \frac{1}{\sqrt{2}}(|x_i\rangle - |1 - x_i\rangle) \\ &= |i\rangle \frac{1}{\sqrt{2}}(-1)^{x_i}(|0\rangle - |1\rangle) \\ &= |i\rangle (-1)^{x_i} \frac{1}{\sqrt{2}}(|0\rangle - |1\rangle) \\ &= (-1)^{x_i} |i\rangle \frac{1}{\sqrt{2}}(|0\rangle - |1\rangle). \end{aligned}$$

2.1.4 Grover's quantum search algorithm

We now consider our first example of a quantum algorithm: Grover's search algorithm. In line with the topic of this thesis, it is a search algorithm. But unlike the quantum walks search algorithm that are the object of our study, this search algorithm doesn't search on a structure (like a graph), but rather on an unstructed space. Specifically, Grover's algorithm solves the following problem.

The search problem.
Let $N = 2^n$. We are given an arbitrary $x \in \{0,1\}^N$. The goal is to find an index i such that $x_i = 1$. We denote the number of solutions in x by t (i.e. the number of ones in x).

Classically, the time and query complexity is $O(N)$, as we would need to query all N bits in the worst-case. We can improve this probabilistically to $O(N/t)$. Grover's quantum search algorithm [Gro96] instead requires only $O(\sqrt{N/t})$ queries, and $O(\sqrt{N/t}\log N)$ other gates. We present it below, loosely based on chapter 6 of [NC10] and chapter 7 of [dW22].

Grover's algorithm starts with an n-qubit register $|0\rangle^{\otimes n}$, and query access to x, i.e. a unitary $O_x : i, b \mapsto i, b \oplus x_i$. Call an index $i \in \{0,1\}^n$ *good* if it corresponds to a marked element, and *bad* otherwise. Consider the uniform superposition of indices $|U\rangle = \frac{1}{\sqrt{N}} \sum_i |i\rangle$. It contains bad and good indices. Our algorithm should increase the amplitude of good indices, and reduce amplitude of bad indices. If this is achieved to a sufficient degree, we will have a high probability to observe a good index upon measurement, and solve the problem.

This is easier said than done. One way to simplify things is to limit ourselves to quantum states consisting of a uniform superposition of good states $|G\rangle = \frac{1}{\sqrt{t}} \sum_{i:x_i=1} |i\rangle$ and a uniform superposition of bad states $|B\rangle = \frac{1}{\sqrt{N-t}} \sum_{i:x_i \neq 1} |i\rangle$. The uniform superposition over all indices can then be written as: $|U\rangle = \sin\theta\,|G\rangle + \cos\theta\,|B\rangle$ for $\theta = \arcsin(\sqrt{t/N})$. By limiting ourselves to these two quantum states $|G\rangle$ and $|B\rangle$, all quantum states left are on the unit circle in the 2D plane spanned by $|G\rangle$ and $|B\rangle$.

Say $|B\rangle$ is the horizontal axis and $|G\rangle$ the vertical axis. Our state $|U\rangle$ is somewhere in the first quadrant, its angle being sharper, the less elements are marked. For instance, if 10% of the elements are marked, $|U\rangle$ will form an angle of 9 degrees with the horizontal axis. Our problem now reduces to rotating our state towards the vertical axis: the closer we are to there, the higher the probability to find a marked element when measuring.

We can achieve a rotation towards this axis by applying two reflections: a reflection through $|B\rangle$, and a reflection through $|U\rangle$. Note that a rotation by an angle 2α is equivalent to reflecting

through lines a and b such that the angle between a and b is α. The angle between $|B\rangle$ and $|U\rangle$ is θ so that our proposal constitutes a rotation of 2θ towards the vertical axis.

Reflecting through $|B\rangle$ can be done with the oracle O_x by setting $b = H|1\rangle$: we simply need to negate the phase of all the bad states. We can reflect through $|U\rangle$ with $2|U\rangle\langle U| - I$. Let the *Grover iterate* $\mathcal{G}$ be the unitary performing these two reflections. Each application of $\mathcal{G}$ rotates us 2θ. This brings us iteratively closer to $|G\rangle$, only to overshoot $|G\rangle$ after too many applications. This raises the question of how many applications of $\mathcal{G}$ we should perform.

After k iterations, we are at state $\sin((2k+1)\theta)|G\rangle + \cos((2k+1)\theta)|B\rangle$, so that when we measure, the probability of seeing a solution is $\sin^2((2k+1)\theta)$. We want this to be (as close as possible to) 1, meaning we wish to converge to an angle of $\pi/2$. What is the optimal value k' for which this occurs?

$$\begin{aligned}((2k'+1)\theta = \frac{\pi}{2} &\iff\\ 2k'+1 = \frac{\pi}{2\theta} &\iff\\ 2k' = \frac{\pi}{2\theta} - 1 &\iff\\ k' = \frac{\pi}{4\theta} - 1/2\end{aligned}$$

Unfortunately, $k' = \frac{\pi}{4\theta} - 1/2$ is not always a positive integer. It sometimes is, for example when $t = N/4$, so that $\theta = \arcsin(\sqrt{t/N}) = \arcsin(\sqrt{1/4}) = \arcsin(1/2) = 30$ degrees. This gives us $k' = 1$, as rotating 60 degrees from a starting angle of 30 degrees brings us exactly to $|G\rangle$. In case k' isn't an integer, we can choose the closest integer k, and apply that many Grover iterations. This yields an error probability, but if we assume $t \ll N$, then θ will be small, so that the error probability also becomes small:

$$\begin{aligned}1 - \sin^2((2k+1)\theta) &= \cos^2((2k+1)\theta)\\ &= \cos^2((2k'+1)\theta + (2(k-k')\theta)\\ &= \cos^2(\pi/2 + 2(k-k')\theta) && \text{since } (2k'+1)\theta = \pi/2\\ &= \sin^2(2(k-k')\theta)\\ &\leq \sin^2(\theta) = \frac{t}{N}\end{aligned}$$

How did we derive the final inequality? Since we rounded k' to the nearest integer k, we have $|k-k'| \leq 1/2$. But then $2|k-k'| \leq 1$ so that $2|k-k'|\theta \leq \theta$. Since $\theta \leq \pi/2$, it follows that $\sin(2|k-k'|\theta) \leq \sin(\theta)$. Since we square these results, and since $|\sin(\alpha)| = |\sin(-\alpha)|$ for all α, we can conclude $\sin^2(2(k-k')\theta) \leq \sin^2(\theta)$.

So, we take the rounded value k as the number of iterations. Note that when we express k non-asymptotically, this is not an upper or lower-bound on the number of iterations, as k can be larger or smaller than k' depending on the input. To overcome this, recall that $k' = \frac{\pi}{4\theta} - 1/2$ and that the difference $|k-k'| \leq 1/2$: we can thus fix the following tight-upper-bound: $k \leq \frac{\pi}{4\theta}$. Rewriting in terms of the input size gives:

$$k \leq \frac{\pi}{4\theta} = \frac{\pi}{4\arcsin(\sqrt{t/N})} \leq \frac{\pi}{4\sqrt{t/N}}.$$

Algorithm 1 (Grover search)**.** Input: Oracle access to bitstring $x \in \{0,1\}^N$, i.e. a unitary $O_x : i, b \mapsto i, b \oplus x_i$. Output: An index i, such that the i'th bit in x is set to 1.

1. Set up the starting state $|U\rangle = H^{\otimes n} |0\rangle$.
2. Repeat the following $k = O(1/\sqrt{t/N})$ times.
   (a) Reflect through $|B\rangle$ by applying O_x with $b = H|1\rangle$.
   (b) Reflect through $|U\rangle$ by applying $H^{\otimes n} R_0 H^{\otimes n}$.
3. Measure and return the contents of the first register.

Note that we only query the oracle once per iteration, so that this algorithm indeed has a query complexity of $O(1/\sqrt{t/N})$ as claimed above: a quadratic improvement over the classical optimum. Note that this is quite surprising: if we search for one element among a million elements, this requires only a thousand checks! We leave the claim that the algorithm requires $O(\sqrt{N/t} \log N)$ other gates to the sources cited above.

2.1.5 Phase estimation

We now consider a second example of a quantum algorithm, which will turn our to form the this backbone of the quantum walk search algorithms that we study in this thesis.

Suppose we have a unitary U with eigenvector ψ, and we wish to determine the corresponding eigenvalue $e^{2\pi i\varphi}$ for $\varphi \in [0,1)$[2]. Classically, this simply comes down to computing $U\psi$. Of course, in the quantum setting, this merely results in a quantum state where ψ has amplitude $e^{2\pi i\varphi}$, which we cannot access. To be able to access the eigenvalue, we would have to express the phase φ in binary in a register of qubits.

Let us briefly reflect on how to express numbers $\varphi \in [0,1)$ as bitstrings. Normally, the n bits in a bitstring denote, from most to least significant, $2^{n-1}, 2^{n-2}..., 2^1, 2^0$, so that the bitstring can represent any natural number below 2^n. We can naturally extend this to the interval $[0,1)$ by letting the bits denote $2^{-1}, 2^{-2}, ..., 2^{-n+1}, 2^{-n}$, so that a bitstring can represent any of 2^n uniformly distributed point in $[0,1)$, i.e. the values $k \cdot 2^{-n}$ for $k = 0, 1, ..., 2^n - 1$. Given φ expressed as such and using n bits, we can multiply by 2^n to shift the bits n places to the left to obtained the regular bitstring: $|2^n\varphi\rangle = |\varphi_1, ..., \varphi_n\rangle$. We want an algorithm to compute this state for us. To see how we might do this, consider applying a Fourier transform to this state:

$$F_N |2^n\varphi\rangle = \frac{1}{\sqrt{2}}(|0\rangle + (-1)^{\varphi_1}|1\rangle) \otimes ... \otimes \frac{1}{\sqrt{2}}(|0\rangle + (-1)^{\varphi_n}|1\rangle) = \frac{1}{\sqrt{N}} \sum_{j=0}^{N-1} e^{2\pi i\varphi j} |j\rangle .$$

This state is relatively easy to obtain: apply a Fourier transform to a register $|0\rangle^{\otimes n}$, and then apply U to a second register containing ψ for j times, where j is the number in the first register. If we then apply an inverse Fourier transform to this state, we obtain $|2^n\varphi\rangle$ as desired. Consider the following algorithm:

[2]Note that all eigenvalues of unitary matrices lie on the unit circle, so that we can express it as we did, and we are really only interested in the phase φ.

Algorithm 2 (Phase estimation)**.** Input: Controlled-access to unitary U, eigenvector ψ of U, and precision $n \in \mathbb{N}$. Output: (Estimate of) the phase of eigenvector ψ.

1. Set up the initial state $|0^n\rangle |\psi\rangle$.
2. Apply the Fourier transform F_N with $N = 2^n$ to the first register, resulting in $\frac{1}{\sqrt{N}} \sum_{j=0}^{N-1} |j\rangle |\psi\rangle$.
3. Apply U to the second register controlled by the first register. That is, if j is the number in the first register, apply $|j\rangle |\psi\rangle \mapsto |j\rangle U^j |\psi\rangle = e^{2\pi i \varphi j} |j\rangle |\psi\rangle$, resulting in $\frac{1}{\sqrt{N}} \sum_{j=0}^{N-1} e^{2\pi i \varphi j} |j\rangle |\psi\rangle$.
4. Apply the inverse Fourier transform F_N^{-1} to the first register, and measure and return the first register.

In our example above, the phase can be written exactly in n bits, meaning there is some $k \in \{0, 1, ..., 2^n - 1\}$ such that $\varphi = k \cdot 2^{-n}$. If this is not possible, the inverse Fourier transform results in a superposition over bitstrings, whose amplitude peaks around the best approximation $\widetilde{\varphi}$ (i.e. $k \cdot 2^{-n}$ for $k \in \{0, 1, ..., 2^n - 1\}$ which minimises $2^n \delta := |\varphi - k \cdot 2^{-n}|$). Specifically, the inverse Fourier transform results in the state:

$$|\alpha(\psi)\rangle = \frac{1}{\sqrt{N}} \sum_{x=0}^{N-1} \sum_{j=0}^{N-1} e^{-\frac{2\pi i j}{N}(x - N\varphi)} |x\rangle |\psi\rangle ,$$

which is exactly equal to $|2^n \varphi\rangle$ in case it can be expressed in n bits, and otherwise gives a probability of at least

$$\frac{1}{2^{2n}} \left| \frac{1 - e^{2\pi i N \delta}}{1 - e^{2\pi i \delta}} \right|^2 \geq \frac{4}{\pi^2} \approx 0.405$$

to observe the closest approximation $\widetilde{\varphi}$.

Theorem 2.1.5 (Phase estimation, [CEMM98])**.** Let U be a unitary given as a black box, ψ an eigenvector of U with eigenvalue $e^{2\pi i \varphi}$, and n a natural number. Then Algorithm 2:

1. outputs φ with probability 1 if φ can be expressed in n bits;
2. outputs the best n-bit representation $\widetilde{\varphi}$ of φ with probability at least $4/\pi^2 \approx 0.405$ otherwise;
3. uses n controlled-U^{2^k} operations (i.e. $O(2^n)$ U operations) and $O(m^2)$ other elementary operations.

What if the input vector $|\psi\rangle$ is not eigenvector? Then we can simply consider the eigendecomposition $|\psi\rangle = \sum_i c_i |\psi_i\rangle$ in the eigenbasis of U, and note that we already know the behaviour of the algorithm on each of these eigenvectors. Thus, we obtain a superposition over the outcomes of each eigenvector weighed by the coefficients c_i in the eigendecomposition of ψ:

$$\sum_i c_i |\alpha(\psi_i)\rangle |\psi_i\rangle .$$

Phase estimation, then, 'resolves' arbitrary states ψ across the eigenbasis of a given unitary U, with probability (quadratically) proportional to the relative weight of each of the eigenvectors in the eigendecomposition of ψ. In other words, phase estimation still 'works' when ψ is not exactly an eigenvector ψ_i of U, at the cost of adding some more uncertainty to the algorithm [NC10, p. 225].

2.1.6 Amplitude estimation

Amplitude estimation is a quantum algorithm that estimate the probability a that a given quantum circuit with given starting state gives a particular (set of) state(s) as outcome.

Theorem 2.1.6 (Amplitude Estimation, [BHMT02], Theorem 12)**.** Let U be a unitary on m qubits and let $U|0^m\rangle = |\psi\rangle = \sin\theta\,|\psi_{\text{good}}\rangle + \cos\theta\,|\psi_{\text{bad}}\rangle$. For every positive integer s there exists a uniformly generated quantum circuit C acting on $s+m$ qubits such that

1. measuring the first register of $C|0^s\rangle|\varphi\rangle$ results in an integer $y \in \{0,1,...,2^s-1\} = [2^s-1]$, corresponding to an estimation $\tilde{\theta} = \pi y/2^s$ of θ, and therefore an estimate $\tilde{a} = \sin^2(\tilde{\theta})$ of the success probability $a = \sin^2(\theta)$;
2. the probability that this estimate is optimal is high: $\Pr(|a-\tilde{a}| \leq \epsilon_a) \geq 8/\pi^2$, where
$$\epsilon_a = \frac{2\pi\sqrt{a(1-a)}}{2^s} + (\frac{\pi}{2^s})^2 \leq \frac{\pi}{2^s} + (\frac{\pi}{2^s})^2.$$
3. C uses s controlled-U^{2^k} operations (i.e. $2^s - 1 \in O(2^s)$ U operations) and $O(s^2)$ other elementary operations.

'Under the hood', amplitude estimation is estimating the angle θ, which is then translated to an estimate $\tilde{a} = \sin^2(\tilde{\theta})$ of a. Using s bits, we estimate θ with a precision of $\pi/2^s$, so that the best estimate of θ (which we output with probability at least $8/\pi^2$) is the nearest $\pi k/2^s$ for an integer $k \in [2^s-1]$. It follows that the difference between θ and our estimate is at most $\epsilon_\theta \leq \pi/2^s$. Translating this error range to a then gives us (see e.g. [BHMT02, Lemma 7])

$$\sin^2(\theta+\epsilon_\theta) - \sin^2(\theta) = \sqrt{a(1-a)}\sin(2\epsilon_\theta) + (1-2a)\sin^2(\epsilon_\theta),$$
$$\sin^2(\theta) - \sin^2(\theta+\epsilon_\theta) = \sqrt{a(1-a)}\sin(2\epsilon_\theta) + (2a-1)\sin^2(\epsilon_\theta).$$

To derive the upper-bound in the lemma, note that $\sin(x) < x$ for small x, so we are safe to remove the sin functions. Note that both $1-2a$ and $2a-1$ are at most 1, so we can remove this factor from the second term. This then yields the upper-bound $\sqrt{a(1-a)}2\epsilon_\theta + \epsilon_\theta^2$. By finally noting that $\sqrt{(a(1-a))}$ is at most $1/2$ (when $a = 1/2$), we get the upper-bound $\epsilon_a \leq \epsilon_\theta + \epsilon_\theta^2$. When we need to use ϵ_a within our algorithms, we will make use of this final upper-bound, as we have no access to a (this is what we are estimating!) within the algorithm.

Sometimes amplitude estimation is phrased using an additional variable k such that:

$$\Pr(|\theta - \tilde{\theta}| \leq k\pi/2^s) \geq 1 - \frac{1}{2(k-1)} = 1 - \frac{1}{2(k2^s/\pi - 1)},$$

phrased in terms of θ for convenience. The utility of this lies in being able to set the error range yourself, and having the error probability increase exponentially with s. However, one can easily see that for $k < 6$ the success probability is lower than $8/\pi^2$. At the same time, the error range widens significantly. It begs the question whether this method does pay off for larger k.

That is, say we wish to bound the error range in our amplitude estimates to at most δ, and say that we have found s so that $\pi/2^s \leq \delta$. If we move to $k \geq 2$ we then need to find s' such that $k\pi/2^{s'} \leq \delta$. Solving $k\pi/2^{s'} = \pi/2^s$ gives $k = 2^{s'-s}$ so that incrementing k adds one bits of precision, and therefore doubles the number of queries. This shows that increasing k to increase the success probability is not optimal: a linear decrease of the success probability requires an exponential increase in queries, whereas Chernoff's bound tells us that simply repeating the $k = 1$ algorithm nets an exponential decrease of the success probability with a linear increase in queries (repetitions).

In our applications of amplitude estimation, then, we move forward with the $k = 1$ algorithm.

2.2 Random and quantum walks

In this section we sketch the intellectual background of the quantum walk algorithms that we study. In section

2.2.1 we introduce classical random walks or Markov chains;

2.2.2 we introduce their quantum counterparts;

2.2.3 we consider how these walks can be used in search algorithms.

2.2.1 Random walks

Informally, a random walk is a stochastic process which describes a path on some mathematical space as a succession of random steps. For example, the following describes a random walk on the integers: start at 0, and successively move $+1$ or -1 with equal probability. Random walks on graphs are an instance of a more general type of stochastic process: Markov chains.

Definition 2.2.1. A discrete-time stochastic process $x = (X_0, X_1, ...) = (X_t)_{t \in T}$ on a state space V with $|V| = n$ and time space T is a discrete-time *Markov chain* if for all t: $\Pr(X_t = v_t | X_{t-1} = v_{t-1}, ..., X_0 = v_0) = \Pr(X_t = v_t | X_{t-1} = v_{t-1})$.

We will only consider Markov chains on finite state-spaces, and so allow ourselves to drop the prefix "finite state". The equation in the definition is called the *Markov property*. It says that proceeding from state v_{t-1} to state v_t depends only on v_{t-1}, and not on further proceeding states. As such, the behaviour of a Markov chain is fully captured by an $|V| \times |V|$ matrix P such that $P(i, j) = \Pr(X_t = v_i | X_{t-1} = v_j)$ for all $t \in T$. We can (and will) thus equate a Markov chain with its *transition matrix* P. To be complete, we should also specify some initial distribution for X_0. This in turn fixes the distribution for each subsequent random variable. Note that we can describe a distribution over state space V with a vector $\sigma \in \mathbb{R}^n$. Let $\Pr_{\sigma_0}(X_t = v_t)$ denote the probability that $X_t = v_t$, given that X_0 is distributed according to σ_0. It is easy to see

that given σ_0, we can compute $\sigma_1 = \sigma_0 P$, and more generally, $\sigma_t = \sigma_0 P^t$[3]. We can now define random walks.

Definition 2.2.2. Let $G = (V, E)$ be a simple undirected graph with edge weights $w : E \to \mathbb{R}_{\geq 0}$. A *random walk* on G is a Markov chain with state space V and transition matrix $P(i, j) = w((v_i, v_j))/w(v_i)$, where $w(v_i)$ is the sum of weights of edges incident to v_i.

It is not hard to see that generalising the above definition to directed graphs makes random walks and Markov chains equivalent: there is a one-to-one correspondence between such random walks and (finite state discrete-time) Markov chains [Lov93]. The class of random walks as defined here are thus a subclass of all Markov chains[4]. We call a distribution π *stationary* for P if $\pi = \pi P$. One of the most fundamental theorems in the study of Markov chains is that most Markov chains have a unique stationary distribution that they converge to. Specifically, all *irreducible* and *positive-recurrent* Markov chains have a unique stationary distribution $\pi(i) = 1/t_i$, where t_i is the expected time for P starting in state v_i to walk back to v_i [LPW$^+$17, Proposition 1.19]. A Markov chain is irreducible if any two states can reach each other, and positive-recurrent if t_i is finite for every state v_i[5]. Markov chains that are also *aperiodic* converge to the unique stationary distribution from every starting distribution [LPW$^+$17, Theorem 4.9][6] A Markov chain is aperiodic if the *period* of each state is 1, where the period of a state x is the greatest common divider of all the times t when x can return to itself.

It is well known that irreducible Markov chains are equivalent to random walks on fully-connected graphs, and aperiodic Markov chain are equivalent to random walks on non-bipartite graphs. Note also that *symmetric* Markov chains (i.e. having a symmetric transition matrix) are equivalent to random walks on regular graphs [LPW$^+$17, AGJ21]. Thus, a random walk on a fully-connected graph has a unique stationary distribution, and if the graph is non-bipartite, the random walk converges to this unique stationary distribution from every starting distribution. It is easily seen that the distribution π with $\pi(i) = \sum_{(v_i,v_j)\in E} w((v_i, v_j))/2W$, where $W = \sum_{e\in E} w(e)$ is the sum of edge weights, is stationary for all random walks.

This stationary distribution has great algorithmic importance. For example, note that the distribution is similar to the uniform distribution (indeed, for regular graphs it *is* the uniform distribution). By deriving the expected time to converge, we can effectively sample from the uniform distribution simply by taking a number of random walk steps; something which can be done even in situations with only local access to a graph. For another example, if you wish to detect whether a graph has a certain property, you might be able to show that the stationary distribution will differ depending on whether the graph has this property, so that repeated applications of the random walk will allow you to detect this. This fact turns out to be the basis for quantum walk search algorithms, which we will study a bit later.

A final definition we will need is the *hitting time*. The *hitting time* is a random variable $\tau_{v,M}$ representing the first time t that a random walk starting at vertex v reaches or 'hits' a

[3]To see this, note that a row i of P is a probability distribution denoting the probability to traverse from state v_i to each of the other states v_j. Multiplying u with P then computes a linear combination of these probability distributions, weighed by the current probability distribution u.

[4]Specifically, the class of *time-reversible* Markov chains. A Markov chain is time-reversibile if $\Pr(X_t = v_t, X_{t+1} = v_{t+1}) = \Pr(X_t = v_{t+1}, X_{t+1} = v_t)$ for all $t \in T$. This should make some intuitive sense: undirectedness of the graph means that we can always walk back along an edge.

[5]Note that finite state Markov chains are always positive-recurrent, so we can ignore this.

[6]The term *ergodic* is often used to describe irreducible and aperiodic matrices, so that this theorem is often stated for ergodic matrices, which can be slightly confusing if the finiteness of the Markov chain is implicit.

vertex in M. Formally, $\tau_{v,M} : \Omega \to \mathbb{N}$ is defined by $\tau_{v,M}(\omega) = \min\{n \in \mathbb{N} \mid X_n(\omega) \in A | X_0 = v\}$. Somewhat confusingly, in the literature, 'hitting time' often refers to the expected value of this random variable. Since this is the quantity that will be of most interest to us as well, we will follow this convention. So let us define the notation $HT(v, M) := \mathbb{E}(\tau_{v,M})$, and $HT(\sigma, M) = \sum_{v_i \in V} \sigma(i) HT(v_i, M)$.

2.2.2 Quantum (random) walks

In classical random walks, though the walker is always in a definite state, randomness arises due to the stochastic transitions between states. This is not possible for quantum algorithms as unitary operations are invertible, and therefore not stochastic; the only randomness that can arise in a quantum algorithm is the collapse of a superposition. We therefore usually speak of a *quantum walk*, omitting the word 'random'.

A necessary requirement for any quantum walk is then to extend our state to make steps between (superpositions of) vertices reversible. Clearly, being able to reverse a step from, say, vertex v to neighbour u requires remembering what v was. One way to achieve this would be to extend our state to encompass two vertices, so that we include all possible transitions $v \to u$ as basis states, giving us the Hilbert space $\mathbb{C}^{|V|} \otimes \mathbb{C}^{|V|}$ with computational basis $\{|a, b\rangle \mid a \in V, b \in B\}$. Of course, this is overkill, as some pairs of vertices might not have edges between them. We could therefore also restrict to the actual 'walk space' of the walk, i.e. the edges in the graph, which would yield the Hilbert space $\mathbb{C}^{\sum_{v \in V} \deg(v)} = \mathbb{C}^{2|E|}$ with computational basis $\{|a, b\rangle \mid a \in V, b \in B, a \sim b\}$. To simplify presentation, we will consider the full $\mathbb{C}^{|V|} \otimes \mathbb{C}^{|V|}$ space, restricting implicitly to the walk space.

Given a basis-state $|v, u\rangle$, we can now interpret v as the current vertex, and u as the vertex we will step to in the next time step (or vice-versa). How can we now implement a walk? Suppose C is a unitary which, for each vertex, sets up a superposition over the neighbours. Let S be a unitary which 'steps' from the current vertices to the neighbours specified in the superposition. Then $W = SC$ can be interpreted as taking one walk step. If we measure after applying C, we collapse the superposition over neighbours to just one neighbour, and S then moves us to this one neighbour, resulting in a classical random walk. But if we don't measure and choose the right C, we could attain interference effects, leading to truly 'quantum' walks.

To see this clearly, consider an example of the unitaries S and C. Let S be the *flip-flop shift* defined by $S|v, u\rangle = |u, v\rangle$ for all $v, u \in V$. Let C be the Grover diffusion operator:

$$C = 2 \sum_{v \in V} |\varphi_v\rangle \langle\varphi_v| - I,$$

where

$$|\varphi_v\rangle = \frac{1}{\sqrt{\deg(v)}} \sum_{u \sim v} |v, u\rangle .$$

That is, for each vertex v, C reflects the basis states $|v, u_i\rangle$ (where u_i's are the neighbours of v) through the uniform superposition $|\varphi_v\rangle$. In effect, this inverses the amplitude of a basis state $|v, u_i\rangle$ around the mean amplitude of basis states having v as current vector. Note that this is not the same as a uniformly spreading the sum of amplitude of basis states having v as current vector over all v's neighbours (i.e. a classical random walk), and in particular this method can lead to negative amplitudes, despite the total amplitude being preserved.

Example 2.2.3. Suppose our graph includes the vertices v, u_1, u_2 and u_3, where v is connected to all vertices u_i. Suppose that the amplitude at $|v, u_1\rangle$ and $|v, u_2\rangle$ is $1/20$ and the amplitude

at $|v, u_3\rangle$ is $1/2$. The sum of amplitude at vertex v is then $12/20$, while the mean over the basis states having a as current vector is $4/12$.

Applying C, we invert the basis states' amplitude around this mean, so that the amplitude at $|v, u_1\rangle$ and $|v, u_2\rangle$ becomes $7/20$ while the amplitude at $|v, u_3\rangle$ becomes $-2/20$. Now applying S, the amplitude at $|u_1, v\rangle$ and $|u_2, v\rangle$ is $7/20$ and the amplitude at $|u_3, v\rangle$ is $-4/20$.

This is called a *coined quantum walk*, a name derived from the unitary C often being called the "coin", and the additional vertex register begin called the "coin register". This is the first type of quantum walk proposed. It was introduced in [ADZ93], extended to arbitrary graphs in [AAKV01], and used as basis for the first quantum walk search algorithm in [SKW03]. Note that both S and C can be different from the flip flop shift and Grover coin. For example, the Hadamard gate is often used as a coin, while in d-regular graphs, there is often a "coin" register of fixed size d, so that the shift from $|i, v\rangle$ simply selects the i'th neighbour of $v : |i, u_i\rangle$.

Shortly after, Szegedy proposed what came to be called the *Szegedy quantum walk* [Sze04]. He defines a walk unitary $W(P)$ on a classical Markov chain P, the idea being that this is a direct quantisation of P. His walk turns out to be equivalent to two applications of the coined quantum walk as defined above, i.e. with Grover "coin" and flip-flop shift. The walk takes place on the edges of a bipartite graph, but if the given Markov chain is a random walk on a non-bipartite graph G, we can simply use its bipartite double cover $G \times {K_2}^7$. Thus, if we consider an arbitrary graph $G = (V, E)$, the walk takes place on $G \times K_2$ with bipartite sets A and B such that $A = B = V$, so that we have the same Hilbert space $\mathbb{C}^{|V|} \otimes \mathbb{C}^{|V|}$ as with the coined quantum walk, with the same computational basis $\{|a, b\rangle \mid a \in V, b \in B\}$.

Szegedy defines his walk as $W(P) = R_2 R_1$ where R_1 and R_2 are Grover diffusion operators, with R_1 reflecting from X to Y, and R_2 from Y to X:

$$R_1 = 2 \sum_{v \in X} |\varphi_v\rangle \langle\varphi_v| - I,$$

$$R_2 = 2 \sum_{u \in Y} |\psi_u\rangle \langle\psi_u| - I,$$

where

$$|\varphi_v\rangle = \frac{1}{\sqrt{\deg(v)}} \sum_{v \sim u} |v, u\rangle,$$

$$|\psi_u\rangle = \frac{1}{\sqrt{\deg(u)}} \sum_{u \sim v} |v, u\rangle.$$

Note that $R_2 = S R_1 S$ (where S is the flip-flop shift): R_1 and R_2 differ only in the direction in which they consider edges, so that swapping the edges before and after applying R_1 results exactly in R_2. As already noted above, the computational bases of the coined and Szegedy walk coincide: they each have one basis state for each directed edge $|vu\rangle$. We can thus easily bring these into a one-to-one correspondence, and it is not hard to see that the result is that R_1 and C (the Grover coin) coincide, so that one step of Szegedy's walk $W_S = R_2 R_1 = S R_1 S R_1$ is then equivalent to two steps of the coined walk W with Grover "coin" C and flip-flop shift S [Won17].

[7] Note that K_2 is the complete graph on 2 vertices, and that $\times$ is the Cartesian product on graphs, so that the result is two copies - X and Y - of G's vertex set V, with $x \sim y$ for $x \in X$ and $y \in Y$ if and only if $x \sim y$ in the original graph G.

One main advantages of Szegedy's walk, and also one of the main motivations of Szegedy, is its direct quantisation of Markov chains. Note that in our definition of φ_v and ψ_u we 'baked in' a classical random walk, by adding uniform amplitudes $1/\sqrt{\deg(v)}$ over edge neighbours. Szegedy defined φ_v and ψ_u relative to a Markov chain P as follows:

$$|\varphi_v\rangle = \sum_{u \in Y} \sqrt{P_{v,u}} \, |vu\rangle = \sum_{v \sim u} \sqrt{P_{v,u}} \, |vu\rangle$$

and likewise for ψ_u. Not only does this allow direct quantisation of classical algorithms involving Markov chains, Szegedy showed it also allowed analysis of properties of W_S (for instance, hitting time) in terms of (the spectrum of) the underlying Markov chain P.

2.2.3 Searching with walks

Recall that Grover's algorithm (see section 2.1.4) solved a search problem on an unstructured search space. We now consider a structured search problem (on a graph), and see how we can use a classical random walk to solve it. Then we consider Szegedy's quantum walk search algorithm, which solves the detection problem quadratically faster than this classical algorithm.

As noted in the introduction, Szegedy's algorithm suffers two main limitations: it has to start from a specific distribution over the entire graph, and only detects whether a marked vertex exists. In Chapter 3 we study Belovs' generalisation of Szegedy's algorithm which allows arbitrary starting distribution (this allows backtracking, which must start from the root of a tree), and in Chapter 4 we study Piddock's generalisation of Belovs' algorithm which also finds a marked vertex.

The graph search problem.
Let $G = (V, E)$ be an undirected graph. Let $M \subseteq V$ be a set of marked vertices. The goal is to determine whether $M \neq \emptyset$ (decision version), or to find some $v \in M$ (search version).

Classical random walk search A classical random walk makes easy work of this problem: starting from some distribution σ, simply walk, check if the vertices you encounter are marked, and return the first marked element you find. The name "hitting time search" is fitting, as the runtime of this algorithm comes down to the hitting time: after that many steps, we expect to have hit a marked element, so that if we haven't, it is safe to conclude no marked elements exist.

Theorem 2.2.4 ([MNRS07])**.** The classical 'hitting time' search algorithm detects the presence of a marked vertex with high probability, and if so finds it, with complexity

$$S + HT(\sigma, M)(U + C),$$

where S is the cost to set up the initial state v, U the cost to perform a random walk step, and C the cost to check whether a vertex is marked.

'Cost' is left intentionally undefined, so as to cover whatever measure of complexity currently of interest. In our case, it will be expected query complexity. This algorithm is optimal in the number of walk steps: the algorithm finds the very first marked element it comes across. It is not optimal in the number of checks, however. An alternative would be to walk until you converge

to the stationary distribution, and only then check whether the current vertex is marked. The algorithm then effectively samples from the stationary distribution, and it can be shown that this algorithm *is* optimal in the number of checks, though clearly not in the number of walk steps [San08][8].

Szegedy's quantum walk search A quantisation of the above 'hitting-time' algorithm was introduced by Szegedy [Sze04], and uses his quantum walk as defined in section 2.2.2. Although his quantisation of Markov chains admitted arbitrary Markov chains, his search algorithm is only defined for irreducible, aperiodic (i.e. ergodic) and symmetric Markov chains, where we recall that this is equivalent to a random walk on a fully-connected, non-bipartite and regular graph. He also only considers unweighed graphs. His algorithm solves the decision problem for all such graphs quadratically faster than the classical algorithm outlined above.

Classically, the 'hitting time' search algorithm works because the random walker is always in a definite vertex: if the hitting time is $HT(\sigma, M)$ and we check each vertex we come across, we expect to find some marked vertex after this many steps. But the quantum walk $W(P)$ is in a superposition over vertices (rather, over edges), and only after collapsing this superposition can we check if the problem is solved. This means hitting time is not a quantum notion, as we don't truly hit the set M with the current vertex, but at most assign a significant portion of amplitude to vertices in M. We could try to define a 'quantum hitting time' along these lines ("expected number of steps before we assign at least x amplitude to marked vertices"), but this wouldn't help us, as this doesn't mean at all that collapsing the superposition after t steps gives high probability of hitting M. It would only mean that after t steps, one of the previous superpositions is expected to yield a marked vertex if we had measured it. In short, the quantum walk is unable to detect whether a current superposition is adequate without collapsing it, and so the walk will walk past decent superpositions.

Szegedy's solution is the alter the underlying classical walk P into P' where outgoing edges from marked vertices are replaced with a self-loop. The walk is then unable to leave marked vertices, so that as time goes on, the probability to observe a marked vertex increases monotonically. This is called the *absorbing walk* of P. We can now define a quantum analogue of hitting time corresponding to the first time when a significant amount of amplitude has been shifted to marked vertices.

To define this, consider first how we might express the hitting time $HT(\pi, M)$ of P in terms of P'. Note that, whenever there are marked vertices $M \neq \emptyset$, the absorbing walk P' will have a different stationary distribution from P (as it behaves differently on marked vertices). In other words, $\pi P'^t$ will at some point start diverging significantly from π, specifically when we first hit a marked vertex, i.e. for $t = HT(\pi, M)$, as this is the time when we expect to first hit a marked vertex. Thus, the smallest t for which $|\pi P'^t - \pi|$ grows significantly large roughly coincides with $HT(\pi, M)$ of P.

Quantising this, we can take the smallest t for which $|\varphi_0 W(P')^t - \varphi_0|$ grows significantly large as a measure for quantum hitting time, where

$$\varphi_0 := \frac{1}{\sqrt{n}} \sum_{x,y \in V} \sqrt{P(x,y)}\, |x, y\rangle\,,$$

is a natural choice for a starting state, as it's invariant under $W(P)$ (though $W(P)$ does not

[8]Though we limit ourselves in this thesis to hitting time search algorithms, there are also quantum versions of this alternative algorithm which samples from the stationary distribution. See [MNRS07].

converge to it), but not under $W'(P)$. Let us now consider Szegedy's quantum hitting time definition and its relation to classical hitting time. Let P_M denote the transition matrix of Markov chain P with rows and columns indexed by vertices in M removed.

Definition 2.2.5. For quantised walk W_P and marked set M we define the hitting time of W_P with respect to target set M as the smallest number of steps $\mathcal{HT}(W_P, M) = T$ for which

$$\frac{1}{T+1}\sum_{t=0}^{T}\left|W(P')^t\varphi_0 - \varphi_0\right|^2 \geq 1 - \frac{|M|}{|V|}.$$

Lemma 2.2.6 ([Sze04], Lemma 6)**.** Let $\mathbf{v}_1, ..., \mathbf{v}_{n-|M|}$ be the normalised eigenvectors of P_M with associated eigenvalues $\lambda_1, ..., \lambda_{n-|M|}$. For $\hat{u} = \frac{1}{\sqrt{n}}\mathbf{1}_{n-|M|}$, define coefficients $v_1, ..., v_{n-|M|}$ such that $\hat{u} = \sum_{k=1}^{n-|M|} v_k\mathbf{v}_k$. Then:

$$\mathcal{HT}(W_P, M) \leq \frac{100}{1-|M|/|V|}\sum_{k=1}^{n-|M|} v_k^2\sqrt{\frac{1}{1-\lambda_k}}.$$

Let P_M be the matrix obtained by removing columns and rows of P indexed by some $x \in M$, and similarly Σ_M for a probability distribution vector σ. Then we can write:

$$HT(\sigma, M) = \sigma_M(I - D(P)_M)^{-1}\mathbf{1},$$

where $\mathbf{1}$ is the all-ones vector [Sze04, Equation (1)]. This expression allows an elegant decomposition using the spectrum of P_M, which is exactly the content of the square root in the above upper-bound: TODO

Corollary 2.2.7 ([Sze04], Corollary 1)**.** For every irreducible, aperiodic (i.e. ergodic) and symmetric Markov chain P with state space X, and for every set $M \subseteq X$ with $|M| \leq |X|/2$, the quantum hitting time is of the order of the square root of the classical hitting time: $\mathcal{HT}(W_P, M) \in O(\sqrt{HT(\pi, M)})$.

How do we use this result to create an algorithm for the detection of marked vertices? Note that in this detection problem, we are effectively given a walk unitary W and wish to determine whether $W = W(P)$ (there are no marked elements) or $W = W(P')$ (there are marked elements). We know from above that φ_0 is invariant under $W(P)$, while φ_0 diverges significantly within $\mathcal{HT}(W_P, M)$ under $W(P')$. By applying the walk W $\mathcal{HT}(W_P, M)$ many times, we can thus detect which setting we are in. We might also find a marked vertex, but note that we don't need to, and so we might detect a marked vertex without finding it.

Theorem 2.2.8 ([Sze04], Lemma 7)**.** Let T be an upper-bound for

$$200 \sum_{k=1}^{n-|M|} v_k^2 \sqrt{\frac{1}{1-\lambda_k}}.$$

Szegedy's quantum hitting time search algorithm determines whether a marked element exists with complexity

$$O(S + T(U + C)) = O(S + \sqrt{HT(\pi, M)}(U + C)),$$

where S is the cost to set up the initial state φ_0, U the cost to apply the quantum walk unitary, and C the cost to check whether a vertex is marked.

Szegedy also showed that in case the graph is state-transitive we can find marked elements through binary search, however, since subsequent algorithms have already solved the search problem for general cases, we won't bother discussing Szegedy's proposal here.

2.3 Classical and quantum backtracking

In section

2.3.1 we define classical backtracking algorithms;

2.3.2 we introduce Montanaro's quantum backtracking algorithm;

2.3.3 we introduce DPLL, a classical backtracking algorithm for SAT.

2.3.1 Classical backtracking

Backtracking is a common technique for solving constraint satisfaction problems (CSP) like SAT. Such problems can be described as follows: given a predicate $P : [d]^m \to \{\bot, \top\}$ where $[d] = \{0, .., , d-1\}$, find an assignment x to the m variables such that $P(x)$ is true, or output "not found" when no such x exists. For some such problems, we have the ability to recognise whether a partial assignment can be extended to a complete solutions, SAT being a prime example. This often allows one to 'prune' the search tree at a high level, cutting off potentially many lower branches of the tree. Let $\mathcal{D} := ([d] \cup \{*\})^m$ where $*$ is interpreted as a blank, so that $x \in \mathcal{D}$ are assignments, where x is complete if it contains no $*$'s. A general classical backtracking algorithm is then as follows.

Algorithm 3 (General classical backtracking)**.** Input: Access to predicate $P : \mathcal{D} \to \{\top, \bot, \text{indeterminate}\}$, heuristic $h : \mathcal{D} \to \{1, ..., n\}$ which returns the next index to branch on from a given partial assignment, and some assignment $x \in \mathcal{D}$. In the initial run, $x = (*)^m$. Output: An assignment $x \in \mathcal{D}$ such that $P(x) = \top$ if it exists, $\bot$ otherwise.

1. If $P(x)$ is true, output x and return.
2. If $P(x)$ is false, or x is complete, return.
3. Set $j := h(x)$.
4. For each $w \in [d]$:
   (a) Set y to x with the j'th entry replaced with w.
   (b) Call this algorithm with assignment y.

2.3.2 Montanaro's quantum backtracking

Since the backtracking algorithm generates the search tree on the fly, neither Grover search nor Szegedy's quantum walk search will be of much help. Both require generating the entire backtracking tree before starting the search, defeating the purpose of backtracking in the first place.

If we would be able to start a quantum walk from the root vertex of the backtracking tree - as we do classically - we would be able to obtain a quadratic speed-up. This is what Montanaro did [Mon16], making use of Belovs' quantum walk search algorithm that could start from arbitrary starting distributions. We will define Belovs' algorithm in Chapter 3, for now taking for granted that it is able to start from any starting distribution.

Montanaro then showed how the quantum walk unitary $R_B R_A$ can be implemented from the black box operations h and P, so that we can use Belovs' algorithm on a classical backtracking tree. In particular, this implementation comes at no extra query cost, so that we can simply study the query complexity of Belovs' and Piddock's quantum walk algorithms, in the knowledge that we can apply them to speed-up classical backtracking algorithms at no extra cost. We assume, as Montanaro does, that we can apply $R_B R_A$ with one query to h and one query to P.

Montanaro restates the complexity $O(\sqrt{RW})$ of Belovs' algorithm in terms of the total nodes in the tree T (equal to W) and the maximum depth of the tree m (upper-bound on $R_{\sigma,M}$). The speed-up over classical backtracking algorithm that he then shows is as follows.

Theorem 2.3.1 ([Mon18], Theorem 1)**.** Let T be an upper-bound on the number of vertices in the tree defined by a classical backtracking algorithm. For any $\delta \in (0, 1)$ there exists a quantum algorithm which, given T, determines with probability at least $1 - \delta$ whether there exists an x such that $P(x) = 1$, and evaluates P and h $O(\sqrt{Tn}\log(1/\epsilon))$ times each.

Importantly, the algorithm requires and an upper-bound T, and this T upper-bounds the size of the entire backtracking tree, not just the size of the tree after a classical backtracking

algorithm finds its first solution (which may be much smaller than the full size).

2.3.3 DPLL algorithm for SAT

Recall that in SAT, we are given a propositional formula, and are asked whether it has a satisfying assignment. Since every propositional formula can be written in conjunctive normal form (CNF) (i.e. a conjunction of disjunctions of (negated) propositions), and in particular in 3-CNF (a conjunction of disjunctions of exactly 3 (negated) propositions), the problem is usually studied for such formulas. In this case, it is referred to as $k-\mathsf{SAT}$. Usually $k=3$ is taken, as for $k \geq 3$ the problem is NP−complete; for $k=1$ or $k=2$ the problem is solvable in polynomial time.

This problem is of course a clear instance of the general CSP outlined above. If we set $d=2$, so that each variable has two options, the predicate $P : 0,1^n \to \bot, \top$ which defines the given CSP simply is our propositional formula. We can also easily instantiate the general backtracking algorithm outlined above: we can implement $P : \mathcal{D} \to \{\top, \bot, \text{indeterminate}\}$ by noting that we can evaluate formulas with partial assignments, which can sometimes tell us whether the formula is satisfiable or not. We can of course implement a heuristic $h : \mathcal{D} \to \{1, ..., n\}$ however we wish, but the key to efficient solving is to exploit the structure of the formula as much as possible to pre-emptively cut off branches of the search space.

The best known algorithm that does this is the DPLL algorithm [DP60, DLL62]. Its heuristic has two main parts: it seeks unit clauses (i.e. clauses containing a single literal) as these must be set to satisfy whatever is in them; and it seeks pure literals (i.e. variables occurring with one polarity) as these can safely be satisfied. Its heuristic first seeks unit clauses and pure literals, and satisfies these. Only after this does it select a variable to recurse on.

Chapter 3

Detection

In this chapter, we study Belovs' quantum walk search algorithm [Bel13]. Belovs' algorithm was the first algorithm able to detect the presence of a marked element in a graph in most the square root of the classical hitting time, while starting from an arbitrary starting distribution. In section

3.2 we motivate and introduce Belovs' definitions;

3.2 we break down and fill in various detail in Belovs' proof of the correctness of his algorithm;

3.3 we give an exact expression for the query complexity of the algorithm, and consider two method to amplify the success probability of the algorithm;

3.4 we specify the optimal combination of these two amplification methods, in turn yielding an optimal configuration of the algorithm in general, improving over the configurations used by Belovs' and Montanaro [Mon16].

3.1 How to quantum walk from arbitrary starting distributions

A major drawback of Szegedy's quantum walk search algorithm (as seen in Section 2.2.3) is its requirement to start from the particular initial state φ_0: it relies on the this state being invariant (i.e. an eigenvalue-1 eigenvector) under the walk operator in case there are no marked elements, while not being invariant in case there are marked elements. Repeated applications of the walk operator then reveals which is the case

How could this be generalised to arbitrary starting distributions σ? Clearly, not every starting distribution has the property of being invariant if and only if there are (no) marked elements. When some σ is not invariant the best we could do is to find some σ' which does have this property, and which significantly overlaps with σ. Repeated applications of the walk to σ can then reveal which is the case. However, as the distance between σ and σ' grows, the change to σ due to this distance could overshadow the change caused by marked vertices existing. A more error-prone method then becomes quantum phase estimation (see 2.1.5), repeated applications of which can resolve σ to an eigenvalue-1 eigenvector (no marked vertices) or some other eigenvector, with high probability.

Does a suitable σ' always exist? The best candidate would be the projection of σ onto the invariant space of the walk operator. But this is clearly too limited to allow arbitrary starting distributions: just consider a starting state that is orthogonal to this invariant space. Belovs'

overcomes this hurdle by expanding the walk space, and defining the walk operator in terms of the starting distribution. This ends up guaranteeing an eigenvalue-1 eigenvector which overlaps significantly with the chosen starting distribution σ if and only if there are marked elements.

Specifically, let $G = (V, E)$ be an undirected, fully-connected and bipartite graph with bipartitions A and B and weights w. Belovs adds an additional register to the Hilbert space for each vertex occurring with non-zero amplitude in the starting state, so that the computational basis becomes $\{|u\rangle \mid u \in S\} \cup \{|e\rangle \mid e \in E\}$, where S is the support of σ. Let $\mathcal{H}_u$ denote the *local space* of u, the space spanned by all $|(u, v)\rangle$ for $(u, v) \in E$, and also $|u\rangle$ in case $u \in S$ [Bel13].

Belovs defines his walk operator by making connections to electrical network theory. Let $W = \sum_{e \in E} w(e)$ be the sum of edge weights. Let $M \subseteq V$ be the set of marked elements. A *flow* from σ to M is a function $f : E \to \mathbb{R}$ such that for all edges $f(uv) = -f(vu)$ and for all non-marked vertices u, the flow has $\sigma_u = \sum_{vu \in E} f(uv)$. In short: σ_u flow is injected into vertex u, flows through the graph, and is removed at a marked vertex. The *energy* of a flow is given by

$$\sum_{e \in E} \frac{p(e)^2}{w(e)}.$$

The *effective resistance* $R_{\sigma,M}$ is the minimal energy of a flow from σ to M. The flow attaining this minimal energy is called the *electrical flow*. Note that the flow at a wire can exceed the weight of the edge (unlike in flow networks), akin to the electrical potential difference ('voltage') exceeding the resistance of a wire. Indeed, the above statement of energy is really just a paraphrasing of Ohm's law, where the flow is the potential difference across an edge.

Belovs defines the walk operator as $W = R_B R_A = \bigoplus_{u \in A} D_u \bigoplus_{u \in B} D_u$ where D_u is a reflection in the local space of vertex u, i.e. the space spanned by $|uv\rangle \in E$ and $|u\rangle$ if $u \in S$. Specifically:

- If u is marked then D_u is the identity.
- If u is not marked, then D_u reflects around the orthogonal complement of ψ_u in $\mathcal{U}$ where

$$\psi_u = \sqrt{\frac{\sigma(u)}{C_1 R}}\,|u\rangle + \sum_{uv \in E} \sqrt{w(uv)}\,|uv\rangle\,,$$

  $C_1 > 0$ is a constant, and R is a given upper-bound on the effective resistance from σ to M. Note that this also applies to $u \notin S$, asn then the first term in the above simply disappears.

Effectively, what this does is add a new starting vertex s' and additional edges $s'u$ for every vertex u with weight $w(s'u) = \sqrt{\frac{\sigma(u)}{C_1 R}}$. Phrased like this, the definition of Belovs quantum walk is equivalent to Szegedy's. It turns out that the invariant vector $|\varphi\rangle$ is the electrical flow state: an encoding of the electrical flow. By increasing C_1, we can alter this state to have more weight on the starting vertices. This then increases the overlap between the starting distribution and $|\varphi\rangle$, and therefore the probability to determine the phase of $|\varphi\rangle$ when doing phase estimation on the starting distribution.

Algorithm 4 (Belovs' quantum walk search)**.** Input: An undirected, strongly connected and non-bipartite graph $G = (V, E)$ with weights w and marked subset $M \subseteq V$, upper-bound R on the effective resistance $R_{\sigma,M}$, constant C, and starting distribution σ. Output: True if $M \neq \emptyset$, false if $M = \emptyset$.

1. Start in state $|\sigma\rangle = \sum_{u \in V} \sqrt{\sigma_u}\,|u\rangle$.
2. Apply phase estimation on unitary $R_B R_A$ and vector $|\sigma\rangle$ with precision $1/(C\sqrt{RW})$.
3. Output true if the resulting eigenvalue is 1, and false otherwise.

Theorem 3.1.1 ([Bel13], Theorem 4)**.** Given an undirected graph $G = (V, E)$ with weights w and marked subset $M \subseteq V$, starting distribution σ, upper-bound W on the weight of the graph and upper-bound R on the effective resistance $R_{\sigma,M}$.

For any $\delta \in (0, 1)$, Algorithm 4 determines whether M is empty with success probability $1 - \delta$ and uses $R_B R_A$ $O(RW)$ times.

Note that the classical hitting time can be written as $H_{\sigma,M} = 2WR_{\sigma,M}$ [Bel13, Theorem 1], so that this is exactly a quadratic improvement over classical random walks, and therefore a true generalisation of Szegedy's detection algorithm to arbitrary starting distributions.

3.2 Belovs' proof

We now give a breakdown of Belovs' proof of the above Theorem 3.1.1, filling in several details left out by Belovs. We will use following standard result.

Lemma 3.2.1 (Effective spectral gap lemma, [LMR$^+$11])**.** Let Π_A and Π_B be two orthogonal projectors in the same vector space, and $R_A = 2\Pi_A - I$ and $R_B = 2\Pi_B - I$ be reflections around their image. Assume P_Θ with $\Theta \geq 0$ is the orthogonal projectors on the span of the eigenvectors of $R_B R_A$ with eigenvalues $e^{i\theta}$ such that $|\theta| \leq \Theta$. Then, for any vector w such that $\Pi_A w = 0$, we have

$$||P_\Theta \Pi_B w|| \leq \frac{\Theta}{2}||w||.$$

Proof of Theorem 3.1.1. We start with the correctness. Let $|\sigma\rangle = \sum_{v \in V} \sqrt{\sigma_u}\,|u\rangle$ be the starting distribution of the algorithm and suppose that the graph containing a marked vertex. We wish to find an eigenvalue-1 eigenvector $|\varphi\rangle$ of U which has a large overlap with $|\sigma\rangle$. We propose

$$|\varphi\rangle = \sqrt{C_1 R} \sum_{u \in S} \sqrt{\sigma_u}\,|u\rangle - \sum_{e \in E} \frac{p_e}{\sqrt{w(e)}}\,|e\rangle\,.$$

Since U reflects about the orthogonal complement of each $|\psi_u\rangle$ in $\mathcal{H}_u$, showing that $|\varphi\rangle$ is

invariant under U means showing $|\varphi\rangle$ is orthogonal to each $|\psi_u\rangle$. Let u be arbitrary and consider

$$\begin{aligned}\langle\varphi|\psi_u\rangle &= \left\langle \sqrt{C_1R}\sum_{u\in S}\sqrt{\sigma_u}\,|u\rangle - \sum_{e\in E}\frac{p_e}{\sqrt{w(e)}}\,|e\rangle \middle| \sqrt{\frac{\sigma(u)}{C_1R}}\,|u\rangle + \sum_{uv\in E}\sqrt{w(uv)}\,|uv\rangle \right\rangle \\ &= \sqrt{\left(\sqrt{C_1R}\sqrt{\sigma_u}\sqrt{\frac{\sigma(u)}{C_1R}}\right)^2 - \sum_{uv\in E}\left(\sqrt{w(uv)}\frac{p_{uv}}{\sqrt{w(uv)}}\right)^2} \\ &= \sqrt{\sigma_u^2 - \sum_{uv\in E}p_{uv}^2} = 0.\end{aligned}$$

Note that the last step follows because flows, by definition, satisfy $|\sigma_u\rangle = \sum_{uv\in E}p_{uv}$ for non-marked vertices u. We have thus shown that our vector $|\varphi\rangle$ is indeed invariant under U. What remains to be shown is that it significantly overlaps with $|\sigma\rangle$:

$$\langle\varphi|\sigma\rangle = \sqrt{C_1R}\sum_{u\in S}\sqrt{\sigma_u}\sqrt{\sigma_u} - 0 = \sqrt{C_1R}\sum_{u\in S}\sigma_u = \sqrt{C_1R}.$$

Note that φ is not in general normalised. Correcting for this, we can express the φ-component of $|\sigma\rangle$ as

$$\left\langle \frac{\varphi}{||\varphi||}\middle|\sigma\right\rangle = \frac{\sqrt{C_1R}}{||\varphi||} = \frac{\sqrt{C_1R}}{\sqrt{C_1R + \sum_{e\in E}\frac{p_e^2}{w(e)}}} = \frac{\sqrt{C_1R}}{\sqrt{C_1R + R_{\sigma,M}}} = \sqrt{\frac{C_1R}{C_1R + R_{\sigma,M}}}.$$

Since R is an upper-bound for $R_{\sigma,M}$, we have

$$\left\langle \frac{\varphi}{||\varphi||}\middle|\sigma\right\rangle = \sqrt{\frac{C_1R}{C_1R + R_{\sigma,M}}} \geq \sqrt{\frac{C_1R}{C_1R + R}} = \sqrt{\frac{C_1}{1+C_1}}.$$

So, we can lower-bound the overlap of $|\sigma\rangle$ and $|\varphi\rangle$ by tuning the constant C_1. Recall from section 2.1.5 that phase estimation returns a superposition of phases corresponding to each eigenvector in the eigendecomposition, with the amplitude of the phase being the overlap of the input vector with the corresponding eigenvalue. The above lower-bound on the overlap is thus precisely the amplitude of the phase 0 in the result of phase estimation, so that we observe phase 0 with probability at least

$$\left|\sqrt{\frac{C_1}{1+C_1}}\right|^2 = \frac{C_1}{1+C_1}.$$

We now move to the negative case. Suppose that no marked elements exists. As explained, we would like to show that $|\sigma\rangle$ has little overlap with any eigenvalue-1 eigenvector of U. We will make use of the Lemma 3.2.1 to show this. To use this Lemma, we need to find a state $|w\rangle$ such that $\Pi_A|w\rangle = 0$ and $\Pi_B|w\rangle = |\sigma\rangle$, where Π_A and Π_B are projectors onto the invariant space of R_A and R_B. We propose

$$w = \sqrt{C_1R}\left(\sum_{u\in S}\sqrt{\frac{\sigma(u)}{C_1R}}\,|u\rangle + \sum_{e\in E}\sqrt{w(e)}\,|e\rangle\right).$$

To show that $\Pi_A w = 0$, we need to show that w has no overlap with the invariant space of A, i.e. no overlap with the orthogonal complements of vectors ψ_u. In other words, we need to show that w can be written as a linear combination of vectors ψ_u. To show that $\Pi_B w = |\sigma\rangle$ we need to show that $|\sigma\rangle$ can be written as a linear combination of the orthogonal complements of vectors ψ_u, while the rest of w can be written as a linear combination of vectors ψ_u. Both are quite clearly the case.

Applying Lemma 3.2.1, we obtain

$$||P_\Theta \Pi_B w|| = ||P_\Theta |\sigma\rangle|| \leq \frac{\Theta}{2}||w||;$$

we have upper-bounded the overlap of $|\sigma\rangle$ with eigenvectors of U with phases $\leq \Theta$ - in particular phase 0 - which is what we wanted. Since we care only about phase 0, we decrease Θ to exclude non-relevant phases. To do so elegantly, let us define

$$\Theta = \frac{1}{C_2\sqrt{1 + C_1 RW}}.$$

where $C_2 > 0$ is a constant, so that

$$||P_\Theta |\sigma\rangle|| \leq \frac{\Theta}{2}||w|| = \frac{1}{2C_2\sqrt{1 + C_1 RW}}\sqrt{1 + C_1 RW} = \frac{1}{2C_2},$$

where we make use of

$$||w|| = \sqrt{\left(\sqrt{C_1 R}\left(\sum_{u\in S}\sqrt{\frac{\sigma_u}{C_1 R}} + \sum_{e\in E}\sqrt{w(e)}\right)\right)^2} = \sqrt{\sum_{u\in S}\sigma_u + C_1 R\sum_{e\in E} w(e)} = \sqrt{1 + C_1 RW}.$$

We can now set an upper-bound on the overlap of $|\sigma\rangle$ and any eigenvectors of U with phase $\leq \Theta$ by tuning the constant C_2, so that we don't have to worry about complex expressions involving W and R. This upper-bound on the overlap is thus precisely the amplitude of the phases $\leq \Theta$ in the result of phase estimation, and thus induces the probability of observing such a phase. Taking the inverse, we obtain a lower-bound on the probability of observing a phase $> \Theta$, and in particular on observing a phase unequal to 0, which is again what we wanted:

$$1 - \left|\frac{1}{2C_2}\right|^2 = 1 - \frac{1}{4C_2^2}.$$

This concludes the correctness proof. Let us turn towards the complexity. Note that the negative case requires running phase estimation to distinguish phases as small as Θ: we guarantee a small overlap between $|\sigma\rangle$ and phases smaller than Θ, but we don't know anything about phases larger than Θ, and so a phase $\Theta + \epsilon$ might already have significant overlap with $|\sigma\rangle$. There is no such requirement for the positive case, as the phase 0 can be expressed exactly even with a single bit. To distinguish between phases as small as Θ, then, requires a number of bits b such that $2^{-b} \leq \Theta$, in other words $O(1/\Theta) = O(\sqrt{RW})$ bits. Queries are only made in the phase estimation subroutine, so the complexity now follows immediately from the complexity of phase estimation (Theorem 2.1.5), which is $O(2^{\log(\sqrt{RW})}) = O(\sqrt{RW})$.

As noted above, by increasing the constants C_1 and C_2, the success probability can become arbitrarily large. Since this does not alter the asymptotic complexity, we are done. □

3.3 Exact complexity and amplifying the success probability

As already noted, queries are made only in the phase estimation subroutine. When phase estimation uses t bits of precision, it makes exactly $2^0 + 2^1 + ... + 2^{t-2} + 2^{t-1} = 2^t - 1$ queries to the given unitary. This then already gives us an exact number of queries made. All that is left is filling in the precision t. As argued above, the precision should be the smallest number of bits t such that $2^{-t} \leq \Theta$, so that $t \geq \log(1/\Theta)$, meaning the smallest possible precision is:

$$t = \lceil \log 1/\Theta \rceil = \left\lceil \log C_2 \sqrt{1 + C_1 RW} \right\rceil.$$

Recall that W and R are fixed by the problem instance, while $C_1 > 0$ and $C_2 > 0$ are chosen by us. For instance, if we choose $C_1 = C_2 = 1$, we get an algorithm that succeeds with probability $1/2$ in case marked elements exist, $3/4$ otherwise, and uses $R_B R_A$ exactly

$$2^t - 1 = 2^{\lceil 1/\Theta \rceil} - 1 = 2^{\lceil C_2 \sqrt{1 + C_1 RW} \rceil} = 2^{\lceil \sqrt{1 + RW} \rceil},$$

times.

We would like to have the algorithm succeed with some precision $1 - \delta$. We already noted above that we can achieve this simply by increasing C_1 and C_2 accordingly. This is also Belovs' proposed method to amplify the success probability. However, it is not clear what the complexity of this is: it increases the precision of phase estimation, but how much?

Alternatively, as explained in Section 4.5.1, we can simply repeat the algorithm $O(\log(1/\delta))$ times. We then accept if the number of accepted runs exceeds the mean of the expected number of accepted outcomes in the positive and negative case. That is, if X is a random variable describing n runs of the algorithm, it has $\mathbb{E}(X) \geq 1/2n$ in the positive case and $\mathbb{E}(X) \leq 1/4n$ in the negative case, so that we want to accept whenever $X > 3/8n$. We then want to limit the probability to diverge from $\mathbb{E}(X)$ by more than $1/n8$, because if we stay within $1/8n$ of the real outcome, we output correctly. Chernoff's bound gives us a $n \in O(\log(1/\delta))$ which suffices for this. This is the method Montanaro proposes to amplify the success probability of Belovs' algorithm [Mon16].

In the following two subsections, we consider the slowdown caused by either method. In the section after this, we compare the two, and determine which (combination) of the two is optimal.

3.3.1 Increasing the constants

To achieve success probability $1 - \delta$ we need to set the constants C_1 and C_2 to:

$$\begin{aligned}
&\frac{C_1}{1 + C_1} \geq 1 - \delta \iff \\
&C_1 \geq (1 - \delta)(1 + C_1) \iff \\
&C_1 \geq 1 + C_1 - \delta - \delta C_1 \iff \\
&\delta C_1 \geq 1 - \delta \iff \\
&C_1 \geq \frac{1}{\delta} - 1.
\end{aligned}$$

$$\begin{aligned}
&1 - \frac{1}{4C_2^2} \geq 1 - \delta \iff \\
&\frac{1}{4C_2^2} \leq \delta \iff \\
&1 \leq 4C_2^2 \delta \iff \\
&\frac{1}{\delta} \leq 4C_2^2 \iff \\
&C_2 \geq \sqrt{\frac{1}{4\delta}} = \frac{1}{2}\sqrt{\frac{1}{\delta}}.
\end{aligned}$$

We then have

$$\begin{aligned}
C_2\sqrt{1+C_1R_{\sigma,M}W} &= \frac{1}{2}\sqrt{\frac{1}{\delta}}\sqrt{1+\left(\frac{1}{\delta}-1\right)R_{\sigma,M}W} \\
&= \frac{1}{2}\sqrt{\frac{1}{\delta}}\sqrt{\left(\frac{1}{\delta}-1\right)\left(R_{\sigma,M}W+\frac{1}{1/\delta-1}\right)} \\
&= \frac{1}{2}\sqrt{\frac{1}{\delta}}\sqrt{\frac{1}{\delta}(1-\delta)\left(R_{\sigma,M}W+\frac{1}{1/\delta-1}\right)} \\
&= \frac{1}{2\delta}\sqrt{(1-\delta)\left(R_{\sigma,M}W+\frac{1}{1/\delta-1}\right)} \\
&\leq \frac{1}{2\delta}\sqrt{R_{\sigma,M}W+2\delta},
\end{aligned}$$

where the final inequality holds only for $\delta \in (0, \frac{1}{2})$, as for these values we have $\frac{1}{1/\delta-1} \leq 2\delta$. The precision with which we need to do phase estimation becomes

$$\Theta = \frac{1}{C_2\sqrt{1+C_1RW}} = \frac{1}{\frac{1}{2\delta}\sqrt{(1-\delta)\left(R_{\sigma,M}W+\frac{1}{1/\delta-1}\right)}} = \frac{2\delta}{\sqrt{(1-\delta)\left(R_{\sigma,M}W+\frac{1}{1/\delta-1}\right)}}$$

so that the number of queries is

$$2^{\left\lceil \log\left(\frac{1}{2\delta}\sqrt{(1-\delta)\left(R_{\sigma,M}W+\frac{1}{1/\delta-1}\right)}\right)\right\rceil}-1 \leq \frac{1}{2\delta}\sqrt{(R_{\sigma,M}W+2\delta)} \in O(\frac{1}{\delta}\sqrt{RW}).$$

We therefore suffer a multiplicative slowdown of at most $1/(2\delta) \in O(1/\delta)$.

Theorem 3.3.1. Given an undirected graph $G = (V, E)$ with weights w and marked subset $M \subseteq V$, starting distribution σ, upper-bound W on the weight of the graph and upper-bound R on the effective resistance $R_{\sigma,M}$.

Running Algorithm 4 with $C_1 = \frac{1}{\delta} - 1$, $C_2 = \frac{1}{2}\sqrt{\frac{1}{\delta}}$ and therefore precision

$$\Theta = \frac{2\delta}{\sqrt{(1-\delta)\left(R_{\sigma,M}W+\frac{1}{1/\delta-1}\right)}}$$

determines whether M is empty with success probability at least $1-\delta$ and uses R_BR_A exactly

$$2^{\left\lceil \log\left(\frac{1}{2\delta}\sqrt{(1-\delta)\left(R_{\sigma,M}W+\frac{1}{1/\delta-1}\right)}\right)\right\rceil}-1 \in O(\frac{1}{\delta}R_{\sigma,M}W)$$

times.

3.3.2 Repeating the algorithm

Suppose, as Montanaro does, that $C_1 = C_2 = 1$, implying success probability $1/2$ and $\mathbb{E}(X) \geq 1/2n$ in the positive case, and $3/4$ and $\mathbb{E}(X) \leq 1/4n$ in the negative case. We then wish to determine the smallest n that limits the probability to diverge from $\mathbb{E}(X)$ by more than $1/8n$ to at most δ. Recall the statement of Chernoff's bound from Section A.1:

$$P(|X - \mu| > \alpha\mu) \leq 2e^{-\frac{\alpha^2\mu}{2+\alpha}}.$$

Our answer lies on the right-side of this inequality. We of course have to treat each case separately, as μ differs. Let us start with the positive case, recalling that $\mu = 1/2n$ and $p = 1/2$. We wish to have $\alpha\mu = 1/8n$ and thus $\alpha = (1/8n)/(1/2n) = 1/4$. Filling in the expression from Chernoff's bound:

$$\begin{aligned}
2e^{-\frac{\alpha^2\mu}{2+\alpha}} &\leq \delta \iff \\
2e^{-\frac{(1/4)^2 n 1/2}{2+1/4}} &\leq \delta \iff \\
e^{-n/72} &\leq \delta/2 \iff \\
-n/72 &\leq \ln(\delta/2) \iff \\
n &\geq -72\ln(\delta/2) \iff \\
n &\geq 72\ln(2/\delta).
\end{aligned}$$

Now the negative case. Recall that $\mu = 1/4n$ and $p = 1/4$. We wish to have $\alpha\mu = 1/8n$ and thus $\alpha = (1/8n)/(1/4n) = 1/2$. We write

$$\begin{aligned}
2e^{-\frac{\alpha^2\mu}{2+\alpha}} &\leq \delta \iff \\
2e^{-\frac{(1/2)^2 n 1/4}{2+1/2}} &\leq \delta \iff \\
e^{-n/40} &\leq \delta/2 \iff \\
n &\geq 40\ln(2/\delta).
\end{aligned}$$

The positive case is thus the worst-case, which is to be expected, as its success probability is much lower than that of the negative case. Since we don't know in advance which case holds (as we lack a distribution on the input), we have to assume the worst-case. We then have the following algorithm.

Theorem 3.3.2. Given an undirected graph $G = (V, E)$ with weights w and marked subset $M \subseteq V$, starting distribution σ, upper-bound W on the weight of the graph and upper-bound R on the effective resistance $R_{\sigma,M}$.

Running Algorithm 4 with $C_1 = C_2$ and therefore precision $\Theta = \dfrac{1}{\sqrt{1 + RW}}$ for $\lceil 72\ln(2/\delta)\rceil$ times and outputting positively if and only if more than $3/8$ of runs do so, determines whether M is empty with success probability at least $1 - \delta$ and uses $R_B R_A$ exactly

$$\lceil 72\ln(2/\delta)\rceil \, (2^{\lceil \log\sqrt{1+RW}\rceil} - 1)$$

times.

3.4 Optimising the algorithm

Let us now compare these two ways to amplify the success probability. Asymptotically, increasing the constants implied a slowdown $O(1/\delta)$, while repeating the algorithm implied an exponentially better $O(\log(1/\delta))$. However, the actual slowdown was roughly $1/(2\delta)$ for the former, and $\lceil 72 \ln(2/\delta) \rceil$ for the latter. This large difference in constant overhead means it is not clear that repeating the algorithm is always better. Indeed, these two expressions are equal for $\delta \approx 0.0020124$, so that repeating the algorithm is the most efficient method only once δ falls below 0.0020124.

A simple optimisation to Belovs' algorithm is therefore the following: given the desired error probability δ, if it is below 0.0020124, repeat the algorithm to amplify the success probability, otherwise, increase the constants C_1, C_2.

However, note that the slowdown suffered by the 'repetition method' was derived by assuming $C_1 = C_2 = 1$. Therefore, it is not clear that the above optimisation is really optimal: perhaps for much smaller δ it would be cheaper to reduce the constants and add some more repetitions, while perhaps for much larger δ the reverse would be better.

Let us therefore ask: given a maximum error probability δ, to what value should we increase the constants C_1 and C_2, and how many times n should we repeat the phase estimation algorithm to minimise the number of queries?

Domain of C_1, C_2 Before we tackle this question, let us make a note about the domain from which we draw C_1 and C_2, and on their relation. Although Belovs' defined these constants as $C_1 > 0$ and $C_2 > 0$, not all such values make sense. Recall that these constants imply a success probability of $p_+ = \frac{C_1}{1+C_1}$ in the positive case, and of $p_- = 1 - \frac{1}{4C_2^2}$ in the negative case. Repeating the algorithm to boost the success probability relies on there being a gap between p_+ and $1 - p_-$, so that the sum of repetitions of the algorithm tends to one side of this gap. In our example above, we set $C_1 = C_2 = 1$ and had $p_+ = 1/2$ and $p_- = 3/4$ so that there was a gap between $1/2$ and $1/4$, and we accepted after n runs if and only if the sum of outcomes was greater than $3/8n$.

In principle, this allows very disproportionate success probabilities between the two cases. This is not desirable, as the number of repetitions required depends on these success probabilities. Lacking a distribution over the income, we will have to assume the worst-case, so that disproportionate success probabilities will increase the complexity needlessly. We saw this in the last section, where the number of repetitions between the two cases different by a factor of almost 2.

Let us then relate the constants C_1 and C_2 so that the success probabilities in the two cases

are equal:

$$\begin{aligned}
p_+ = p_- &\iff \\
\frac{C_1}{1+C_1} = 1 - \frac{1}{4C_2^2} &\iff \\
\frac{C_1 4C_2^2}{1+C_1} = 4C_2^2 - 1 &\iff \\
C_1 4C_2^2 = (4C_2^2 - 1)(1+C_1) &\iff \\
C_1 4C_2^2 = 4C_2^2 + C14C_2^2 - 1 - C_1 &\iff \\
4C_2^2 = 1 + C_1 &\iff \\
C_2 = \frac{1}{2}\sqrt{1+C_1}.
\end{aligned}$$

Since from now on we will choose only C_1 (and not C_2), let us use C to refer to C_1. Given $p_+ = p_-$, it is obvious that we should have $p_+ \in (1/2, 1]$ to guarantee a 'gap' between p_+ and $1 - p_-$. What values of C does this correspond to?

$$p_+ > 1/2 \iff \frac{C}{1+C} > 1/2 \iff 2C > 1 + C \iff C > 1.$$

Note that this corresponds to $C_2 > \frac{1}{\sqrt{2}}$. The following function gives the number of queries that the algorithm configured with C and n makes.

$$f(C, n, R, W) = n(2^{\lceil \log(C_2\sqrt{1+CRW}) \rceil} - 1) = n(2^{\lceil \log(\frac{1}{2}\sqrt{1+C}\sqrt{1+CRW}) \rceil} - 1) = n(2^{\left\lceil \log \sqrt{(1+C)(1+CRW)} \right\rceil - 1} - 1).$$

Optimising C and n Let us now consider our question. Since we have a one-sided error, we can use the exact expression for the error probability of the algorithm after n repetitions from Proposition A.1.1, where we can fill in $p = (1 + C)/C$.

$$\delta \leq (1-p)^n \sum_{i=0}^{n/2} \binom{n}{i} \left(\frac{p}{1-p}\right)^i = (\frac{1}{1+C})^n \sum_{i=0}^{n/2} \binom{n}{i} \left(\frac{\frac{1+C}{C}}{\frac{1}{1+C}}\right)^i = (\frac{1}{1+C})^n \sum_{i=0}^{n/2} \binom{n}{i} \left(\frac{(1+C)^2}{C}\right)^i.$$

Thus, filling in some choice for C and n, the above gives you the exact error probability. Alternatively, given a desired maximum δ and either n or C, the above given a much simplified expression, which can be solved for the missing variable n or C. In particular, choosing $n = 1$ gives us the results of Section 3.3.1, and choosing C just above 1 gives us something akin to the results of Section 3.3.2 (but note quite, as there, p_+ and p_- differed, but we now fix them to be equal).

An optimal setting of n and C would minimise the number of queries $f(C, n, R, W)$. Since the number of queries is a function of both C and n and on the input-dependent R and W, it is not clear that there is a global optimum (i.e. across all input instances, all R and W). It might be that C or n induce a greater slowdown for larger instances than for smaller instances, or vice versa.

However, it is easily seen that this is not the case: n doesn't discriminate and simply slows the whole expression down by a factor n. Similarly, $\sqrt{(1+C)(1+CRW)}$ shows us that C scales the term $\sqrt{RW}$ by $\sqrt{C + C^2}$, and adds a constant factor $\sqrt{C}$ to the complexity. All of these slowdowns are then independent of R and W.

It follows that for any δ, there is a globally optimal setting for C and n. To find it, consider the following method. For each $n = 1, 2, 3, ...$, fill in the above expression for δ in terms of C and n. This leaves C as the only unknown, simplifying the expression, allowing us to easily solve it for C. We can then compute the number of queries $f(C, n, R, W)$. If this is larger than the number of queries for the previous n, stop and output the n corresponding to the lowest number of queries seen thus far.

To see why this works, we claim that once we find f increases from n to $n+1$, f will continue to rise monotonically. This is because as n rises, C drops, as we don't need a large C anymore to amplify the success probability. However, we must have $C > 1$, so at some point C will not be able to drop further, and incrementing n makes the query count strictly and monotonically increase. Thus, once f starts increasing, we know the optimal n is among the n we have already checked. Since the success probability increases exponentially with n, the number of required iterations is small, making this approach feasible (even if finding C might be costly).

Unfortunately, there is a slight caveat here. The actual number of queries doesn't start rising monotonically from any point on due to the ceiling function. Instead, it continuously rises and falls, in a staircase-like pattern, which eventually starts tending upwards. If we could determine the point from which this upwards momentum starts, we could find the minimum by searching all smaller n. We can do this by upper-bounding the expression by removing the ceiling function.

$$f(C, n, R, W) = n(2^{\left\lceil \log \sqrt{(1+C)(1+CRW)} \right\rceil - 1} - 1) \leq n(\sqrt{(1+C)(1+CRW)} - 1).$$

This yields a well-behaved function which does start rising monotonically from the minimum on. Using this we can locate the n before which the optimum lies, allowing us to check each smaller n and find the optimum. In Table 3.1 we present optimal values for various δ. Of course, these

δ	n	C
1/10	1	9
1/100	5	9.8176
1/1000	9	10.394
1/10000	13	10.756
1/100000	17	11.007
$\cdot 10^{-6}$	21	11.192
$\cdot 10^{-7}$	25	11.335
$\cdot 10^{-8}$	29	11.450

Table 3.1: Optimal values for n and C for various δ. These hold for all instances, i.e. for any W and R.

values hold relative to the upper-bound on the number of queries $f(C, n, R, W)$. As explained, the ceiling function in f prevents it from ever rising monotonically, meaning there is no global optimum for f itself. Thus, computing the optimal C and n relative to f for various sizes of R and W will result in slightly differing C and n. We see this in Figure 3.1.

Both C and n seem to stay neatly within a range of roughly $[7, 13.5]$ and their behaviour within the range seems periodic. The global optima $C = 10.394$ and $n = 9$ seem to play a special role, with the orange lines being at these optima for an extended period. Studying the behaviour of the real optima as plotted here, (i.e. relative to the exact f) might allow one to set C and n to be even closer to the real optimum, on average. We leave this optimisation for future work, settling for our optima relative to the upper-bounded f.

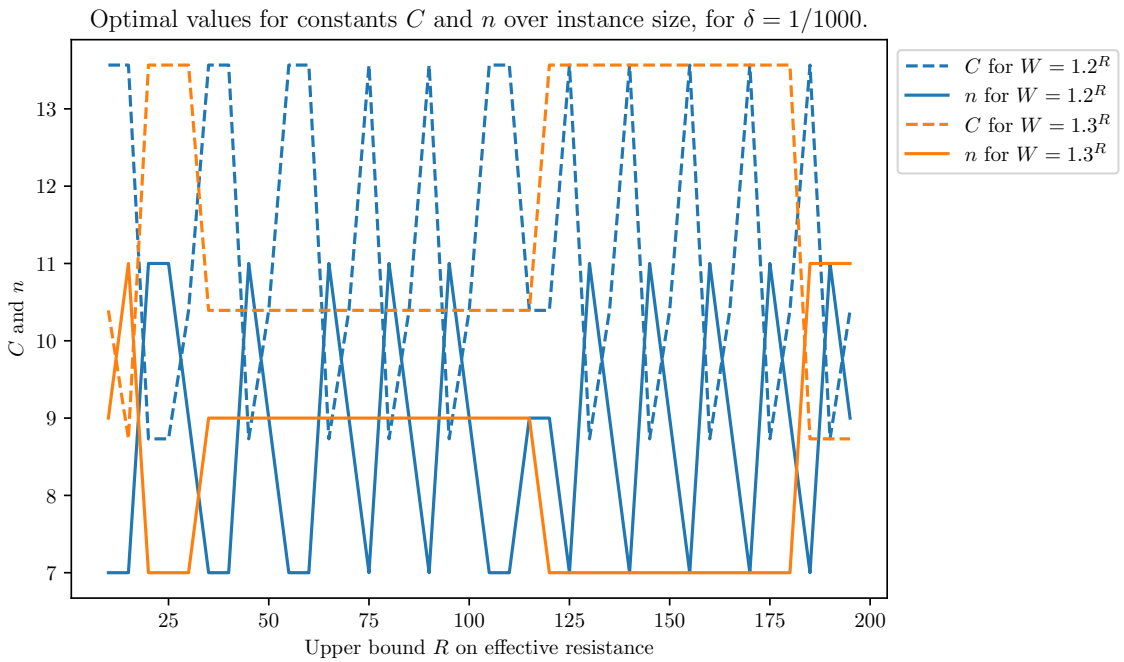

Figure 3.1: Optimal values for n and C for $\delta = 1/1000$ over increasing R and $W = 2^{1.2}$ or $W = 2^{1.3}$.

Theorem 3.4.1. Given an undirected graph $G = (V, E)$ with weights w and marked subset $M \subseteq V$, starting distribution σ, upper-bound W on the weight of the graph and upper-bound R on the effective resistance $R_{\sigma,M}$.

For any success probability $1 - \delta$, there is an optimal choice of constants C, n so that running Algorithm 4 with precision $\Theta = \dfrac{1}{\sqrt{(1+C)(1+CRW)}}$ for n times and outputting positively if and only if more than $3/8$ of runs do so, determines whether M is empty with success probability at least $1 - \delta$ and uses $R_B R_A$ a minimal number of times

$$f(C, n, R, W) = n(2^{\left\lceil \log \sqrt{(1+C)(1+CRW)} \right\rceil - 1} - 1) \in O(\sqrt{RW})$$

times. Table 3.1 contains optimal C, n for various δ, and the discussion preceding it gives a general method to determine C, n for any δ.

We finally note that the f is classically computable. All the parameters of f have to be provided to the quantum algorithm. We can assume, then, that if we have an input instance of interest, we should already be able to determine these parameters. Computing f using these parameters is trivial.

In our experiments, we will, among other things, be interested in the optimal performance of Belovs' algorithm, i.e. when R and W are tight upper-bounds. To that end, we note that

we can classically compute the effective resistance $R_{\sigma,M}$ and the sum of weights of any input graph in polynomial time. In particular, for our application to backtracking, this will require constructing the entire backtracking tree.

Chapter 4

Search

In this chapter, we study two quantum walk search algorithms. The first is a simple binary search algorithm making use of Belovs' detection algorithm from Chapter 3, the second is Piddock's search algorithm, which was the first quantum walks earch algorithm to work on general graphs — a title shared with the independent and concurrent work [AGJ21]. We derive an exact expression for the query complexity of both algorithm.

In section

4.1 we consider how to construct a search algorithm from a detection algorithm using binary search.

4.2 we consider a far more efficient search algorithm due to Montanaro, that only works when there is a unique marked vertex.

4.3 inspired by this, we give the motivation for Piddock's general search algorithm and outline the two main components, which we tackle in the next two sections;

4.4 we consider the first major component of Piddock's algorithm: an algorithm to estimate the effective resistance of a graph. We derive an exact expression for its query complexity and show how the parameters of this expression can be computed classically.

4.5 we consider the second major component of Piddock's algorithm, derive an exact expression for its query complexity and show how the parameters of this expression can be computed classically.

4.6 we combine the previous two section into Piddock's final algorithm, derive an exact expression for its expected query complexity and show how the parameters of this expression can be computed classically.

Across the last three section, we derive some new upper-bounds on the workings of Piddock's algorithm and propose some small optimisations.

4.1 Detection and binary search

There is an obvious way to turn a detection algorithm into a search algorithm at only a logarithmic multiplicative slowdown: binary search on the search tree. First apply the detection algorithm to the root of the tree, and if it outputs no, you are done. If it outputs that marked

elements exist, check whether the root is marked, and if it is, you are done. If it isn't, the marked element must be in one of the subtrees at the children of the root. Apply the detection algorithm at each such subtree, and when one returns positively, check whether that child vertex is marked. If so, you are done. If none of these children were marked, the marked element must be in one of the subtrees at children who returned positively when applying the detection algorithm. Pick one of these children, and continue down their subtree. Repeat until a marked element is found.

If each vertex has a constant number of children, and the depth of the tree is $m \in O(\log T)$, where T is an upper-bound on the tree size, the number of repetitions of the detection algorithm is $O(1) \cdot m \in O(\log T)$. Of course, this finds a marked vertex conditioned on the detection algorithm being correct all m times. By Lemma A.1.2, this requires reducing the error probability of each run of the detection algorithm to $O(1/m)$, which by Lemma A.1.1 requires doing a majority vote of at least $O(\log m)$ repetitions, for a total of $O(n \log m)$ repetitions, and a final complexity of $O(\sqrt{RW} m \log m)$. For backtracking, $T = W$ and $R \leq m$, so that we can write the complexity as $O(\sqrt{T} m^{3/2} \log m)$. This method and complexity was noted already by Montanaro [Mon18, p.11-12].

Note that we need to count each repetition as contributing to the final complexity fully, i.e. contributing a factor $\sqrt{RW}$. It may seem that later in the algorithm, as we are deeper into the tree, these bounds can be decreased, as we are considering a sub-tree that can be exponentially smaller than the full tree. Unfortunately, in general, the tree could extremely skewed, so that the size and depth of the subtree drops by only 1 at each level.

Let us work out the exact query complexity of doing binary search. From Theorem 3.4.1, we know that $f(C, n, R, W)$ gives the number of queries made by one run of the optimised detection algorithm, where we assume some desired success probability $1 - \delta$, which implies optimal constants C and n by Table 3.1. How many repetitions of the detection algorithm do we need to do to guarantee binary search succeeds with probability $1 - \delta$ as well?

If the input instance is satisfiable, we need to repeat the detection algorithm to walk to a solution, say at a depth r. Of course, this requires not just r repetitions to walk to the solution, but also additional repetitions to make sure the probability of all runs being successful is still $1 - \delta$. To compute the required number of repetitions, recall from Proposition A.1.2 that guaranteeing with probability $1 - \delta$ that all r runs are correct, requires the base probability to be at least $p \geq \sqrt[r]{1 - \delta}$, and by Proposition A.1.1, this requires n_2 repetitions, where n_2 is the solution to

$$1 - \sqrt[r]{1 - \delta} \leq (1 - p)^{n_2} \sum_{i=0}^{n_2/2} \binom{n_2}{i} \left(\frac{p}{1 - p} \right)^i \in O(\log \frac{1}{\delta}).$$

Theorem 4.1.1. Given an undirected graph $G = (V, E)$ with weights w and marked subset $M \subseteq V$, starting distribution σ, upper-bound W on the weight of the graph, upper-bound R on the effective resistance $R_{\sigma,M}$ and upper-bound $r_{\max}$ on the depth of G.

For any success probability $1-\delta$, binary search using the optimised detection algorithm from Theorem 3.4.1 can determine whether M is empty, and if not, return an $m \in M$, with probability $1-\delta$ while using $R_B R_A$ exactly

$$f_{\text{binary search,unsat}}(C, n, R, W, r, n_2) = \lceil n_2 \rceil \cdot f(C, n, R, W) \in O\left(\sqrt{RW} \log\log W\right)$$

times in case M is empty, and

$$f_{\text{binary search,sat}}(C, n, R, W, r, n_2) = \lceil n_2 \rceil \cdot r \cdot f(C, n, R, W) \in O\left(\sqrt{RW} \log W \log\log W\right)$$

otherwise. Here, r is the depth of the marked element m returned, and n_2 is the solution of

$$1 - \sqrt[r_{\max}]{1-\delta} \leq (1-p)^{n_2} \sum_{i=0}^{n_2/2} \binom{n_2}{i} \left(\frac{p}{1-p}\right)^i \in O(\log \frac{1}{\delta}).$$

We recall from the discussion of Theorem 3.4.1 that f is classically computable, and that we can even classically compute tight upper-bounds R and W given a graph (in our case, the classical backtracking tree). In addition, if we run a backtracking algorithm, like DPLL in our case, its heuristic leads the classical backtracking algorithm to a particular first solution at depth r. Since our binary search algorithm follows the same heuristic, it also walks towards this marked element at depth r. Running the classical backtracking algorithm then allows us to compute r classically. Given the upper-bound $r \leq r_{\max}$, we can easily compute n_2 by solving the given equation. All parameters we need to determine the query complexity are then classically computable, as desired.

4.2 The electrical flow state: efficient search on trees with a unique marked element

The binary search algorithm repeatedly computes the state $|\varphi\rangle$, but only uses it to check if the phase is equal to 0, as we found that $|\varphi\rangle$ is an eigenvector with phase 0 if and only a marked vertex exists. Given this intimate connections between $|\varphi\rangle$ and the existence of marked vertices, it seems fair to ask whether the content of this state might help us to find marked vertices more efficiently. Recall what this vector looked looks like

$$|\varphi\rangle = \frac{1}{\sqrt{2}} |r\rangle + \frac{1}{\sqrt{2n}} \sum_{x \neq r, x \rightsquigarrow x_0} (-1)^{l(x)} |x\rangle .$$

Half the amplitude is at the root $|r\rangle$, the other half is spread uniformly over the path from root to the marked vertex x_0. Collapsing this state then gives us, with probability $1/2$, some vertex on the path from root to x_0, and we expect to get a vertex in the middle of the path. But this then advances us much further down the path than the simple binary search, which advanced

us exactly once for each $O(1)$ repetitions of the detection algorithm. In essence, we are doing binary search over the tree depth, rather than the tree size, so that we expect to need $O(\log n)$ repetitions.

Now, phase estimation only approximates $|\varphi\rangle$. To create a useable algorithm we would need to determine the precision required of this approximation to ensure the correct success probability. Working out these details nets a final complexity of $O(\sqrt{RW}\log^3 m)$ [Mon18, p.12-14]. Crucially, however, this approach relies on there existing exactly one marked element. When more than one marked element exists, the eigenvalue-1 eigenspace of $R^B R^A$ is populated by multiple vectors of the form of $|\varphi\rangle$, each encoding the path to 'their' marked element. We expect these paths to overlap significantly near the top of the tree, as the tree is narrow at the top. The result is that we don't expect to traverse halfway along the depth of the tree anymore, but we instead expect to stay much closer to the top (akin to the classical binary search, which traverses one level at each step), reducing the efficiency of this approach as more marked elements are introduced. Note that this is quite counter-intuitive: we would expect that having more marked elements would make it easier to find one.

We should make a final note about the state $|\varphi\rangle$. The definition of $|\varphi\rangle$ given just above assumes that we configured the quantum walk unitary with $C = 1$. Recall from Section 3.3 that increasing the constant C also increased the amount of amplitude that $|\varphi\rangle$ assigns to the starting vertices in. This made $|\varphi\rangle$ overlap more with our starting state, increasing the probability to obtain $|\varphi\rangle$ (and its phase) through phase estimation, and therefore the success probability. We found the optimal value for C (given a desired success probability) which allowed us to optimally configure Belovs' algorithm. We didn't really care about the mutilation this caused to $|\varphi\rangle$ since we never used to actual content of this state: we only cared about the corresponding phase. Now we do care about the content of $|\varphi\rangle$, in particular, we don't want too much amplitude to be at the starting vertices, as we want to move away from these. Increasing C then increases the success probability of detection, but decreases the probability to walk towards a marked vertex. It follows that our optimisation won't work here, although a small increase in C may still be warranted: this is a question we don't consider here.

4.3 How to search efficiently on arbitrary graphs

The requirement of a unique marked element makes the above method unsuitable for most problems. Fortunately, the above method inspired more general solutions using $|\varphi\rangle$ to efficiently find a marked element in case multiple marked elements exist. This was done first by Jarret and Wann, whose algorithm works only for trees, and has a complexity of $O(\sqrt{R_{\max}W}\log^4(|M|R_{\sigma,M}))$, where $|M|$ is the number of marked elements, $R_{\max}$ the maximum effective resistance of all subtrees, and $R_{\sigma,M}$ the effective resistance between the root and the set of marked elements [JW18]. This was then generalised to arbitrary graphs by Piddock with a complexity of $O(\sqrt{R_{\sigma,M}W}\log^3(|M|))$ [Pid19]. The latter is then not only more general, but also more efficient (as far as the asymptotic complexity is concerned). Let us therefore study Piddock's algorithm, and attempt to give a tightened upper-bound (or even exact expression) for its query complexity.

Recall that in a tree we could use the electrical flow, which encodes the flow from root to marked elements, to walk towards to a marked element. In general graphs this method is not feasible, as marked vertices could go in differing directions, so that collapsing $|\varphi\rangle$ to iteratively walk towards a marked element has no guarantee to converge to a marked element. For example,

if we have a line with the starting vertex in the middle and a marked element at an equal distance on either side, we expect to stay at the starting vertex.

Piddock's algorithm therefore uses a different strategy: simply repeatedly measure $|\varphi\rangle$ in the hopes of directly observing a marked vertex. To make this approach feasible, he proposes two alterations to the graph to shift amplitude away from the starting states in $|\varphi\rangle$, and towards the marked vertices. This increases the probability to observe a marked vertex when measuring $|\varphi\rangle$ to $\Omega(1/\log|M|)$, requiring an expected $\log M$ repetitions to find a marked element. He shows that we can prepare the required electrical flow state with $O(\log^2|M|)$ repetitions of the detection algorithm, resulting in the final complexity of $O(\sqrt{R_{\sigma,M}W}\log^3|M|)$.

The two alterations are, first, adding an additional starting vertex s' and additional edges $s'u$ for every vertex u with weight $w(s'u)=\frac{\sqrt{\sigma_u}}{\eta}$, and second, adding an additional vertex k' and edge kk' with $w(kk')=1/x$ for each marked vertex $k\in M$. The new set of marked elements M' will be the set of these new vertices k'. The variables η and x are parameters of the algorithm. Recall that in Belovs' algorithm we would increase C to decrease the weight at starting edges (akin to increasing η here), which in turn increases the amplitude at the edges in $|\varphi\rangle$. We use the same principle here: by decreasing η we decrease the amplitude at starting edges in $|\varphi\rangle$, and by increasing x we increase the amplitude at marked edges in $|\varphi\rangle$. By finding appropriate settings for η and x, Piddock shows that we truly get a probability of $\Omega(1/\log|M|)$ to find a marked vertex when collapsing $|\varphi\rangle$, while still retaining a complexity of $O(\sqrt{R_{\sigma,M}})$ to run the detection algorithm (this last point is not obvious, as setting η and x changes the graph, so that its effective resistance may no longer be of the same order as the unaltered graph).

Piddock's algorithm then has three main components. We introduce these in the subsequent three section, ending each sections with a classically computable exact expressions for this component's query complexity. In the third section, this culminates in an exact expression for the complexity of the full search algorithm.

1. Determine an adequate η. We do this in Section 4.4, resulting in η equal to two times the effective resistance of the altered graph with η and $x=0$. We define Algorithm 5 to find this η.

2. Determine an adequate x. We do this in Section 4.5, resulting in a range $[\eta,b]$ from which to sample x, where b is chosen such that the effective resistance of the altered graph where $x=b$ is twice the effective resistance of the altered graph where $x=\eta$. We define Algorithm 6 to find this b.

3. Repeatedly sample x, construct R_BR_A for the choice of η and x, run phase estimation to construct $|\varphi\rangle$, and measure this state. This is expected to give a marked vertex within $O(\log|M|)$ repetitions. We give the final algorithm performing this step in Section 4.6, resulting in Algorithm 7.

Let us use $R_{s',M'}$ to refer to the effective resistance of the graph where the two alterations have been made, and likewise, $R_{\sigma,M'}$ and $R_{s',M}$ for the effective of the graph where only one of the alterations have been made.

4.4 Algorithm to estimate the effective resistance

As explained, by lowering η, we lower the amount of amplitude at starting edges $|s'u\rangle$ in $|\varphi\rangle$, which is desirable as we want $|\varphi\rangle$'s amplitude to be as concentrated at the marked edges as

possible. But recall from Chapter 3 that decreasing η (akin do decreasing C) *lowers* the success probability of phase estimation, as the overlap between the starting distribution and $|\varphi\rangle$ decreases. How should we strike a balance between these two? We follow Piddock and aim to have phase estimation be successful with probability roughly $1/2$. This is somewhat arbitrary, but we omit finding the optimal balance in this work.

How can we find an η giving a success probability of $1/2$? Recall from Section 2.1.6 that we can use amplitude estimation to estimate the probability a of obtaining a particular (set of) state(s) as outcome of a quantum algorithm. With probability at lest $8/pi^2$, the estimate $\tilde{a}$ is in $[a-\epsilon_a, a+\epsilon_a]$ for $\epsilon_a \leq \frac{\pi}{2^s} + (\frac{\pi}{2^s})^2$, where s is the number of bits we use for amplitude estimation.

We can use this on phase estimation circuit of interest to estimate the success probability for a given η. Note also that increasing η can only increase the amplitude at the starting edges, so that the success probability of phase estimation grows monotonically with η. In principle, then, if we start small η, and keep increasing it, we should eventually reach a success probability of $1/2$, and using amplitude estimation, we can detect for which η this is the case.

The only question is how long this may take: perhaps η can be anywhere in a very large range, so that searching for it like this is not feasible. That is, we need to understand more precisely how η relates to the success probability. Consider the following lemma.

Lemma 4.4.1 (Lemma 6, [Pid19])**.** Let U_A and U_B be the quantum walk operators for G' with $\{s'\} \cup M'$ the set of vertices where D_x acts trivially. Then running phase estimation on $U_A U_B$ starting in the state $|\psi_s\rangle = \sum_u \sigma_u |s'u\rangle$ for with $t = \left\lceil \log(\sqrt{\eta W + 2}/\epsilon) \right\rceil$[1] bits of precision, outputs 0^t with probability a such that

$$\frac{\eta}{R_{s',M'}} \leq a \leq \frac{\eta}{R_{s',M'}} + \epsilon$$

leaving the other register in a state $|\tilde{\varphi}\rangle$ such that

$$\frac{1}{2} || \, |\tilde{\varphi}\rangle \langle\tilde{\varphi}| - |\varphi\rangle \langle\varphi| \, ||_1 \leq \sqrt{\frac{\epsilon}{a}}.$$

This lemma tells us that the success probability is roughly equal to $\eta/R_{s',M'}$. Now, as Piddock notes, for any η we can upper-bound $R_{s',M'}$ by $\eta + R_{\sigma,M'}$. This is because $\eta + R_{\sigma,M'}$ is the energy of the unit flow from s' to M' of the form: send σ_u flow from s' to u, and then send the electrical flow from σ to M'. If follows once $\eta \geq R_{\sigma,M'}$ the success probability is at least $1/2$. This is a reasonable upper-bound. For example, in a tree, $R_{\sigma,M'}$ is upper-bounded by the tree-depth.

What remains to be filled in is the starting value for η, the amount with which we increase η, and the stopping condition. We copy Piddock's starting value of $\eta = 1/W$, and his proposal to double η at each iteration. He doesn't motivate these choices, and we also omit an analysis of their optimality, just like with the choice to seek a success probability of $1/2$.

For the stopping condition, Piddock proposes to stop once we find an estimate $\tilde{a}$ that exceeds $1/2$. In a way, this is quite terrible. Since the estimate is in $[a-\epsilon_a, a+\epsilon_a] = [\eta/R_{s',M'} - \epsilon_a, \eta/R_{s',M'} + \epsilon + \epsilon_a]$, this stopping condition guarantees that $\eta/R_{s',M'} - \epsilon_a > 1/2$, meaning $\eta/R_{s',M'} > 1/2$. We therefore always end up beyond $1/2$, while our goal is to reach $1/2$.

A small optimisation would be to stop slightly earlier: once $\tilde{a} > 1/2 - \epsilon_a$. It is easy to see that $\tilde{a} \in [1/2 - \epsilon_a, 1/2 + \epsilon + \epsilon_a]$ is the range of all $\tilde{a}$ that can correspond to $\eta/R_{s',M'} = 1/2$. By

[1] Although Piddock's statement of the lemma expresses the precision asymptotically, this exact statement follows directly from the last line of his proof.

stopping once we enter the lower-end of this range, we actually have a shot at finding exactly $\eta/R_{s',M'} = 1/2$ [2].

With this, the contour of the algorithm is defined. What is left is to guarantee that it is successful with desired probability $1-\delta$. The algorithm makes numerous amplitude estimation calls, each successful with probability at least $8/\pi^2$. We might be inclined to argue that we should amplify the probability to make sure that, with probability $1-\delta$ all successive amplitude estimation calls are correct. This would be quite problematic because - we will see this in moment - the number of required repetitions is a function of the effective resistance of the graph: which is essentially what we are estimating in this algorithm. It follows that we wouldn't be able to determine r during run-time, preventing us from configuring the algorithm adequate for our desired success probability.

Fortunately, this isn't the case. The algorithm will start by doing, say, $r-1$, amplitude estimation runs whose correct outcome is below the success threshold. If any of these early amplitude estimation calls fail, it's not necessarily a problem, as it doesn't change the trajectory of η: we always just double it. What is important is that the crucial r'th run - the first run where the η/R that we estimate goes over the success threshold - is correct with probability $1-\delta$. Of course, we don't know a priori which run this is, so to achieve this we need to amplify the success probability of each amplitude estimation run to $1-\delta$, requiring $n \in O(\log(1/\delta))$ repetitions, which can be determined exactly using Proposition A.1.1.

The one possible problem here would be if one of the earlier $r-1$ runs fails, and the inaccurate amplitude estimate is so far off, that it goes over the success threshold, making the algorithm terminate too early. However, since the error range of amplitude estimation is normally distributed, this is extremely unlikely: the probability to obtain an estimate a distance d from the correct output drops exponentially with d. Really, then, this could only happen when the earlier run was already very close to the success threshold, but not quite there. In that case, this outcome wouldn't be a problem, as we don't require the real estimate η/R to be within a fixed distance of $1/2$.

Due to lack of time, we omit the analysis of this situation, and simply assume that by setting the success probability of each amplitude estimation run to $1-\delta$, we assume that the crucial r'th outputs successfully with probability $1-\delta$ as well, i.e. we assume the above problematic situation doesn't occur.

4.4.1 Final algorithm and complexity

We can now define the algorithm. We make use here of the upper-bound on the inaccuracy of the final estimate, which we derive in Appendix A.2.1: the inaccuracy between the final $\eta/R_{s',M'}$

[2] This is not at all the only sensible option. One problem with this success range is that it doesn't terminate for η that are very close to the success range, causing η to double, possibly yielding a much too large η. A simple optimisation (which we discuss a bit later) is to just remember the previous estimate, and return the corresponding η if the final terminating estimate was much worse. Nonetheless, even then there will still be many cases where it would have been better to accept η that are just a bit below the success range. Widening the range downwards might then in some contexts be desirable.

Another problem is that, though the range encompasses all and only $\tilde{a}$ that could be approximations of the η that we are after (i.e. with $\eta/R_{s',M'} = 1/2$), the range of underlying η is quite a bit wider than just this η. Specifically, note that given a $\tilde{a}$, the possible range of $\eta/R_{s',M'}$ underlying it is $\eta/R_{s',M'} \in [\tilde{a}-\epsilon-\epsilon_a, \tilde{a}+\epsilon_a]$. It follows that the proposed success range can result in an η with $\eta/R_{s',M'} \in [1/2-2\epsilon_a-\epsilon, 1/2+2\epsilon_a+\epsilon]$. Depending on the error ranges this could get somewhat far from $1/2$. Of course, error ranges always exist, and can be reduced arbitrarily, but nonetheless, tightening the success range could be desirable.

and $1/2$ is at most $3/2 - \sqrt{2}$, and decreases as the number of starting vertices grows. There, we also propose the final addition to the algorithm: we return not the η with which we hit the accepting condition, but we return η or $\eta/2$, depending on whose estimate $\tilde{a}$ was closer to $1/2$. This roughly halves the worst-cast inaccuracy, at no extra query cost.

Algorithm 5 (Find η such that $\eta/R_{s',M'} \approx 1/2$)**.** Input: An undirected $G = (V, E)$ with weights w and marked subset $M \subseteq V$, starting distribution σ, upper-bound W on the weight of the graph. Constants s, ϵ and $n \in O(\log(1/\delta))$. Output: With probability at least $1 - \delta$: η such that $|\frac{\eta}{R_{s',M'}} - \frac{1}{2}| \leq 3/2 - \sqrt{2}$.

1. Set $x = 0$ for the rest of the algorithm. Initialise $\eta = 1/W$.
2. Construct G' and $U_A U_B$ for η and x.
3. Run amplitude estimation n times, with the all zero string as good string and s bits of precision on the second register of the phase estimation circuit with unitary $U_A U_B$, input vector $|\psi_s\rangle$ and precision $\sqrt{\eta W + 2}/\epsilon$. Let the median outcome be $\tilde{a}$.
4. If $\tilde{a} > 1/2 - \epsilon_a$ output η or $\eta/2$ depending on whose estimate $\tilde{a}$ is closer to $1/2$.
5. Double η and go back to step 2.

Each iteration, we run amplitude estimation n times, each run of which runs the phase estimation circuit $2^s - 1$ times. Phase estimation makes $2^t - 1$ queries when using t precision bits. What is t in our case? Well, the precision of phase estimation depends on the current η. Specifically, we use precision $\sqrt{\eta W + 2}/\epsilon$, so that we use $t = \lceil \log(\sqrt{\eta W + 2}/\epsilon) \rceil \leq \log(\sqrt{\eta W + 2}/\epsilon) + 1$ bits. Let r_1 be the number of iterations we do. The total number of queries is then

$$f_1(s, \epsilon, n, r_1, W) = n(2^s - 1) \sum_{i=0}^{r_1 - 1} (2^{\lceil \log\left(\sqrt{\frac{1}{W} \cdot 2^i \cdot W + 2}/\epsilon\right) \rceil} - 1) = n(2^s - 1) \sum_{i=0}^{r_1 - 1} (2^{\lceil \log\left(\sqrt{2^i + 2}/\epsilon\right) \rceil} - 1).$$

To upper-bound f_1, recall that η is upper-bounded by $O(R_{\sigma,M'})$. It follows that the maximum number of iterations is in $O(R_{\sigma,M'} W)$. Each iteration runs amplitude estimation a constant n number of times. The complexity is then dominated by the final run of phase estimation with $O\left(\sqrt{R_{\sigma,M'} W}\right)$, so that $f_1 \in O\left(\sqrt{R_{\sigma,M'} W}\right)$.

Note that s, ϵ and n are constants chosen beforehand. The only real unknown, then, are the sum of weights W and the number of iterations r. The first is easily computed classically. The second is seems trickier: it depends non-trivially on the relation between η and $R_{s',M'}$ (as this determines how fast we grow towards the success threshold). Deriving an expression for r_1 in terms of more easily found features of the graph, even the effective resistance, then seems quite hard.

Instead, there is a much simpler computational approach to find r_1: simply simulate the iterations of Algorithm 5. Since we can classically compute the effective resistance of a graph, we can run the algorithm classically by swapping the amplitude estimation call with a classical algorithm to compute the effective resistance, or rather, we first compute the effective resistance, and then divide η by it to compute $\eta/R_{s',M'}$, and we then have to round this to the estimate that amplitude estimation would have found had it used s bits of precision. Recall that amplitude

estimation returns (when it succeeds) the best estimate of the effective resistance that it can write using s bits. We can easily determine this best estimate, for instance by checking all the 2^s possible estimates of the effective resistance, and choosing the best one. Specifically, these are $sin^2(\tilde{\theta})$ where $\tilde{\theta}$ are all the 2^s equidistance points in $[0, \pi]$ (or just the first half, as $\sin^2$ is symmetric between the two halves of this range).

However, in our application (backtracking algorithm for SAT) we only consider trees. For these[3], the relation between η and $R_{s',M'}$ is simple. Recall that $R_{s',M'} \in [R_{\sigma,M'} + \eta/\sqrt{d}, R_{\sigma,M'} + \eta]$. Since a tree has a single starting vertex, this range collapses to just $R_{\sigma,M'} + \eta$: we perfectly increase the effective resistance by η. Determining r_1 and the inaccuracy is then much simpler. We want the first η such

$$\frac{\eta}{R_{\sigma,M'} + \eta} > 1/2 - \epsilon \text{ which implies } \eta > \frac{2R_{\sigma,M'}\epsilon_a + R_{\sigma,M'}}{1 - 2\epsilon_a},$$

where we note this η tends to $R_{\sigma,M'}$ as ϵ_a tends to 0[4]. For our use-case of trees, the classical algorithm to determine the number of queries that Algorithm 5 makes then becomes:

1. Count the total number of vertices in the graph, call this W. Note that this corresponds to the sum of weights, as each weight is 1, and there is one-to-one correspondence between nodes and edges.

2. Compute the effective resistance of the original graph with new marked edges of weight $1/x$ added $R_{\sigma,M'}$. Note that in our case $x = 0$, but in principle this algorithm also works for different x.

3. Compute the number of repetitions $r_1 = \left\lceil \log\left(W \frac{2R_{\sigma,M'}\epsilon_a + R_{\sigma,M'}}{1 - 2\epsilon_a}\right)\right\rceil$.

4. Compute the value of f_1 using r_1 and W.

5. Optionally: compute the value of η after r_1 iterations: $\eta = 2^{r_1}/W$, and compute the estimate of $\eta/(R_{s',M'} + \eta)$ that amplitude estimation would have output. These might be of use depending on the use-case of the algorithm. Indeed, for our subsequent use, we will need η.

Note that we haven't considered the impact of x (i.e. the weights $1/x$ of the marked edges) on $R_{\sigma,M'}$: it might be that for too large x, $R_{\sigma,M'} \notin O(R_{\sigma,M})$, so that the algorithms' complexity is no longer in terms of the effective resistance of the original graph. To guarantee the complexity in terms of the original graph, we therefore require x which give $R_{\sigma,M'} \in O(R_{\sigma,M})$. Indeed, the final search algorithm only uses such x, netting a final complexity in terms of the weight and effective resistance of the original graph.

The choice of the precision of phase estimation and amplitude estimation (i.e. the choice of ϵ and s) is not hugely relevant, and mostly up to choice depending on the what the context demands. Recall that the range from which we draw our estimates is $\tilde{a} \in [\eta/R_{s',M'} - \epsilon_a, \eta/R_{s',M'} +$

[3]Or rather, whenever there is a single starting vertex.

[4]Of course, the algorithm doesn't check this exact value for $\eta/R_{s',M'}$, but finds a rounded estimate, and it might be that, although this is the first iteration where $\eta/R_{s',M'}$ exceeds the stopping threshold, the estimate lies below the threshold. To be complexity correct, then, we should check whether this is the case, and if not, iteratively double η and stop once we do go over the stopping threshold. However, this would make the approach much more complicated, and since we will use a decently large s, this will virtually never happen.

$\epsilon + \epsilon_a]$, and this is tightened as the precisions increase. How tight this is, is not that relevant for accuracy of the final estimate, however. What is more important is the stepping size and stopping condition, as there determine how close to $1/2$ the actual value of $\eta/R_{s',M'}$ gets to be; if this is terrible, then tightening the error ranges around this value won't help much. Lowering the error ranges is thus no guarantee for success.

Recall that we derived an upper-bound on this inaccuracy of $3/2 - \sqrt{2}$. We then at least want to choose s and ϵ to prevent the error range from worsening this upper-bound. We show in Appendix A.2.2 that under only this one condition, the optimal choice becomes $s = 7$ and $\epsilon = 3/2 - \sqrt{2} - 2\frac{\pi(128+\pi)}{16384} \approx 0.03549$. We will move into our experiments with these two values. With these two fixed, we have established the following.

Theorem 4.4.2. Given an undirected graph $G = (V, E)$ with weights w and marked subset $M \subseteq V$, starting distribution σ, upper-bound W on the weight of the graph, and constants s, ϵ and $n \in O(\log(1/\delta))$. If s and ϵ are chosen so that $2\epsilon_a(s) + \epsilon \leq \epsilon_\eta(d)$, where d is the size of the support of σ, Algorithm 5 returns an η such that $|\eta/R_{s',M} - 1/2| \leq \epsilon_\eta(d) \leq 3/2 - \sqrt{2} \approx 0.0858$ with probability at least $1 - \delta$, and uses

$$f_1(s, \epsilon, n, r_1, W) = n(2^s - 1) \sum_{i=0}^{r_1 - 1} (2^{\lceil \log\left(\sqrt{2^i+2}/\epsilon\right) \rceil} - 1) \in O\left(\sqrt{R_{\sigma,M'} W}\right)$$

queries, where r_1 is the number of iterations the algorithm has to make. The cheapest configuration is $s = 7$ and $\epsilon = 3/2 - \sqrt{2} - 2\frac{\pi(256+\pi)}{65536} \approx 0.060942$ giving complexity

$$127 \sum_{i=0}^{r_1 - 1} (2^{\lceil \log\left(16.4092\sqrt{2^i+2}\right) \rceil} - 1).$$

4.5 Finding marked elements using the effective resistance estimate

As explained, by increasing x, we increase the amount of energy at the marked edges in the electrical flow (i.e. amplitude at marked edges in $|\varphi\rangle$), thereby increasing the probability that we find a marked element when collapsing $|\varphi\rangle$. But conversely, by increasing the energy at the edges, we increase the effective resistance $R_{s',M'}$, which increases the complexity of phase estimation. Thus, while a very large x may give a very high probability to find a marked vertex when measuring $|\varphi\rangle$, constructing $|\varphi\rangle$ may become so expensive (this requires phase estimation), that it would have been cheaper to have x be smaller.

How should we strike a balance between these two? Piddock's proposal is to make sure

1. not to lower x so much that the energy increase causes $R_{s',M'} \notin O(R_{\sigma,M})$, as then the final asymptotic complexity would no longer be in terms of the effective resistance $R_{\sigma,M}$ of the original graph would longer make the asymptotic complexity;

2. to still increase x far enough to guarantee a constant lower-bound on the probability to

find a marked vertex:

$$\frac{x\sum_{k\in M} f_{kk'}^2}{R_{s',M'}} \in \Omega(1).$$

An initial attempt at meeting these conditions might be to set $x \in O(R_{s',M'})$. We can find such an x by using Algorithm 5, which gives us an estimate $x \approx R_{s',M'}/2 \in O(R_{s',M'})$. This gives (i), as the extra flow at the marked edges is $x\sum_{k\in M} f_{kk'}^2 \leq x$, so that $R_{s',M'} \leq R_{s',M} + x = 2x$, which implies that $x \in O(x) = O(R_{s',M'})$. To see whether this gives (ii), note that the probability to hit a marked element becomes

$$\frac{x\sum_{k\in M} f_{kk'}^2}{R_{s',M'}} = \Omega(1)\sum_{k\in M} f_{kk'}^2.$$

This is almost a constant lower-bound on the probability to find a marked element: only $q(x) = \sum_{k\in M} f_{kk'}^2$ stands in our way. Unfortunately, the best lower-bound we can give for $\sum_{k\in M} f_{kk'}^2$ is $\Omega(1/|M|)$, which is in case when the flow is spread uniformly over all the marked edges, see Proposition A.3.1. The probability to observe a marked vertex when collapsing $|\varphi\rangle$ then becomes $\Omega(1/|M|)$, which would imply a search algorithm $O(\sqrt{R_{\sigma,W}}|M|^3)$, which isn't very good. To progress, we need to understand more deeply the relationship between f, x and R. For this, consider the following Lemma.

Lemma 4.5.1 (Lemma 7, [Pid19])**.** Let $R_{s',M'}(x)$ be the effective resistance when additional edges of resistance $1/x$ are added to each marked vertex. Then

$$\frac{d}{dx}R_{s',M'} = \sum_{k\in M} f_{kk'}^2 = q(x).$$

How is this lemma useful? Consider again the fraction denoting the probability to observe a marked vertex. The lemma tells us that the antiderivative of $q(x) = \sum_{k\in M} f_{kk'}^2$ is $R_{s',M'}$, so that if we draw x from some interval $[a, b]$, and use an integral to express the expected probability to hit a marked vertex, we divide out the $R_{s',M'}$ factor in the denominator. All that's left now is to get rid of the factor x in the numerator. We can do this by choosing the probability density function $1/(x\log(b/a))$. The expected probability to observe a marked element then becomes

$$\int_a^b \frac{xq(x)}{R_{s',M'}(x)}\frac{1}{x\log(b/a)} = \int_a^b \frac{q(x)}{\log(b/a)R_{s',M'}(x)} \geq \frac{1}{\log(b/a)R_{s',M'}(b)}\int_a^b q(x) = \frac{R_{s',M'}(b) - R_{s',M'}(a)}{\log(b/a)R_{s',M'}(b)},$$

where we note that the inequality holds because $R_{s',M'}$ grows monotonically with x. If we now fix $[a, b]$ so that $R_{s',M'}(x)$ grows by a constant multiplicative factor across the interval, say a factor of 2, we have

$$\frac{R_{s',M'}(b) - R_{s',M'}(a)}{\log(b/a)R_{s',M'}(b)} = \frac{2R_{s',M'}(a) - R_{s',M'}(a)}{\log(b/a)2R_{s',M'}(a)} = \frac{R_{s',M'}(a)}{\log(b/a)2R_{s',M'}(a)} = \frac{1}{2\log(b/a)}.$$

that is, the probability to hit a marked vertex is $\Omega(1/\log(b/a))$. Whether a different constant factor would be more optimal is a question for future research.

It remains to be shown that we can meet (i). To do so, we need to select a and b so that $R_{s',M'}(a) \in O(R_{\sigma,M})$ and $R_{s',M'}(b) \in O(R_{\sigma,M})$, as this guarantees that for any $x \in [a, b]$ we meet condition (i). We will set $a = \eta \approx R_{s',M}/2$ - i.e. the output of Algorithm 5 when using

$x = 0$ - and find b by setting $x = \eta$, iteratively doubling x, estimating $\eta/R_{s',M'}(x)$ using step 3 of Algorithm 5, and stopping once this estimate has halved compared to $\eta/R_{s',M'}(a)$ (implying the numerator has doubled).

The extra flow at the marked edges with weight $1/a$ is $a\sum_{k\in M} f_{kk'}^2 \leq a$, so that $R_{s',M'}(a) \leq R_{s',M} + a = 3/2R_{s',M} \in O(R_{s',M}) = O(R_{\sigma,M})$. Since the effective resistance doubled when going from a to b, we have $R_{s',M}(b) \in O(2R_{s',M}(a)) = O(R_{\sigma,M})$. With that, both conditions are met. In fact, for this choice of a and b, we can show that $b/a \in O(|M|)$, so that the probability that we find a marked vertex is $\Omega(1/\log(|M|))$.

Lemma 4.5.2. Let $a = \eta = R_{s',M'}/2$ and $R_{s',M'}(b) = 2R_{s',M'}(a)$. Then $b/a \in O(|M|)$.

Proof. Recall from Lemma 4.5.1 that $q(x) = x\sum_k f_{kk'}^2$ is the derivative of $R_{s',M'}(x)$. Recall from Proposition A.3.1 that $q(x) \geq 1/|M|$. We can then write

$$\frac{b-a}{|M|} \leq R_{s',M'}(b) - R_{s',M'}(a) \iff \frac{b}{a} \leq 1 + |M|\frac{R_{s',M'}(b) - R_{s',M'}(a)}{a}.$$

Applying $R_{s',M'}(b) = 2R_{s',M'}(a)$ gives

$$\frac{b}{a} \leq 1 + |M|\frac{2R_{s',M'}(a) - R_{s',M'}(a)}{a} = 1 + |M|\frac{R_{s',M'}(a)}{a}$$

and finally noting that $a = \eta = R_{s',M}/2$ so that $R_{s',M'}(a) \leq R_{s',M} + a = 3/2R_{s',M'}$ results in

$$\frac{b}{a} \leq 1 + |M|\frac{R_{s',M'}(a)}{a} \leq 1 + |M|\frac{3/2R_{s',M}}{R_{s',M}/2} = 1 + 3|M|.$$

□

Say we wish the algorithm succeeds with a probability $1 - \delta$. Just like with the previous algorithm, we assume that it suffices to amplify the success probability of each iteration to $1-\delta$, so that the crucial first run whose correct outcome goes over the success threshold is correct with probability $1 - \delta$. This algorithm also perform amplitude estimation once per iteration. Just like with Algorithm 5, then, it suffices to repeat amplitude estimation $n \in O(\log(1/\delta))$ repetitions, which can be determined exactly using Proposition A.1.1.

4.5.1 Influence of inaccuracy of a and b on probability to hit a marked vertex

Above, we derive the lower-bound of $1/(2\log(b/a))$ assuming, first, that a is exactly equal to $\eta = 1/2R_{s',M'}$ and, second, that we find b such that $2R_{s',M'}(a) = R_{s',M'}(b)$. What happens the this lower-bound if either of these conditions is relaxed?

For the first condition, recall that we always find $\eta/R_{s',M'}$ within ϵ_η of $1/2$. It follows that our estimate of $\eta = 1/2R_{s',M'}$ then diverges at most $\epsilon_\eta R_{s',M'}$ from $1/2R_{s',M'}$ in either direction. Say that we still hit the second condition with this inaccurate a. It follows that the lower-bound still comes down to $\frac{1}{2\log(b/a)}$. What does the b look like that we find when running our algorithm with this different a? Well, this in unclear, and depends on how the effective resistance changes with a, i.e. with decreasing the edge weight at the marked edges. Analytically determining the change in success probability for a different a is then something we don't attempt. But note that the quantum algorithm has a and b, and is thus able to determine this success probability.

Not only this, but as we will see below, we will be able to efficiently compute these classically, allowing us to determine the expected query count of the final search algorithm, as required.

For the second condition, it is not hard to see that finding a too large b, i.e. such that $2R_{s',M'}(a) > R_{s',M'}(b)$, increases the probability to find a marked vertex. Recall from above the the lower-bound on the probability to find a marked vertex is

$$\frac{R_{s',M'}(b) - R_{s',M'}(a)}{\log(b/a)R_{s',M'}(b)} = \frac{1}{2\log(b/a)}.$$

Let us consider what happens when we loosen the above condition to include our error. Say $R_{s',M'}(b) = 2R_{s',M'}(a) + d$, so that a positive d means b is 'too large', and vice versa.

$$\frac{R_{s',M'}(b) - R_{s',M'}(a)}{\log(b/a)R_{s',M'}(b)} = \frac{2R_{s',M'}(a) + d - R_{s',M'}(a)}{\log(b/a)(2R_{s',M'}(a) + d)} = \frac{R_{s',M'}(a) + d}{\log(b/a)(2R_{s',M'}(a) + d)}.$$

Note that $\log(b/a) > 1$ (as we double a at least once), so that when d is positive, we can double the the d in the denominator to lower-bound the success probability:

$$\frac{R_{s',M'}(a) + d}{\log(b/a)(2R_{s',M'}(a) + d} \geq \frac{R_{s',M'}(a) + d}{\log(b/a)2(R_{s',M'}(a) + d)} = \frac{1}{2\log(b/a)}.$$

In other words, when we increase b beyond the stopping condition, the probability to observe a marked vertex rises.

In the previous algorithm, we decided to move the stopping threshold in slightly to account for the error range. We might consider doing the same here, i.e. changing $\eta/R_{s',M'}(b) < \eta/2R_{s',M'}(\eta)$ to $\tilde{a} < \eta/(2R_{s',M'}(\eta)) - \epsilon_\eta(d)$ since any estimate $\tilde{a} \in [\eta/(2R_{s',M'}(\eta)) - \epsilon_\eta(d), \eta/(2R_{s',M'}(\eta)) + \epsilon_\eta(d)]$ can imply $\eta/R_{s',M'}(b) = \eta/2R_{s',M'}(\eta)$ (which is what we want to find).

In the previous algorithm, this made sense, as we wanted to estimate the effective resistance as best as possible. But now, exceeding the stopping threshold too far actually *increases* the probability to find a marked vertex. This then isn't necessarily bad. It might be: it could be that the reduction in expected query complexity due to the increased success probability is less than the extra cost to increase b further. But this is an optimisation question we don't consider in this work.

4.5.2 Final algorithm and complexity

We can now define the algorithm. We make use here of the upper-bound on the inaccuracy, which we derive in Appendix A.2.3: the inaccuracy between $\eta/R_{s',M'}(b)$ and $\eta/2R_{s',M'}(a)$ is at most $1/5$, and increases as the number of marked element grows. In particular, for $|M| = 1$, the inaccuracy is at most $3/40$.

Algorithm 6 (Find a, b such that $R_{s',M'}(b) \approx 2R_{s',M'}(a)$)**.** Input: An undirected $G = (V, E)$ with weights w and marked subset $M \subseteq V$, starting distribution σ, upper-bound W on the weight of the graph. Constants s, ϵ and $n \in O(\log(1/\delta))$. Output: With probability at least $1 - \delta$: a and b with $|\eta/R_{s',M'}(b) - \eta/2R_{s',M'}(a)| \leq \epsilon_b(|M|) = \frac{1}{5} - \frac{1}{6|M|+2} \leq 1/5$.

1. Set $x = 0$ and run Algorithm 5 with constants s, ϵ and n to find η such that $\eta/R_{s',M} \approx 1/2$.
2. Set $x = \eta$.
3. Run step 3 of Algorithm 5 with constants s, ϵ and n to estimate $\eta/R_{s',M'}(x)$.
4. Double x, return to step 3, until the output has havled, i.e. $\eta/R_{s',M'}(x) < 2\eta/R_{s',M'}(\eta)$.
5. Output $a = \eta$ and $b = x$.

We make one call to Algorithm 5 with $x = 0$, which costs $f_1(s, \epsilon, n, r_1, W)$ queries, where r_1 is the number of iterations the algorithm makes. We then do a number of repetitions of step 3 of Algorithm 5, each of which costs $(2^s - 1)(2^{\lceil \log(\sqrt{\eta W}/\epsilon) \rceil} - 1)$. Note that this is thus independent of the value of x which changes at each iteration. Let r_2 be the number of repetitions of this step. Then the total number of queries that Algorithm 6 makes is

$$\begin{aligned} f_2(s, \epsilon, n, r_2, r_1, W) &= f_1(s, \epsilon, n, r_1, W) + r_2(2^s - 1)(2^{\lceil \log(\sqrt{\eta W}/\epsilon) \rceil} - 1) \\ &= n(2^s - 1)\left(\sum_{i=0}^{r_1 - 1}(2^{\lceil \log(\sqrt{2^i + 2}/\epsilon) \rceil} - 1\right) + r_2(2^s - 1)\left(2^{\lceil \log(\sqrt{\eta W}/\epsilon) \rceil} - 1\right). \end{aligned}$$

To upper-bound f_2, recall from Lemma 4.5.2 that $b/a \in O(|M|)$. Since the number of iterations of doubling a to get to b is $\log(b/a)$, it follows that $r_2 \in O(\log(|M|))$. Since the last entry of the summation dominates the asymptotic complexity of f_2, we can upper-bound as follows:

$$f_2 \leq \sum_{j=0}^{O(\log(|M|)} O\left(\sqrt{R_{\sigma,M'}(2^j \cdot \eta)W}\right) \in O(\log(|M|)\sqrt{R_{\sigma,M}W}),$$

where we used the fact that for any x that the algorithm can come across, $R_{\sigma,M'}(x) \in O(R_{\sigma,M})$.

Note that s, ϵ and n are constants chosen beforehand. The only real unknown, then, are the sum of weights W, the number of iterations r_1 that Algorithm 5 does, and the number of repetitions r_2 of step 3 of Algorithm 5. As argued in the previous section, the first two can be computed classically. In fact, by simulating Algorithm 5, we can even determine the η and estimate of $\eta/R_{s',M'}$ that Algorithm 5 produces. It is not hard to see that we can extend this approach to Algorithm 6 to determine r_2: we take the η that the simulation of Algorithm 5 outputs, and then replace step 3, which estimates $\eta/R_{s',M'}(x)$, by the previously explained classical algorithm that produces this same estimation. Once we hit the stopping condition, we know r_2 (as well $a = \eta$, $b = x$, etc). For our use-case of trees, the classical algorithm to determine the number of queries that Algorithm 6 makes then becomes:

1. Compute W, the total number of nodes in the original graph. This corresponds to the sum of weights, as each weight is 1, and there is one-to-one correspondence between nodes and edges.

2. Simulate Algorithm 5 with $x = 0$ to determine r_1 and η. That is:
   (a) Compute the effective resistance of the original graph, $R_{\sigma,M}$.
   (b) Compute the number of iterations $r_1 = \left\lceil \log\left(W \frac{2R_{\sigma,M}\epsilon_a + R_{\sigma,M}}{1 - 2\epsilon_a}\right)\right\rceil$.
   (c) Compute the output after r_1 iterations: $\eta = 2^{r_1}/W$.
3. Set $x = \eta$. Now simulate the loop of Algorithm 6:
   (a) Compute the effective resistance $R_{s',M'}(x)$ of the original graph with new starting edge of weight $1/\eta$ and new marked edges of weight $1/x$.
   (b) Compute $\eta/R_{s',M'}(x)$: this is the value we are estimating by running amplitude estimation. Compute the rounded estimate of $\eta/R_{s',M'}(x)$ that amplitude estimation would have returned.
   (c) If this estimate has halved compared to the estimate of $\eta/R_{s',M'}(\eta)$ (computed in the first iteration of this loop), then stop. Else, double x and repeat.
4. Let r_2 be the number of iterations done in the previous loop. Compute the value of f_2 using r_1, r_2 and W.
5. Optionally: output any of the following. For our subsequent use, we will need the first and third items.
   (a) $a = \eta$ and $b = x$;
   (b) $R_{s',M'}(a)$ and $R_{s',M'}(b)$.
   (c) the estimates of $\eta/R_{s',M'}(x)$ and $\eta/R_{s',M'}(x)$.

Although this method will work for general graphs, might we again be able to simplify because we will limit ourselves in this work to trees? The relation between x and $R_{s',M'}(x)$ is certainly much simpler for trees. However, we can't apply the same trick as earlier, as it is now the number of marked elements that impact this relation, and this number of marked elements is variable. Specifically, recall from Proposition A.3.1 that $R_{s',M'}(x) \in [R_{s',M} + x/|M|, R_{s',M} + x]$. If we count the number of marked elements, then, we can limit the uncertainty, but still not determine exactly how the effective resistance changes with x.

Theorem 4.5.3. Given an undirected graph $G = (V, E)$ with weights w and marked subset $M \subseteq V$, starting distribution σ, upper-bound W on the weight of the graph, and constants s, ϵ and $n \in O(\log(1/\delta))$. If s and ϵ are chosen so that $2\epsilon_a(s) + \epsilon \leq \epsilon_\eta(d)$, where d is the size of the support of σ, Algorithm 6 returns a and b with $R_{s',M'}(b) - 2R_{s',M'}(a) = d \leq \frac{1}{5} - \frac{1}{6|M|+2} \leq \frac{1}{5}$ such that

(i) for all $x \in [a, b]$, $R_{s',M'}(x) \in O(R_{\sigma,M})$, so that for all x, phase estimation runs in $O(\sqrt{R_{\sigma,M}W})$, and Algorithm 5 runs in $O(\log(R_{\sigma,M}W))$;

(ii) drawing $x \in [a, b]$ and constructing and measuring $|\varphi\rangle$ for this choice of x returns a marked vertex with probability at least

$$\frac{R_{s',M'}(a) + d}{\log(b/a)(2R_{s',M'}(a) + d} \in \Omega\left(\frac{1}{\log(b/a)}\right) = \Omega\left(\frac{1}{\log(|M|)}\right).$$

It does this with probability at least $1 - \delta$, and uses

$$f_2(s, \epsilon, n, r_2, r_1, W) = f_1(s, \epsilon, n, r_1, W) + r_2(2^s - 1)(2^{\lceil \log(\sqrt{\eta W}/\epsilon)\rceil} - 1) \in O(\log(|M|)\sqrt{R_{\sigma,M}W},$$

where we recall that

$$f_1(s, \epsilon, n, r_1, W) = n(2^s - 1)\sum_{i=0}^{r_1-1}(2^{\lceil \log(\sqrt{2^i+2}/\epsilon)\rceil} - 1)$$

where r_2 is the number of iterations that Algorithm 6 does, and r_1 is the number of iterations that Algorithm 5 does.

4.6 Algorithm for efficient search on arbitrary graphs

Having established Theorems 4.4.2 and 4.5.3, we almost have our search algorithm. By running Algorithm 6, we find a, b such that drawing $x \in [a, b]$ using pdf $1/(x \log(b/a))$ allows us to find a marked vertex with probability at least

$$p \geq \frac{R_{s',M'}(a) + d}{\log(b/a)(2R_{s',M'}(a) + d)} =$$

where $d = R_{s',M'}(b) - 2R_{s',M'}(a)$. We can then simplify to

$$p \geq \frac{R_{s',M'}(b) - R_{s',M'}(a)}{\log(b/a)R_{s',M'}(b)}.$$

The expected number of repetitions before the algorithm finds a marked vertex is $1/p$. If we want to bound the run-time of the algorithm and terminate it after $1/p$ tries, we need to know $1/p$. Although we know a and b, we of course only have estimates of $R_{s',M'}(a)$ and $R_{s',M'}(b)$, and can therefore also only estimate the error d. Of course, we will 'simulate' Algorithm 6 to determine the expected output a and b, which also involved classically computing $R_{s',M'}(a)$

and $R_{s',M'}(b)$, so that we exactly determine d. In turn, we can exactly determine the expected number of repetitions $1/p$. However, the quantum algorithm itself will have to make due with an upper-bound on $1-p$ which it uses to bound the number of iterations it makes. Let us derive this upper-bound.

Let $\tilde{a}$ and $\tilde{b}$ be our estimates of $\eta/R_{s',M'}(a)$ and $\eta/R_{s',M'}(b)$, respectively. Recall that these estimates are drawn from the range $[\eta/R_{s',M'} - \epsilon_a, \eta/R_{s',M'} + \epsilon + \epsilon_a]$. Note that we can compute $\eta/\tilde{a}$ and $\eta/\tilde{b}$ (since we know η; we set $a = \eta$), giving estimates of the $R_{s',M'}$'s with error range

$$\left[R_{s',M'} - \frac{\eta}{\epsilon_a}, R_{s',M'} + \frac{\eta}{\epsilon + \epsilon_a}\right].$$

The error estimate $\tilde{d} = \eta/\tilde{b} - 2\eta/\tilde{a}$ that the algorithm can compute is then in the range

$$\tilde{d} \in \left[d - \frac{\eta}{\epsilon_a} - \frac{2\eta}{\epsilon + \epsilon_a}, d + \frac{2\eta}{\epsilon_a} + \frac{\eta}{\epsilon + \epsilon_a}\right].$$

The lower d, the worse. Since we don't know where we are in this range, then, we might be at the largest possible $\tilde{d}$, but have to assume that the actual underlying d is the lower possible. We should then subtract $\frac{2\eta}{\epsilon_a} + \frac{\eta}{\epsilon+\epsilon_a}$ from our estimate, to guarantee that we lower-bound the success probability. The lower-bound we work with then becomes

$$\frac{\eta}{\tilde{b}} - \frac{2\eta}{\tilde{a}} - \left(\frac{2\eta}{\epsilon_a} + \frac{\eta}{\epsilon + \epsilon_a}\right) \geq d.$$

Finally, we need to choose the precision with which we construct $|\varphi\rangle$. The more bits we use for phase estimation, the better our estimate of $|\varphi\rangle$ will be. Specifically, recall from Lemma 4.4.1 that phase estimation gives us an estimate $|\tilde{\varphi}\rangle$ such that

$$\frac{1}{2}|| \, |\tilde{\varphi}\rangle \langle\tilde{\varphi}| - |\varphi\rangle \langle\varphi| \, ||_1 \leq \sqrt{\frac{\epsilon}{a}}.$$

That is, the normalised trace distance between $|\varphi\rangle$ and its estimate is less than $\sqrt{\frac{\epsilon}{a}}$. One characterisation of the normalised trace distance between two states is that it is largest probability difference that the two states could give to the same measurement outcome [Wil16, Lemma 9.1.1]. Recall that $|\varphi\rangle$ gives us the measurement outcome that we want wish probability at least p. It then suffices to reduce the trace distance to p, so that

$$\sqrt{\frac{\epsilon}{a}} \leq p \iff \frac{\epsilon}{a} \leq p^2 \iff \epsilon \leq ap^2 \leq p^2 \in O(\log^2(1/|M|)).$$

Algorithm 7 (Find a marked vertex)**.** Input: An undirected $G = (V, E)$ with weights w and marked subset $M \subseteq V$, starting distribution σ, upper-bound W on the weight of the graph. Constants s, ϵ and $n \in O(\log(1/\delta))$. Output: With probability at least $1-\delta$: an element $m \in M$.

1. Run Algorithm 6 with s, ϵ and n to find a, b and estimates $\tilde{a}, \tilde{b}$ of $\eta/R_{s',M'}(a), \eta/R_{s',M'}(b)$, respectively.
2. Let $d = \eta/\tilde{b} - 2\eta/\tilde{a} - \left(\frac{2\eta}{\epsilon_a} + \frac{\eta}{\epsilon+\epsilon_a}\right)$.
3. Let $p = \dfrac{R_{s',M'}(a) + d}{\log(b/a)(2R_{s',M'}(a) + d)}$.
4. Repeat (at most $1/p$ times):
   (a) Sample $x \in [a, b]$ using pdf $1/(x \log(b/a))$.
   (b) Construct $U_A U_B$ using $\eta = a$ and x and run phase estimation with precision $\sqrt{\eta W + 2})/p^2$ to construct $|\varphi\rangle$.
   (c) Measure $|\varphi\rangle$ and check if this yields a marked element. If so, output it.

Recall that for appropriate n, Algorithm 6 is correct with probability $1 - \delta$. Conditioning on this correct outcome, we know that p really is a lower-bound on the probability to find a marked vertex when measuring $|\varphi\rangle$, so that we really expect to find a marked vertex after $1/p$ repetitions.

Now for the complexity. We of course run Algorithm 6, taking over its complexity $f_2(s, \epsilon, n, r_2, r_1, W)$ as derived earlier. Constructing $|\varphi\rangle$ means calling phase estimation with $\sqrt{\eta W + 2})/p^2$ bits. The expected number of times we need to measure $|\varphi\rangle$ before finding a marked vertex is $1/p$, so that we express the expected query complexity as

$$f_3(s, \epsilon, n, p, r_2, r_1, W) = f_2(s, \epsilon, n, r_2, r_1, W) + \left\lceil \frac{1}{p} \right\rceil (2^{\lceil \log(\sqrt{\eta W+2})/p^2) \rceil} - 1),$$

where

$$p = \frac{R_{s',M'}(b) - R_{s',M'}(a)}{\log(b/a) R_{s',M'}(b)}.$$

Now, if we bound the run-time of Algorithm 7 using the lower-bound $\tilde{d}$ on d (i.e. we measure $|\varphi\rangle$ at most $1/p$ times) then the above underestimates the total complexity after $1/p$ runs: using $\tilde{d}$ means underestimating p and therefore overestimating $1/p$. However, this doesn't change expected number of iterations, which is $1/p$ regardless of how the algorithm estimates $1/p$. We omit an analysis of the exact number of queries in the worst-case, i.e. using the largest possible $1/p$.

We can easily upper-bound f_3:

$$f_3 \in O\left(\frac{1}{p}\frac{\sqrt{R_{\sigma,M} W}}{p^2}\right) = O\left(\frac{1}{p^3}\sqrt{R_{\sigma,M}}\right) = O(\sqrt{R_{\sigma,M} W} \log^3(b/a)) = O(\sqrt{R_{\sigma,M} W} \log^3(|M|)).$$

Note that s, ϵ and n are constants chosen beforehand. The only real unknown, then, are the sum of weights W, the function $r_1(x)$ giving the number of iterations of Algorithm 5 when using x,

the number of iterations r_2 of this algorithm and p. As argued in the previous section, the first three can be computed classically by simulating Algorithm 6. We also saw that this simulation can output us, among other things, the a and b that Algorithm 7 would have found, as well as $R_{s',M'}(a)$ and $R_{s',M'}(b)$. With these, then, we can compute p, which is the last unknown. For our use-case of trees, the classical algorithm to determine the expected number of queries that Algorithm 7 makes then becomes:

1. Simulate Algorithm 6 to determine $W, r_1, r_2, a, b, R_{s',M'}(a)$ and $R_{s',M'}(b)$.
2. Compute $d = R_{s',M'}(b) - 2R_{s',M'}(a)$ and
$$p = \frac{R_{s',M'}(a) + d}{\log(b/a)(2R_{s',M'}(a) + d)}.$$
3. Compute the value of f_3.

Theorem 4.6.1. Given an undirected graph $G = (V, E)$ with weights w and marked subset $M \subseteq V$, starting distribution σ, upper-bound W on the weight of the graph and constants s, ϵ and $n \in O(\log(1/\delta))$. If s and ϵ are chosen so that $2\epsilon_a(s) + \epsilon \leq \epsilon_\eta(d)$, where d is the size of the support of σ, Algorithm 7 returns a marked vertex $m \in M$ with probability at least $1 - \delta$ and uses an expected number of queries

$$f_3(s, \epsilon, n, p, r_2, r_1, W) = f_2(s, \epsilon, n, r_2, r_1, W) + \left\lceil \frac{1}{p} \right\rceil (2^{\lceil \log(\sqrt{\eta W + 2})/p^2) \rceil} - 1) \in O(\sqrt{R_{\sigma,M} W} \log^3(|M|)).$$

We recall that
$$p(a, b, R_{s',M'}(a), R_{s',m'}(b)) = \frac{R_{s',M'}(a) + d}{\log(b/a)(2R_{s',M'}(a) + d)},$$
that $d = R_{s',m'}(b) - 2R_{s',m'}(a)$, and finally that

$$\begin{aligned} f_2(s, \epsilon, n, r_2, r_1, W) &= f_1(s, \epsilon, n, r_1, W) + r_2(2^s - 1)(2^{\lceil \log(\sqrt{\eta W}/\epsilon) \rceil} - 1) \\ &= n(2^s - 1)(\sum_{i=0}^{r_1 - 1} (2^{\lceil \log(\sqrt{2^i + 2}/\epsilon) \rceil} - 1) + r_2(2^s - 1)(2^{\lceil \log(\sqrt{\eta W}/\epsilon) \rceil} - 1), \end{aligned}$$

where r_2 is the number of iterations that Algorithm 6 does, and r_1 is the number of iterations that Algorithm 5.

Chapter 5

Experiments

In this chapter we put the expressions for the complexities of Belovs' and Piddock's quantum walk search algorithms to the test, by comparing a classical DPLL backtracking algorithm for SAT to a quantum backtracking algorithm based on Belovs' and Piddock's quantum walk search algorithms. In section

5.1 we outline specifically what algorithms we will study, and recall how we can determine their query complexity classically;

5.2 we outline the 3SAT instances that we test on'

5.3 we outline the query complexities of all these algorithms on these problem instances;

5.4 we discuss the results and draw some conclusions.

5.1 The algorithms

Below we present the algorithms for 3SAT we will test. Recall that queries in the context of a backtracking algorithm like 3SAT means queries to the heuristic h and predicate P, c.f. Section 3. We consider some configurations of Belovs' detection algorithm, but our real interest is in search algorithms (specifically, to find a single marked element, not all marked elements). The inclusion of Belovs' detection algorithm only serves its inclusion in binary search algorithms. For each quantum algorithm, we use a success probability $1 - \delta = 1 - 1/1000$.

1. The DPLL algorithm, see Section 2.3.3. This is the classical backtracking algorithm that we will try to obtain a quantum speed-up over. The query complexity is the size of the backtracking tree upon termination.

   We used the $C++$ implementation of [AC22] whose source code can be found at `https://github.com/andricicezar/truesat/tree/master/cpp_solver`. We modified it to output its size upon termination, but also the size of the fully expanded backtracking tree, the depth of the tree, the depth of the first found solution, as well as the graph itself. We will need all of these to compute the complexities of the later quantum algorithms.

2. Belovs' detection algorithm. Recall from Theorem 3.4.1 and Table 3.1 that the optimal configuration for $\delta = 1/1000$ is by setting $C = 10.394$ and $n = 9$, which gives a query complexity of

$$f(C, n, R, W) = 9(2^{\lceil \log \sqrt{11.394 + 118.43RW} \rceil - 1} - 1).$$

To determine the number of queries, we then have to provide upper-bounds W and R on the sum of weights and effective resistance of the backtracking tree. Since the weights of each edge in the backtracking tree is 1, this comes down to, respectively, upper-bounds on the size and depth of the full backtracking tree of DPLL. We consider two choices of these bounds.

(a) The naive configuration $R = n_{\text{vars}}$ and $W = 2^R$. We are then effectively doing Grover search (see Section 2.1.4) over the entire backtracking tree.

(b) The optimal, but unrealistic configuration, where we set W equal to the size of the full backtracking tree, and R equal to the effective resistance $R_{\sigma,M}$ of the full backtracking tree. In fact, when the instance is unsatisfiable, we even set $R = 1$, as the effective resistance to marked elements is undefined, and when the instance is satisfiable, we set $W = 1$, as the success probability in the positive case is independent of the upper-bound W (see item (iv) in this enumeration). This algorithm is completely unrealistic, but serves as a useful benchmark: this is the result we are tending towards by trying to set the upper-bounds. Note that we can't use only the size of the backtracking tree upon finding the first solution for W, as the quantum algorithm walks on the full backtracking tree, unable to discriminate it from the subtree of the first solution.

3. Belovs' detection algorithm, as configured by Belovs, i.e. increasing C_1 and C_2 to amplify success probability, see Section 3.3.1, and as configured by Montanaro, i.e. repeating the algorithm to amplify success probability, see Section 3.3.2. We include these to see the performance increase due to our optimisation (see Section 3.4).

4. Binary search using the detection algorithm with optimal upper-bounds. Theorem 4.1.1 tells us that the query complexity of this is given by

$$f_{\text{binary search,unsat}}(C, n, R, W, r, n_2) = \lceil n_2 \rceil \cdot f(C, n, R, W) \in O\left(\sqrt{RW} \log\log W\right)$$

and

$$f_{\text{binary search,sat}}(C, n, R, W, r, n_2) = \lceil n_2 \rceil \cdot r \cdot f(C, n, R, W) \in O\left(\sqrt{RW} \log W \log\log W\right).$$

We already know C, n, R, W. As explained above, we let the DPLL algorithm output the depth of the first solution r. This just leaves n_2. As explained in Theorem 4.1.1, we find n_2 by setting $r_{\max}$ to the number of variables in the problem instance and solving

$$1 - \sqrt[r_{\max}]{1-\delta} \leq (1-p)^{n_2} \sum_{i=0}^{n_2/2} \binom{n_2}{i} \left(\frac{p}{1-p}\right)^i.$$

5. Binary search, where we estimate W. This would be an actually realistic algorithm, as we can't simply assume that we know W. Montanaro gives us a way to estimate W [Mon18, p. 9]). First, set $W = 1$. Run the search algorithm. If you find a marked element, check if it is marked. If not, double W, and repeat. If you don't find a marked element, output that no marked elements exist. Keep going, until eventually W reaches the actual size of the tree, and we expect to output correctly with our desired probability. This requires only logarithmically many repetitions in W.

To see why this method works, we recall the proof of Section 3.2 and note that W only occurs in the correctness proof of the negative case. Indeed, when the algorithm is given a satisfiable instance, it will solve it with the given success probability $1 - \delta$, regardless of the provided W. Thus, it follows that when we try a too small upper-bound W, the worst that can happen is that the negative case fails, i.e. that we have an unsatisfiable instance and the algorithm claims it is satisfiable. However, since we now have a search algorithm, we can just check whether the vertex that is returned really is marked.

In short, when we have a satisfiable instance, the algorithm will be correct on the first try (i.e. when using initial upper-bound $W = 1$), so that not knowing W doesn't incur any extra cost. By contrast, the algorithm will not output correctly and will instead fail repeatedely until we reach a large enough W, specifically after $\lceil \log(W) \rceil$ repetitions (though it is not quite as bad a doing this many repetitions of the binary search algorithm, as we start with small W at each iteration). At each such repetition, the algorithm will walk from root down to a leaf. Since we don't know the behaviour of this walk, we for now assume that the number of steps is the worst-case, i.e. the depth of the tree. The complexity overhead then becomes at most $\lceil \log(W) \rceil$ times the depth of the tree. If we study the way an unsatisfiable instance will traverse the tree when it binary searches (which might be far less than the full depth), we may be able to lower this second factor and make the expression exact, but we ignore this for now.

Note, finally, that we also need to provide the upper-bound R. For now, we simply set this to the depth of the tree.

6. Piddock's search algorithm, see Section 4.6. Recall from Theorem 4.6.1 that its query complexity is given by the function $f_3(s, \epsilon, n, p, r_2, r_1, W)$. We proposed the classical algorithm Algorithm 4.6 which takes the graph in question (in our case, the classical backtracking tree that let DPLL output) and computes p, r_2 and r_1 [1]. We noted in Theorem 4.4.2 that the cheapest configuration of the first two constants is $s = 7$ and $\epsilon \approx 0.03549$. Since we already computed W above, this just leaves n.

   As noted in Theorem 4.6.1, we need to choose $n \in O(\log(1/\delta))$ to guarantee a success probability of $1 - \delta$. More precisely, Proposition A.1.1 tells us that the smallest n that suffices for this is the solution to

$$\delta \leq (1-p)^n \sum_{i=0}^{n/2} \binom{n}{i} \left(\frac{p}{1-p}\right)^i = (1 - \frac{8}{\pi^2})^n \sum_{i=0}^{n/2} \binom{n}{i} (\frac{8/\pi^2}{1 - 8/\pi^2})^i,$$

   where we use the success probability $8/\pi^2$ of amplitude estimation. In Table 5.1, we note the values of the right-side of this inequality for some n. It shows, in particular, that to achieve our desired $\delta \leq 1/1000$, we need to set $n = 9$.

[1] Some small notes on our implementation of this: we don't let DPLL output the entire classical backtracking tree, as any branch to a non-solution is not relevant for computing the effective resistance. We therefore only save the paths to marked vertices. We save the found solutions p, r_2 and r_1, so that we can later recompute f_3 easily for different choices of ϵ and s. A slight caveat here is that changing s changes how we round effective resistance estimates in the algorithm, which can influence p, r_2 and r_1. To allow for this, we also save the underlying values that we round using s, so that we can really recompute f_3 for a different s. Specifically, we should save η and each and each $R_{s',M'}(x)$, allowing us to compute each $\eta/R_{s',M'}(x)$, which we can then round using the new s. It might sometimes be that for a different s we require additional iterations of doubling x, so we should also save the graph itself. Finally, we should save the final x, i.e. b, so that we can recompute p if needed.

n	Upper bound on δ	
1	0.1895	
3	0.02909	$< 1/10$
5	0.006263	$< 1/100$
7	0.001087	
9	0.0001748	$< 1/1000$
11	0.00002711	$< 1/10000$
13	0.000004119	$< 1/100000$

Table 5.1: The error probability of amplitude estimation after performing a certain number of repetitions and outputting the median outcome.

n_{vars}	50	75	100	125	150	175	200	225
n_{clauses}	218	325	430	538	645	753	860	960
Satisfiable instances	1000	100	100	100	100	100	100	100
Unsatisfiable instances	1000	100	100	100	100	100	100	100

Table 5.2: Overview of the 3SAT instances, sourced from the SATLIB project, see `https://www.cs.ubc.ca/~hoos/SATLIB/benchm.html`. The instance were generated uniformly at random: each literal was selected uniformly at random from the set of all variables and their negations, while clauses containing duplicate literals and or tautological clauses (i.e. containing a literals and its negation) are rejected.

One addition we propose here is the run Belovs' detection algorithm once prior to running Piddock's algorithm. This detections whether an instances is satisfiable at all, so that for unsatisfiable instances, we don't have to run Piddock's more expensive algorithm. This is cheaper than using Piddock's algorithm to decide unsatisfiable instances: you would then have to finish performing Algorithm 5: in case the instance is unsatisfiable, this algorithm won't terminate in the expected time, after which you could conclude that the instance really is unsatisfiable.

5.2 The data

The 3SAT instances that we test on are sourced from the the SATLIB project, see `https://www.cs.ubc.ca/~hoos/SATLIB/benchm.html`. They contain varying n_{vars} and n_{clauses} and were generated uniformly at random. Specifically, each literal was selected uniformly at random from the set of all variables and their negations, while clauses containing duplicate literals and or tautological clauses (i.e. containing a literals and its negation) are rejected. For each n_{vars}, the corresponding n_{clauses} is in the so-called phase transition region: being far below this region implies a very high probability of satisfaction, being above this region implies a very low probability of satisfaction[2]. Table 5.2 for an overview of the set of our 3SAT instances.

[2]See the paragraph labelled "Phase transitions" of `https://www.cs.ubc.ca/~hoos/SATLIB/benchm.html`.

5.3 The results

We now look at the query complexity of all the algorithms listed in the previous section, on all the instances in the data set described above. Unfortunately, classically computing all the parameters necessary estimate to compute the query complexity f_3 of Piddock's algorithm was feasible only for instances in up to 125 variables. For 150 and 175 variables, even after almost a day of computing, just under half of the 100 instances had finished, and these were the instances with the smallest backtracking tree. Some other algorithms (for example, the detection algorithm using optimal upper-bounds) require the effective resistance of the original graph. We were not able to compute this for instances larger than 175 variables.

Let us first compare the classical DPLL algorithm with the two base detection algorithm (i.e. Grover, and optimally tight bounds). We plot their query complexities over the number of variables in the sat instances in Figure 5.1.

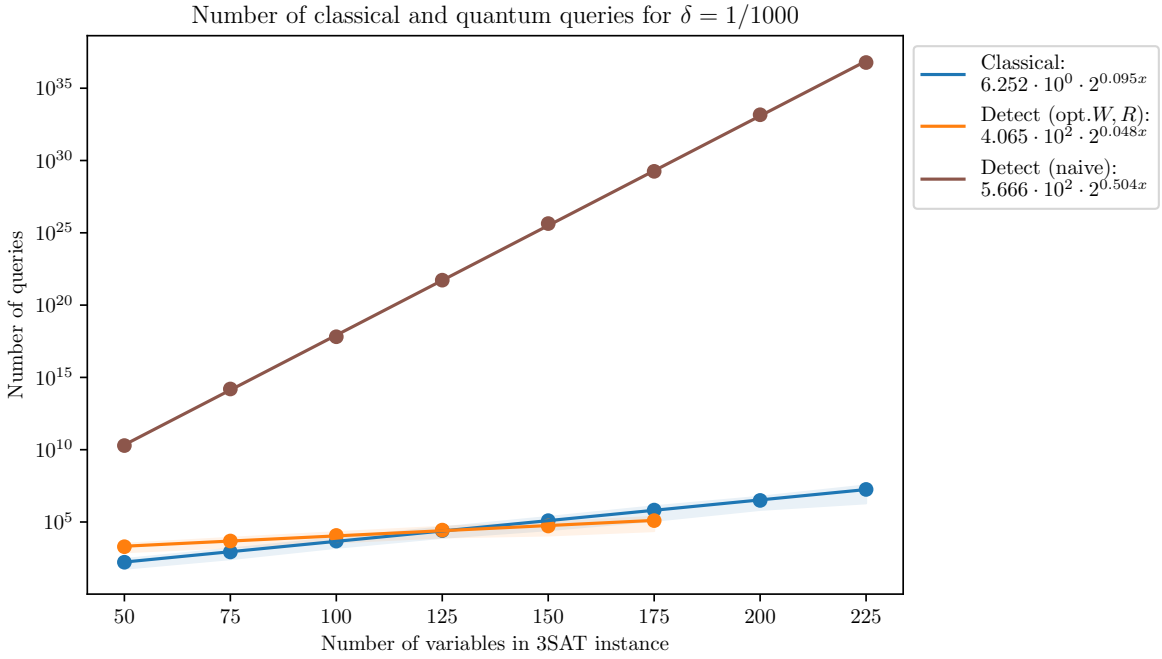

Figure 5.1: Average query complexity over the number of variables in input 3SAT instances for the classical DPLL algorithm, the base quantum detection algorithm with naive upper-bounds on W and R, and with tight upper-bounds on W and R. The lines are exponential fits to the data points and the shaded area is the standard deviation.

We can see that the naive detection algorithm really is terrible, scaling far worse than either the DPLL algorithm or the optimal detection algorithm. This was to be expected: the full search space is 2^x, where x is the number of variables in the SAT instance. The naive algorithm square roots this size, and then adds the additional overhead that the quantum algorithm suffers. By contrast, both the DPLL algorithm and the detection algorithm using optimal bounds use the much smaller actual tree size that DPLL induces.

Indeed, we see that the detection algorithm using optimal bounds does relatively well. It

scales significantly better than the classical algorithm, so that despite having a somewhat larger base complexity, it obtains a quantum speed-up starting at instances in 125 variables. Instances of this size are still easily solved by the classical DPLL algorithm on an average computer, despite the exponential complexity.

Of course, this algorithm is very unrealistic, as we can't perfectly determine the two upper-bounds at no extra cost. Nonetheless, the algorithm provides a useful baseline: quantum speed-ups are possible here, even for relatively small instances. It is now a matter of trying to find these upper-bounds at a low-enough cost to retain the speed-up. Next, in Figure 5.2, we compare the complexity of the detection algorithm to the suboptimal configurations that Belovs and Montanaro use, to see the extent of our optimisation.

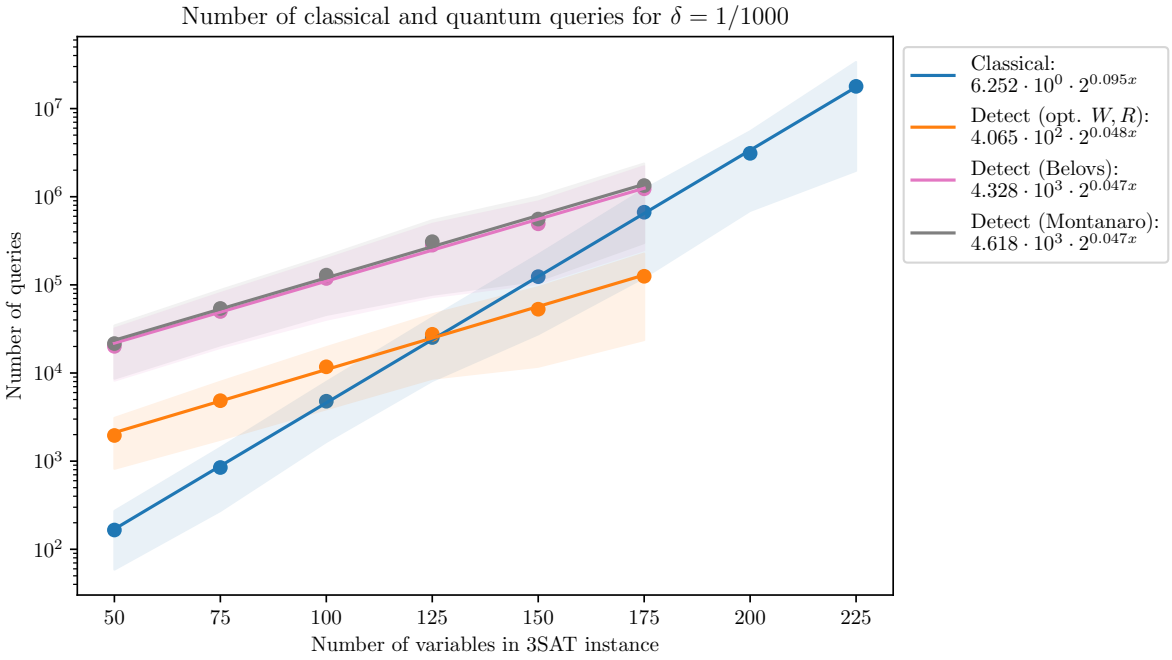

Figure 5.2: Average query complexity over the number of variables in input 3SAT instances for the classical DPLL algorithm, the base quantum detection algorithm with naive upper-bounds on W and R, and the same base detection algorithm but using Belovs' and Montanaro's suboptimal configurations (see Sections) with tight upper-bounds on W and R. The lines are exponential fits to the data points and the shaded area is the standard deviation.

Interestingly, Belovs and Montanaro's approaches yield an almost equal complexity. As expected, the scaling of their approaches is equal to that of our algorithm, as the overhead depends only on the success probability which is constant. Nonetheless, the base complexity is reduced by an order of magnitude, making our algorithm roughly 10 times as fast across all input sizes.

We also see that the standard deviation for all algorithms is quite significant. This is due to the difference in complexity between satisfiable and unsatisfiable instances. From now on, then, let us split up the complexity into these two groups of instances. Next, in Figure 5.3, we compare the classical, optimal detection, and binary search algorithm, where we split the

complexity of the latter two into satisfiable and unsatisfiable.

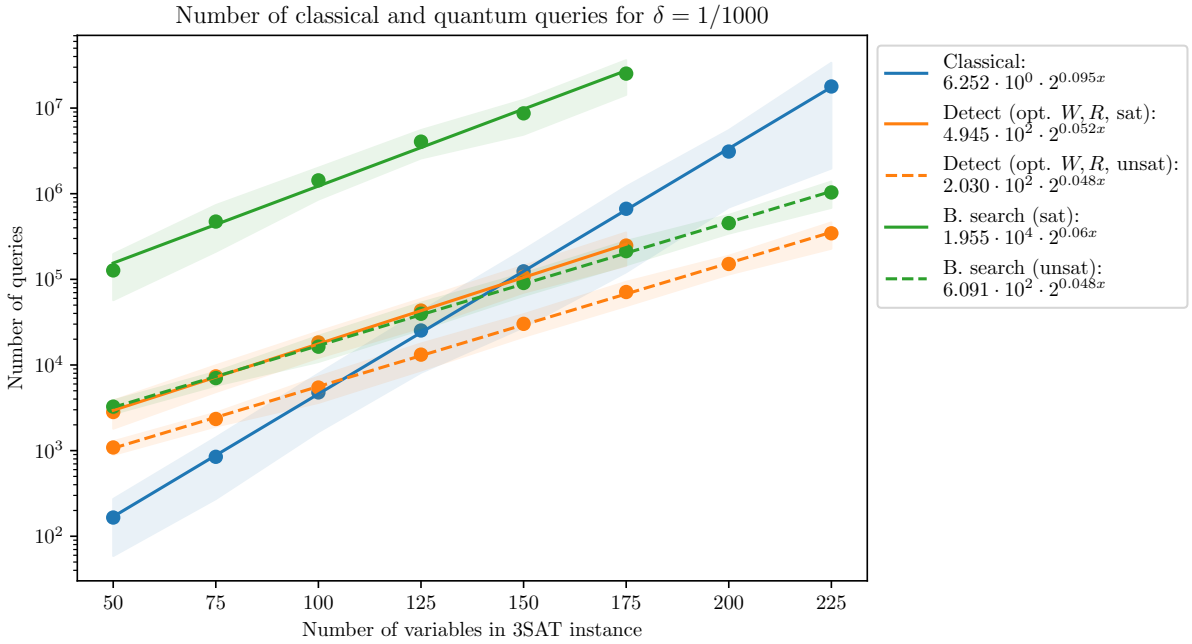

Figure 5.3: Average query complexity over the number of variables in input 3SAT instances for the classical DPLL algorithm, the base quantum detection algorithm with naive upper-bounds on W and R, and the binary search algorithm using this detection algorithm. For the latter two, we plot satisfiable and unsatisfiable instances separately. The lines are exponential fits to the data points and the shaded area is the standard deviation.

Recall that for unsatisfiable instances, going from detection to binary search just means suffering a small number of repetitions to amplify the success probability, explaining the constant increase for those instances. Across all instances we tested, this came down to a constant overhead [3]. By contrast, for satisfiable instances, there is more non-trivial dependence on the depth of the first found solution. We see this reflected in a significantly increased scaling, as well as a more complex pattern of the data points, that are more distant from the fitted line. In addition, the base complexity has increased, so that for the satisfiable binary search algorithm, a quantum speed-up has disappeared from our plot. Nonetheless, extrapolating our lines implies that the quantum speed-up should be recovered for instances in roughly 330 variables.

Next, in Figure 5.4, we consider the binary search algorithm that estimates the upper-bound on W.

Note that for satisfiable instances, estimating W comes at no extra cost, so that binary search and binary search while estimating W perfectly coincide for satisfiable instances. By contrast, unsatisfiable instances suffer a significant slowdown when estimating W, and, coincidentally, it makes their complexity roughly equal to that of satisfiable instances. This is really quite surprising: the satisfiable instances' slowdown comes from having to repeat while walking to the

[3] Indeed, as already hinted at in Section 4.1, for none of the instances we tested there was a difference in the required number of repetitions when we assumed we required n_{var} correct repetitions with probability $1 - \delta$, or just r correct repetitions with probability $1 - \delta$, where r is the actual depth of the first found solution.

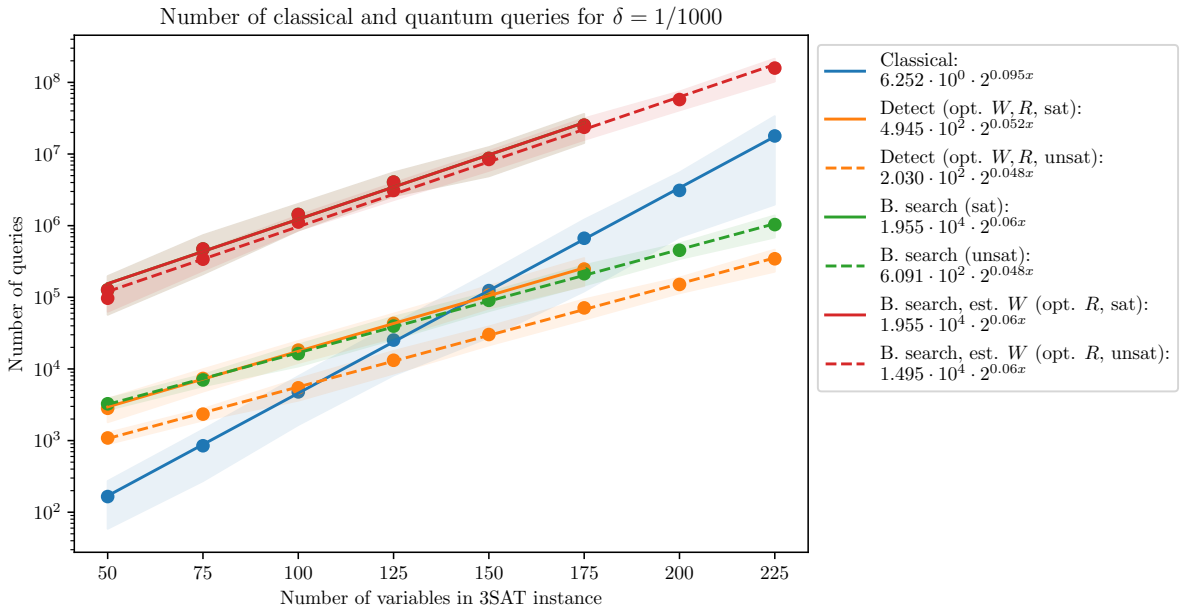

Figure 5.4: Average query complexity over the number of variables in input 3SAT instances for the classical DPLL algorithm, the base quantum detection algorithm with naive upper-bounds on W and R, and the binary search algorithm using this detection algorithm, and the same binary search algorithm but which estimates the upper-bound W. For the latter three, we plot satisfiable and unsatisfiable instances separately. The lines are exponential fits to the data points and the shaded area is the standard deviation.

depth of the first solution. The unsatisfiable instances' slowdown comes from having to repeat roughly $\log W$ times, and each time also walking down the tree, but now to the full depth of the tree (this is an assumption we had to make, making this complexity the first upper-bound we see: all the previous complexities were exact). The unsatisfiable instances then has to do roughly $\log W$ more repetitions, but because all but the last repetitions use a much smaller W, their complexity is smaller, so that apparently, their sum is roughly equal to that of satisfiable instances.

The scaling of the binary search algorithm that estimates W, then, is unchanged from the binary search algorithm for satisfiable instances, and we reiterate that by extrapolating we expect to see a quantum speed-up at instances in some 330 variables. Now, this algorithm is still not yet realistic, as we still assume an optimal upper-bound on R. Let us loosen this, and set R equal to the number of variables in the instance: a trivial upper-bound on the effective resistance. The range between these two algorithm is then where a real quantum algorithm would lie, depending on how well you upper-bound R. For comparisons sake, we also consider the binary search algorithm with the trivial upper-bound $R = n_{\text{vars}}$ for Belovs' and Montanaro's configuration, to see how much our optimisation mattered in the final usable quantum algorithm. This is shown in Figure 5.5.

We see that the algorithm with the trivial upper-bound on R scales slightly worse and has a slightly larger base complexity than the algorithm using optimal R. Extrapolating the lines implies that now, a quantum speed-up will appear starting at instances in roughly 545 variables.

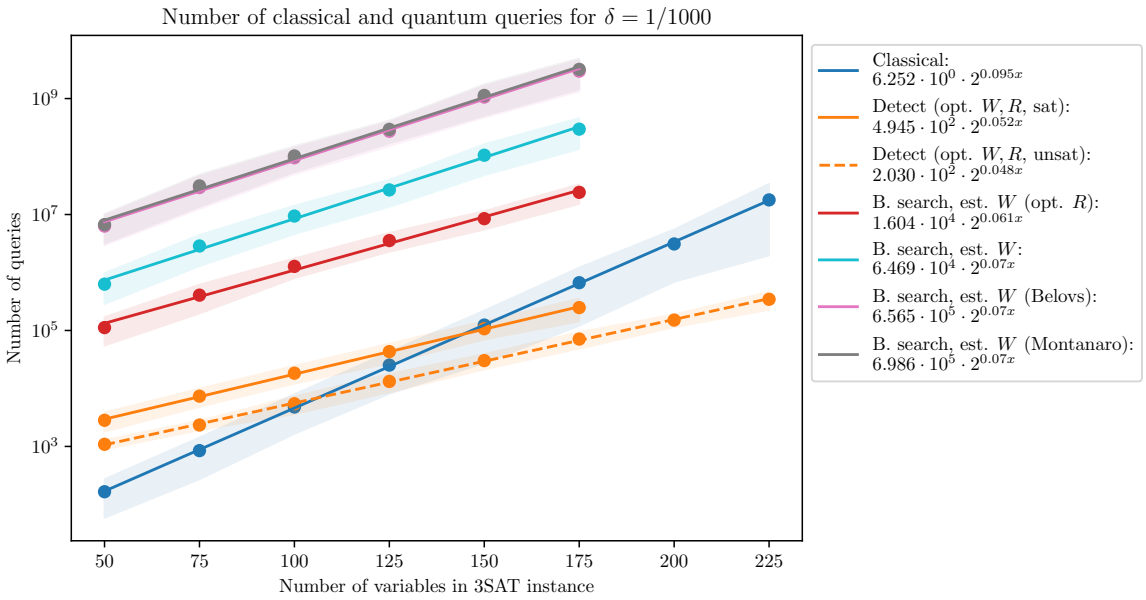

Figure 5.5: Average query complexity over the number of variables in input 3SAT instances for the classical DPLL algorithm, the same binary search algorithm that estimates W using an optimal upper-bound R, the same algorithm but with a the worst-case upper-bound R, and the same algorithm but with the detection algorithm configured as Belovs and Montanaro would configure it. The lines are exponential fits to the data points and the shaded area is the standard deviation.

We stress here that we have used quite a loose upper-bound, and that the effective resistance is almost always far, far lower than the number of variables in the formulas, if only because backtracking trees almost never have full depth.

Further, we see that the our optimisation still saves roughly a factor 10 compared to Belovs' and Montanaro's configurations of the base detection algorithm. Extrapolating their complexity implies a quantum speed-up will appear starting at instances in roughly 640 variables. To finish off, in Figure 5.6 we compare the complexity of Piddock's algorithm with the optimal binary search algorithm.

We see that for satisfiable instances Piddock's algorithm scales slightly better than the binary search algorithm with trivial upper-bound on R. Piddock's algorithm scales roughly the same as the binary search algorithm using optimal upper-bound on R, but with a larger base complexity. Extrapolating the line implies that Piddock's algorithm will obtain a quantum speed-up at instances in roughly 425 variables: almost in the middle between the 330 variables given by the optimal upper-bound R and the 545 variables given by the worst-case upper-bound R. Depending on how well one picks R, then, using binary search may be (significantly) faster than using Piddock's algorithm.

Now, we should note here that Piddock's algorithm still requires an upper-bound on W. Determining this is trickier than for the detection algorithm, however. Though Piddock's algorithm only uses W to fix the precision of phase estimation, it is not clear that the positive case is independent on this upper-bound (like with the detection algorithm). If this is not the

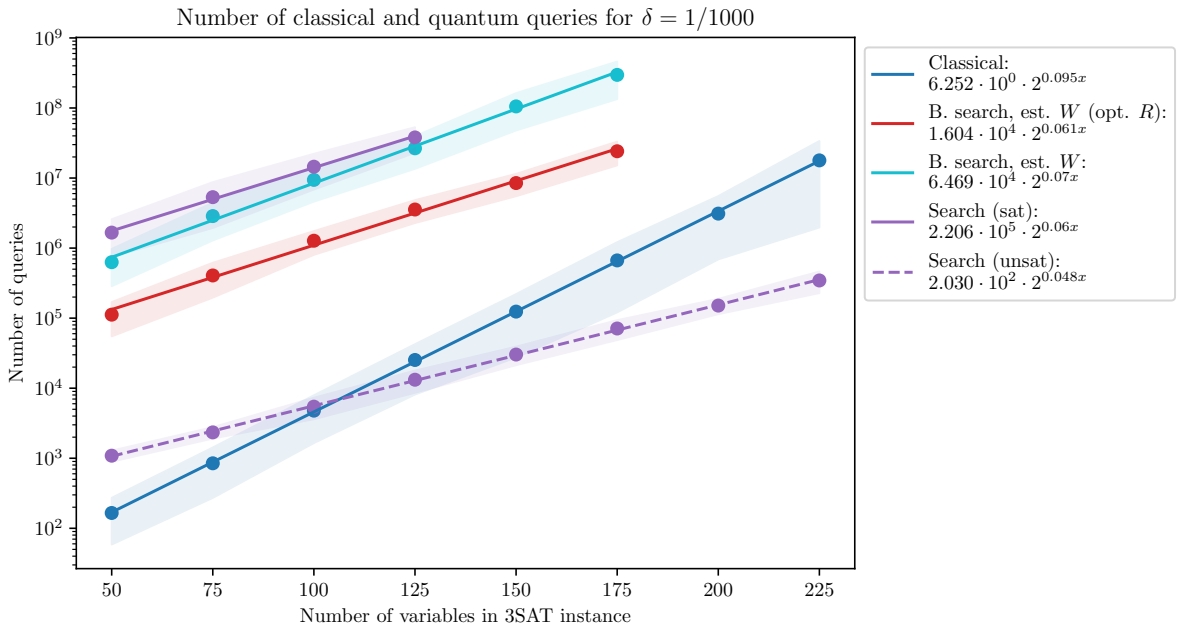

Figure 5.6: Average query complexity over the number of variables in input 3SAT instances for the classical DPLL algorithm, the same binary search algorithm that estimates W using an optimal upper-bound R, the same algorithm but with a the worst-case upper-bound R, and Piddock's search algorithm, split into satisfiable and unsatisfiable instances. The lines are exponential fits to the data points and the shaded area is the standard deviation.

case, it is not clear how we can estimate W. Furthermore, even if it is, the overhead caused to the unsatisfiable is less clear, as a failing phase estimation subroutine means Algorithms 5 and 6 may show quite different behaviour, for example using far more iterations. We omit this analysis, and simply note that Piddock's algorithm most likely will worsen after factoring in the cost to estimate W. In particular, we expect the complexity for unsatisfiable instances to worsen significantly, so that Piddock's complexity on satisfiable instances is more representative of the actual complexity, keeping the overhead for estimating W at the back of our head.

5.4 Discussion

To summarise, we see that for uniformly randomly generated 3SAT instances both a binary search algorithm based on Belovs' detection algorithm and Piddock's search algorithm achieve a quantum speed-up over the classical DPLL algorithm. This is not directly observed experimentally, but only based on extrapolating the scaling of the query complexities we observe.

Specifically, the quantum query complexity of the detection-based binary search algorithm is between $1.604 \cdot 10^4 \cdot 2^{0.061x}$ and $6.469 \cdot 10^4 \cdot 2^{0.07x}$, depending on how tight the upper-bound R is. It then obtains a quantum speed-up between

$$\frac{6.252 \cdot 2^{0.095x}}{1.604 \cdot 10^4 \cdot 2^{0.061x}} = 3.898 \cdot 10^{-4} \cdot 2^{0.034x} \quad \text{and} \quad \frac{6.252 \cdot 2^{0.095x}}{6.469 \cdot 10^4 \cdot 2^{0.07x}} = 9.665 \cdot 10^{-5} \cdot 2^{0.025x},$$

which, modulo constants, means a polynomial speed-up of an order between $0.095/0.061 \approx 1.56$ and $0.095/0.07 \approx 1.36$, which starts from instances in roughly 330 variables and roughly 545 variables.

Piddock's algorithm, where we don't yet account for the cost for estimating W, has a complexity of roughly $1.042^x 6.469 \cdot 10^4$. This is then the optimal complexity of this algorithm which we, in all likelihood, won't quite achieve. The quantum speed-up is then at most

$$\frac{6.252 \cdot 2^{0.095x}}{1.05^x 6.469 \cdot 10^4} = 2.205 \cdot 10^5 \cdot 2^{0.06x},$$

which, modulo constants, means a polynomial speed-up of order at most $0.095/0.06 \approx 1.58$, which starts form instances in (at least) roughly 430 variables.

Interestingly, Piddock's algorithm doesn't perform much better than the binary search algorithm. Indeed, perhaps it won't even perform better, after the cost of estimating W is taken into account. This is despite its superiority in asymptotic complexity. We can explain this from the significant additive overhead that Piddock's algorithm suffers. In essence, after the algorithm determines the range $[a, b]$, Piddock's algorithm simply runs phase estimation (i.e. the detection algorithm) repeatedly, just like the binary search algorithm. However, Piddock's algorithm only needs $O(\log |M|)$ repetitions, instead of binary search's $O(\log W)$. This is the source of the asymptotic superiority of Piddock. However, before Piddock can start doing this, it needs significant work to determine the range $[a, b]$, which the binary search algorithm doesn't have to.

What this estimation comes down to - in part - is estimating the effective resistance. This is somewhat that we don't do for binary search, as we have to provide the binary search algorithm with an upper-bound on the effective resistance ourselves. In a way, we require Piddock's algorithm to perform work that we don't require the binary search algorithm to do. It would be interesting to see how well the binary search algorithm would do if we don't give an upper-bound R, but let it estimate the effective resistance. This is a question for future work.

Now, we should note that even for 330 variables, which is the smallest where we observe a quantum speed-up, we reach a somewhat large complexity: at least $1.734 \cdot 10^{10}$ queries. If we translate this one-to-one to the number of operations, this is the point where a classical computer will start to struggle. Indeed, our computer didn't manage to solve any SAT instances in 330 variables after multiple hours of trying. Whether the observed quantum speed-ups are of significance, then, depends on how large a quantum computer one has: it has to be of a significant size to be able to handle a time complexity corresponding one-to-one to the query complexity.

But of course, the time complexity will exceed the query complexity. We haven't accounted for the large overhead that quantum operation carry, mostly due to error correcting. This could require many orders of magnitude of constant overhead per quantum operation, c.f. [BMT+22]. Whether a quantum speed-up for SAT will manifest in practice, is then not clear. But recall that we wanted to see whether a speed-up would even occur in the query complexity model: if we didn't find a speed-up here, then certainly we won't find a speed-up in practice. Thus, if one believes that the constant overhead for error correcting will come down significantly, then our results show that a speed-up over the DPLL algorithm for 3SAT using either Belovs' or Piddock's algorithms is possible.

Chapter 6

Conclusion

In this work we have provided efficiently classically computable exact expressions for the query complexity of Belovs' quantum walks detection algorithm, the query complexity of a binary search procedure for trees using Belovs' detection algorithm, and the expected query complexity of Piddock's quantum walk search algorithm. These expression allow the computation of the query complexity of these algorithms on problem instances of interest.

We provided an optimal configuration of Belovs' detection algorithm and experimentally show that this saves roughly a factor of 10 in query complexity compared to to either Belovs' or Montanaro's proposed configurations. We also provide a small optimisation and additions Piddock's algorithm, and upper-bounds on the inaccuracy of two subroutines of Piddock's algorithm.

We used our complexity expressions to compute the (expected) query complexity of the search algorithm on 3SAT, comparing to the classical DPLL algorithm. For the detection-based binary search algorithm, we observe a quantum speed-up of order between 1.36 to 1.56 (depending on the algorithm's configuration), which manifests in SAT instances in roughly 330 to 545 variables and on. For Piddock's search algorithm, we find a polynomial speed-up of order at most 1.58, which starts to occur in SAT instances in least 430 variables, where we note that Piddock's algorithm requires an upper-bound on the size of DPLL's backtracking tree, and computing this should worsen its complexity. Surprisingly, then, Piddock's algorithm doesn't perform obviously better than the binary search algorithm, despite a clear asymptotic advantage. Nonetheless, Piddock's algorithm works on arbitrary graphs, and the binary search algorithm only on trees.

But of course, the time complexity will exceed the query complexity. We haven't accounted for the large overhead that quantum operation carry, mostly due to error correcting. This could require many orders of magnitude of constant overhead per quantum operation, c.f. [BMT+22]. Whether a quantum speed-up for SAT will manifest in practice, is then not clear. But recall that we wanted to see whether a speed-up would even occur in the query complexity model: if we didn't find a speed-up here, then certainly we won't find a speed-up in practice. Thus, if one believes that the constant overhead for error correcting will come down significantly, then our results show that a speed-up over the DPLL algorithm for 3SAT using either Belovs' or Piddock's algorithms is possible.

We left many stones unturned, and list a few directions for future research below.

6.1 Future research

6.1.1 More experiments using the existing theory

We have provided a method to compute the query complexity of Belovs' detection algorithm, Piddock's search algorithm, and the binary search algorithm using Belovs' detection algorithm on any given problem instance of any search problem on graphs. We have only considered uniformly at random generated instances of 3SAT.

This is clearly very limited, and applying our method to other 3SAT instances and other search problems has the potential to tell us much more about possible quantum speed-ups. Remember that all we have established in this work is that a quantum speed-up does materialise for uniformly at random generated 3SAT instances: whether this speed-up is retained for different instances of this problem, for similar problems, let alone for very different problems, is something that can be investigated using the theory we have given.

In addition, it would be interesting to test Piddock's algorithm with a different s on the same data. Recall that this sets the precision of amplitude estimation, in turn determining how the estimates of η/R are rounded, which could in turn change the execution of Piddock's algorithm. Note that a larger s doesn't mean a better or more precise outcome: the outcome will be a marked vertex regardless. What it changes is the interim values the algorithm finds (η and b), and how long it takes to determine these, which are both large factors in the complexity. It may be possible to find some kind of optimum for s, perhaps depending on the structure of the input graph. Experiments might help to find such an optimum.

6.1.2 Optimise Piddock's algorithm

Throughout Chapter 4 we noted some possible optimisations for Piddock's quantum walks search algorithm. These are of large interest to us: we saw that our optimisation of Belovs' algorithm saved roughly a factor of 10 in the query complexity. If something similar can be achieved for Belovs' algorithm, it may suddenly perform quite a bit better than the binary search algorithm we considered.

First, we aim to find η such that $\eta/R_{s',M'} = 1/2$. Neither we nor Piddock argue that this choice is optimal. Recall that the larger we make η, the more amplitude is at the starting edges, and therefore the larger the success probability of phase estimation. Recall that we run phase estimation to construct $|\varphi\rangle$, and measuring this state giving us a shot at finding a marked vertex. If phase estimation fails half the time, that means half the time we won't even have a short at finding a marked vertex. Increase η then directly drops the expected number of repetitions to find a marked vertex. But conversely, increasing η also means the probability of observe a marked vertex drops, as move amplitude is at the starting edges, and therefore not at the marked edges. It seems there must be some optimal balance here.

Next, we currently seek b such that $R_{s',M'}(b) = 2R_{s',M'}(a)$. We showed in Section 4.5.1 that the larger the b we end up finding (i.e. the larger the ratio between $R_{s',M'}(b)$ and $R_{s',M'}(a)$), the higher the probability to find a marked vertex when measuring $|\varphi\rangle$. However, at the same time, the larger b, the larger the cost of preparing $|\varphi\rangle$. Neither we nor Piddock investigate whether the ratio of 2 is the optimal trade-off between these two.

Finally, for Algorithm 5, we could investigate whether the starting value of $\eta = 1/W$ and the doubling of η at each iteration are optimal. It might be better to start with a larger η or increase η faster, so that we get to the final η quicker. Conversely, it might be better to increase η slower,

to obtain a more accurate final result. Deriving an expression for the expected inaccuracy of η given a certain starting value and amount of increase would help to answer this. This would require more in depth study of how η changes the effective resistance.

6.1.3 Other quantum walk frameworks

Concurrently with Piddock's work on a general quantum walk search algorithm, a group of authors proposed a different quantum walk search algorithm for general graphs [AGJ21]. It's query complexity is $O(\sqrt{\log(RW)RW\log\log(RW)})$. This algorithm works quite differently from Piddocks, relying on quantum fast-forwarding instead of quantum phase estimation. It would be interesting to analyse this algorithm and see whether it gives a speed-up compared to Piddock in practice.

Similarly, as we noted in Chapter 3, Jarret and Wann have also given an algorithm that uses the electrical flow state $|\varphi\rangle$ to find marked elements, but this algorithm works only in trees [JW18]. Despite its asymptotic complexity seeming somewhat worse than Piddock's, it would be worth giving exact expressions for their complexity, and seeing how Jarret and Wann's algorithm compares to Piddock's in practice.

6.1.4 Actual tree size

The quantum algorithms all used an upper-bound T on the full size of classical backtracking tree. For some satisfiable instances, this can be much larger than the size of the classical tree upon finding the first solution. It was shown in [AK17] that this initial subtree can be determined, after which a quantum walk can be done on just this initial tree, making the complexity dependend on this potentially far smaller subtree. It would be interesting to give an exact expression for the query cost of this altered algorithm, to see whether it gives a speed-up in practice.

6.1.5 Estimate effective resistance for binary search algorithms

The polynomial speed-up of between 1.36 to 1.56 that we found for the binary search algorithm depending on how tightly the effective resistance upper-bound was. Now, recall that Piddock's included Algorithm 5 that can estimate the effective resistance of a graph. If we use this, we can set the upper-bound tightly, yielding (close to) the speed-up of 1.56. It would be interesting determine the final speed-up of the binary search algorithm after incorporating the cost of running Algorithm 5.

6.1.6 Further optimise Belovs' algorithm

In Chapter 3 we noted a few possible ways to further optimise Belovs' algorithm.

The first has to do with the application of the spectral gap lemma. This lemma upper-bounds the overlap between the vector $|\varphi\rangle$ and the eigenvectors with phase less than Θ of the quantum walk unitary. The goal here was to upper-bound the overlap between $|\varphi\rangle$ and the eigenvector(s) with phase 0. By giving this slightly more general statement, which includes other small phases that are not 0, we most likely paint a pessimistic picture of the success probability. Exactly expressing the overlap between $|\varphi\rangle$ and 0 might therefore reveal a slightly larger success probability using the same number of precision bits (i.e. the same complexity).

Another problematic component of the use of the effective spectral gap lemma is that if we set our precision larger than Θ, we have no idea what happens to the success probability, and

have to assume that the algorithm simply fails. That is, if we supply the algorithm with upper-bounds R and W that are too low, we have to assume the algorithm fails, even if in practice it may not. We ran in to this in our procedure to estimate W. We assumed that the algorithm would fail continuously up until we increased W enough to really become an upper-bound on the sum of weights of our graph. This implied a logarithmic number of repetitions in W. If we can express the success probability in terms of the distance between the given bound W and the actual sum of weights, we might be able to turn this into a constant overhead. Something similar was shown for estimating a parameter of Grover search in [CFNW22a].

Finally, our optimised values for the constants C and n were relative to an upper-bound on the query complexity of Belovs' algorithm. We had to do this, as there was no global optimum relative to the actual expression for the complexity. This was because the ceiling function in the expression for the complexity. Understanding the behaviour of this real expression for the complexity may allow us to give even better values for C and n, though these would then depend on R and W, so it is unclear whether this would be feasible.

6.1.7 More efficient estimation of the complexity of Piddock's algorithm

Recall that we managed to improve the efficiency of the classical algorithm to estimate the query complexity of Piddock's algorithm by recalling that for trees, the number of iterations of Algorithm 5 was very easy too determine. We attempted something similar for Algorithm 6, which forms by far the most costly part of the algorithm to compute the query complexity of Piddock's algorithm. Indeed, due to this, we were not able to consider SAT instances beyond 125 variables.

Unfortunately, we were not able to solve this, as it requires a deeper understanding of the effective resistance from root to marked leaves in trees. We managed to show the following result.

Proposition 6.1.1. Let G be a weighed tree with root r and edge weights equal to 1. Let $M \subseteq G$ be marked subset of vertices. If the unique paths from r to each $m \in M$ don't overlap, then the electric flow from s to M is the flow that assigns each path an equal sum of flow.

In a real backtracking tree, the paths will overlap however, making the situation more complex. Investigating how this changes the electrical flow may thus allow us to hugely speed-up the computation of the query complexity of Piddock's algorithm.

Bibliography

[AAKV01] D. Aharonov, A. Ambainis, J. Kempe, and U. Vazirani, Quantum walks on graphs, in *Proceedings of the Thirty-Third Annual ACM Symposium on Theory of Computing - STOC '01*, ACM Press, Hersonissos, Greece, 2001, pp. 50–59. https://doi.org/10.1145/380752.380758.

[ADZ93] Y. Aharonov, L. Davidovich, and N. Zagury, Quantum random walks, *Physical Review A* **48** no. 2 (1993), 1687–1690. https://doi.org/10.1103/PhysRevA.48.1687.

[AL18] T. Albash and D. A. Lidar, Adiabatic quantum computation, *Reviews of Modern Physics* **90** no. 1 (2018), 015002. https://doi.org/10.1103/RevModPhys.90.015002.

[AK17] A. Ambainis and M. Kokainis, Quantum algorithm for tree size estimation, with applications to backtracking and 2-Player games, in *Proceedings of the 49th Annual ACM SIGACT Symposium on Theory of Computing, STOC 2017*, Association for Computing Machinery, New York, NY, USA, 2017, pp. 989–1002. https://doi.org/10.1145/3055399.3055444.

[AC22] C.-C. Andrici and S. Ciobaca, A Verified Implementation of the DPLL Algorithm in Dafny, *Mathematics* **10** no. 13 (2022), 2264. https://doi.org/10.3390/math10132264.

[AGJ21] S. Apers, A. Gilyén, and S. Jeffery, A unified framework of quantum walk search, in *38th International Symposium on Theoretical Aspects of Computer Science (STACS 2021)* (M. Bläser and B. Monmege, eds.), *Leibniz International Proceedings in Informatics (LIPIcs)* **187**, Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 2021, pp. 6:1–6:13. https://doi.org/10.4230/LIPIcs.STACS.2021.6.

[BBC+01] R. Beals, H. Buhrman, R. Cleve, M. Mosca, and R. De Wolf, Quantum lower bounds by polynomials, *Journal of the ACM* **48** no. 4 (2001), 778–797. https://doi.org/10.1145/502090.502097.

[Bel13] A. Belovs, Quantum Walks and Electric Networks, (2013). https://doi.org/10.48550/ARXIV.1302.3143.

[BMT+22] M. E. Beverland, P. Murali, M. Troyer, K. M. Svore, T. Hoefler, V. Kliuchnikov, G. H. Low, M. Soeken, A. Sundaram, and A. Vaschillo, Assessing requirements to scale to practical quantum advantage, November 2022.

[BHMT02] G. Brassard, P. Hoyer, M. Mosca, and A. Tapp, Quantum Amplitude Amplification and Estimation, **305**, 2002, pp. 53–74. https://doi.org/10.1090/conm/305/05215.

[BBD+09] H. J. Briegel, D. E. Browne, W. Dür, R. Raussendorf, and M. Van den Nest, Measurement-based quantum computation, *Nature Physics* **5** no. 1 (2009), 19–26. https://doi.org/10.1038/nphys1157.

[CFNW22a] C. Cade, M. Folkertsma, I. Niesen, and J. Weggemans, Quantifying Grover speed-ups beyond asymptotic analysis, March 2022. https://doi.org/10.48550/arXiv.2203.04975.

[CFNW22b] C. Cade, M. Folkertsma, I. Niesen, and J. Weggemans, Quantum Algorithms for Community Detection and their Empirical Run-times, March 2022. https://doi.org/10.48550/arXiv.2203.06208.

[CEMM98] R. Cleve, A. Ekert, C. Macchiavello, and M. Mosca, Quantum algorithms revisited, *Proceedings of the Royal Society of London. Series A: Mathematical, Physical and Engineering Sciences* **454** no. 1969 (1998), 339–354. https://doi.org/10.1098/rspa.1998.0164.

[DLL62] M. Davis, G. Logemann, and D. Loveland, A machine program for theorem-proving, *Communications of the ACM* **5** no. 7 (1962), 394–397. https://doi.org/10.1145/368273.368557.

[DP60] M. Davis and H. Putnam, A Computing Procedure for Quantification Theory, *Journal of the ACM* **7** no. 3 (1960), 201–215. https://doi.org/10.1145/321033.321034.

[Deu85] D. Deutsch, Quantum theory, the Church–Turing principle and the universal quantum computer, *Proceedings of the Royal Society of London. A. Mathematical and Physical Sciences* **400** no. 1818 (1985), 97–117. https://doi.org/10.1098/rspa.1985.0070.

[Gro96] L. K. Grover, A fast quantum mechanical algorithm for database search, in *Proceedings of the Twenty-Eighth Annual ACM Symposium on Theory of Computing - STOC '96*, ACM Press, Philadelphia, Pennsylvania, United States, 1996, pp. 212–219. https://doi.org/10.1145/237814.237866.

[JW18] M. Jarret and K. Wan, Improved quantum backtracking algorithms using effective resistance estimates, *Physical Review A* **97** no. 2 (2018), 022337. https://doi.org/10.1103/PhysRevA.97.022337.

[KSV02] A. Kitaev, A. Shen, and M. Vyalyi, *Classical and Quantum Computation*, *Graduate Studies in Mathematics* **47**, American Mathematical Society, Providence, Rhode Island, May 2002. https://doi.org/10.1090/gsm/047.

[LMR+11] T. Lee, R. Mittal, B. W. Reichardt, R. Spalek, and M. Szegedy, Quantum Query Complexity of State Conversion, in *2011 IEEE 52nd Annual Symposium on Foundations of Computer Science*, IEEE, Palm Springs, CA, USA, October 2011, pp. 344–353. https://doi.org/10.1109/FOCS.2011.75.

[LPW+17] D. A. Levin, Y. Peres, E. L. Wilmer, J. Propp, and D. B. Wilson, *Markov Chains and Mixing Times*, second edition ed., American Mathematical Society, Providence, Rhode Island, 2017.

[Lov93] L. Lovász, Random Walks on Graphs: A Survey, *Combinatorics, Paul Erdos is Eighty* **2** (1993), 46.

[MNRS07] F. Magniez, A. Nayak, J. Roland, and M. Santha, Search via quantum walk, in *Proceedings of the Thirty-Ninth Annual ACM Symposium on Theory of Computing*, ACM, San Diego California USA, June 2007, pp. 575–584. https://doi.org/10.1145/1250790.1250874.

[Mon16] A. Montanaro, Quantum walk speedup of backtracking algorithms, January 2016.

[Mon18] A. Montanaro, Quantum walk speedup of backtracking algorithms, *Theory of Computing* **14** no. 1 (2018), 1–24. https://doi.org/10.4086/toc.2018.v014a015.

[NC10] M. A. Nielsen and I. L. Chuang, *Quantum Computation and Quantum Information*, 10th anniversary ed ed., Cambridge University Press, Cambridge ; New York, 2010.

[Pid19] S. Piddock, Quantum walk search algorithms and effective resistance, December 2019.

[San08] M. Santha, Quantum walk based search algorithms, August 2008.

[SKW03] N. Shenvi, J. Kempe, and K. B. Whaley, Quantum random-walk search algorithm, *Physical Review A* **67** no. 5 (2003), 052307. https://doi.org/10.1103/PhysRevA.67.052307.

[Sze04] M. Szegedy, Quantum Speed-Up of Markov Chain Based Algorithms, in *45th Annual IEEE Symposium on Foundations of Computer Science*, IEEE, Rome, Italy, 2004, pp. 32–41. https://doi.org/10.1109/FOCS.2004.53.

[Wil16] M. M. Wilde, *From Classical to Quantum Shannon Theory*, November 2016. https://doi.org/10.1017/9781316809976.001.

[dW22] R. de Wolf, Quantum Computing: Lecture Notes, August 2022.

[Won17] T. G. Wong, Equivalence of Szegedy's and coined quantum walks, *Quantum Information Processing* **16** no. 9 (2017), 215. https://doi.org/10.1007/s11128-017-1667-y.

Appendix A

A.1 Amplifying success probability

Suppose we have a probabilistic algorithm X that outputs 1 with probability $p > 1/2$ and 0 with probability $1 - p$ and wish to amplify the success probability to some $c > (0.5, 1)$. We can do this by performing multiple, say n, independent runs of the algorithm and outputting a majority vote. Intuitively, since 1 is more likely, this should decrease the probability that we output 0. It turns out that this drop is exponential in the number of repetitions.

Let a run of the algorithm be described by a random variable X_i. When doing n runs, we consider the random variable $X = X_1 + X_2 + ... + X_n$, whose expected outcome is $\mathbb{E}(X) \geq pn$, and we are interested in studying the probability to diverge from this expected outcome, which we claimed above drops exponentially with n.

Note that after n repetitions, there are 2^n possible outcomes, which we might express as the 2^n subsets $S \subseteq [n] = \{1, ..., n\}$, where $i \in S$ means run i of the algorithm returned 1. In the positive case, we want to end up with a majority of runs returning 1, and therefore a set S such that $|S| > n/2$. The majority vote procedure fails when we obtain a set S with $|S| \leq n/2$. For each such S, the probability of obtaining it is equal to the probability of obtaining $|S|$ successful runs (each occurring with probability p) and $n - |S|$ unsuccessful runs (each occurring with probability $1 - p$), i.e. $p^{|S|}(1 - p)^{n-|S|}$.

We can thus express the probability to observe any such outcome. If we now sum the probabilities to observe any set that corresponds to an incorrect output, we can express the probability that our majority vote fails. We get:

$$\delta \leq \sum_{S \subseteq [n] \wedge |S| \leq k/2} p^{|S|}(1-p)^{k-|S|} = ((1-p)p)^{n/2} \sum_{S \subseteq [n] \wedge |S| \leq k/2} (\frac{1-p}{p})^{n/2-|S|} < ((1-p)p)^{n/2} 2^n \cdot 1 = (2\sqrt{(1-p)p}))^n$$

where we note that the first equality follows from [KSV02, Equation (4.1)]. We have $p(1-p) < 1/4$, as this expression is maximised to $1/4$ when $p = 1/2$. It follows that $\sqrt{p(1-p)} < 0.5$ and finally $2\sqrt{(1-p)p)} < 1$, so that raising n makes the error probability δ arbitrarily small.

How large should n become to achieve some desired error rate δ'? Using the upper-bound, we want to find n such that $(2\sqrt{(1-p)p)})^n = \delta'$, so that

$$n \leq \log_{2\sqrt{(1-p)p)}} \delta = \frac{\log \delta'}{\log(2\sqrt{(1-p)p)}} = \frac{-\log \frac{1}{\delta'}}{\log(2\sqrt{(1-p)p)}} = \frac{1}{-\log(2\sqrt{(1-p)p)}} \log \frac{1}{\delta'} \in O(\log \frac{1}{\delta'}).$$

Recall from above that $2\sqrt{(1-p)p} < 1$ so that the log of this expression is negative, hence $-\log$ of this expression is positive. In other words, to achieve some error probability δ', it suffices to increase n logarithmically in $1/\delta'$, that is, by increasing n, δ' drops exponentially.

What if we want to determine n exactly? For a given n, we can work out the error probability δ exactly to

$$\begin{aligned}
\delta &\leq \sum_{S\subseteq[n]\wedge|S|\leq k/2} p^{|S|}(1-p)^{k-|S|} \\
&= \sum_{i=0}^{n/2} \binom{n}{i} p^i (1-p)^{n-i} && \text{Since there are } \binom{n}{i} \text{ subsets of size } i. \\
&= (1-p)^n \sum_{i=0}^{n/2} \binom{n}{i} p^i (1-p)^{-i} \\
&= (1-p)^n \sum_{i=0}^{n/2} \binom{n}{i} \left(\frac{p}{1-p}\right)^i,
\end{aligned}$$

so that in our application, assuming we have p and the desired error probability δ', we might be able to solve the above equation for n. When this fails analytically, it can be done easily computationally: simply try increasing n, until the expression for δ drops below the desired δ'. By the above, we know this should happen in linear time, even if δ' is exponentially small.

Proposition A.1.1. Doing n repetitions of a probabilistic decision algorithm that outputs correctly with probability $p > 1/2$ and outputting a majority vote results in an error probability of at most

$$\delta \leq (1-p)^n \sum_{i=0}^{n/2} \binom{n}{i} \left(\frac{p}{1-p}\right)^i \leq \frac{1}{-\log(2\sqrt{(1-p)p})} \log(\frac{1}{p}) \in O(\log(\frac{1}{p})).$$

It follows that to achieve some error probability δ' it suffices to do $n \in O(\log(1/delta'))$ repetitions, specifically, n that is a solution to the inequality

$$\delta' \leq (1-p)^n \sum_{i=0}^{n/2} \binom{n}{i} \left(\frac{p}{1-p}\right)^i.$$

Reducing the error probability exponentially thus requires linearly many repetitions. The above works only for a one-sided error. For a two-sided error (even where one error rate may be $\geq 1/2$), we wouldn't quite do a majority vote. Instead, when, say, $p_+ \geq a$ and $p_- \geq b$, we expect $\mathbb{E}(X) \geq an$ in the positive case, and $\mathbb{E}(X) \leq (1-b)n$ in the negative case (where we interpret $X_i = 1$ as a positive output). To distinguish these two cases, then, we can set the threshold at their average: we accept the input whenever $X > n(a + (1-b))/2$. Of course, this only works when there is a gap between $p_+ = a$ and $1 - p_- = 1 - b$, i.e. when $1 - b < a$. For example, if $p_+ \geq 1/2$ and $p_- \geq 3/4$, we get $\mathbb{E}(X) \geq 1/2n$ in the positive case and $\mathbb{E}(X) \leq 1/4n$ in the negative case, so that the threshold becomes $X > 3/8n$.

It is a little trickier to express the error probability exactly in this scenario, i.e. to list all outcomes of the experiment where we don't hit the success threshold. In particular, to do so, we would have to do it separately for the positive and negative case. We can however upper-bound this easily by appealing to Chernoff's bound. This bound says that the probability to diverging from the expected value of the sum of n independent samples from a random variable decreases exponentially with n [dW22]:

$$P(|X - \mu| > \alpha\mu) \leq 2e^{-\frac{\alpha^2\mu}{2+\alpha}}.$$

A.1.1 Making sure r consecutive runs of an algorithm are correct

Lemma A.1.2. Given an algorithm that succeeds with probability at least $> 1/2$. Guaranteeing that r consecutive runs of the algorithm are successful with probability $1-\delta$ with $\delta \leq 1-1/e \approx 0.6321$ requires reducing the error probability of the base algorithm to $p \in O(1/r)$.

Proof. Say we amplify the success probability to p. The probability that all r runs are correct is then p^r so that we want to solve $p^r \geq 1-\delta$ for p, giving $p \geq \sqrt[r]{1-\delta}$. To understand this slightly better, note that we can write out the lowered error probability of our base algorithm as $1-p \leq 1-\sqrt[r]{1-\delta}$. It is not hard to see that $1-\sqrt[r]{1-\delta} \leq 1/r$ for $r \geq 2$ and $0 \leq \delta \leq 1-((r-1)/r)^r \leq 1-1/e \approx 0.6321$, where the last inequality follows because $\lim_{r\to\infty}((r-1)/r)^r = 1/e$.

It follows that the new error probability of the base algorithm is $1-p \leq 1/r \in O(1/r)$, assuming that the required error probability satisfies $\delta \leq 1-1/e \approx 0.6321$. □

A.2 Upper-bound on the inaccuracy of Piddock's algorithms

A.2.1 Algorithm 5

Let us use $R_{s',M'}(c)$ to refer to the effective resistance of the graph where we set $\eta = c$. Recall from Proposition A.3.1 that $R_{s',M'}(c) \in [R_{\sigma,M'} + c/\sqrt{d}, R_{\sigma,M'} + c]$. What instance would lead to the worst-case outcome for out algorithm, i.e. an η that is as large as possible? We would want the largest possible η that yields an estimate $\eta/R_{s',M'}(\eta) \approx \tilde{a}$ that just fails to hit the stopping condition, i.e. $\tilde{a} = 1/2 - \epsilon_a$, so that after doubling η, we hit the stopping condition with the largest possible 2η. Looking at the range from which we draw $\tilde{a} = 1/2 - \epsilon_a$, we see that the largest η that can yield $\tilde{a} = 1/2 - \epsilon_a$ is η such that $\eta/R_{s',M'}(\eta) = 1/2$.

So, consider doubling this η. We then hit the stopping condition with $2\eta/R_{s',M'}(2\eta)$. This expression is maximised when the numerator, i.e. the effective resistance, is minimised, which gives

$$\frac{2\eta}{R_{s',M'}(2\eta)} = \frac{2\eta}{R_{\sigma,M} + 2\eta/\sqrt{d}},$$

so that the distance of this maximal outcome to the goal of $1/2$ becomes

$$\epsilon_\eta(d) = \frac{2\eta}{R_{\sigma,M} + 2\eta/\sqrt{d}} - \frac{1}{2} = \frac{4\sqrt{d}\eta - \sqrt{d}R_{\sigma,M} - \eta}{2(\sqrt{d}R_{\sigma,M} + \eta)}.$$

Since we are most interested in the case $d = 1$, let us consider

$$\epsilon_\eta(1) = \frac{4\eta - R_{\sigma,M} - \eta}{2(R_{\sigma,M} + \eta)} = \frac{3}{2} - \frac{2R_{\sigma,M}}{\eta + R_{\sigma,M}}.$$

To relate these values, we note that we can write

$$\frac{\eta}{R_{s',M'}(\eta)} = \frac{1}{2} = \frac{\eta}{R_{\sigma,M} + \eta/a} \iff R_{\sigma,M} = 2\eta - \eta/a,$$

where $a \in [1, \sqrt{d}]$, depending on how uniformly the flow is spread over the d starting edges. Of course, we only have one starting edge, so that $a = 1$. This tells us that $R_{\sigma,M} = \eta$ which implies $\epsilon_\eta(1) = 3/2 - \frac{2\eta}{2\eta} = 1/2$. This is quite terrible: since we try to estimate $1/2$ with values in $[0,1]$, an error range of $1/2$ means we can't assume anything: we might as well return a random value.

One way around this is to note that this worst-case is quite unrealistic. In this scenario, we first check a very good η that was just outside our success range, jump to the very bad 2η that *is* in the success range, and terminate. If we would just remember the previous estimate $\tilde{a}$, we would see that it is much closer to $1/2$, so that returning η would be much better than returning 2η. This would turn our worst-case into a very good case, at no extra query cost. Consider now this altered algorithm: we remember the previous $\tilde{a}$, and upon reaching the accepting condition, check if the current estimate or the previous estimate is closer to $1/2$, and return the corresponding η. What would be the worst-case of this altered algorithm be?

We would want an η so that the distance from $\eta/R_{s',M'}$ and x to $1/2$ is equal, as then relying on the previous η is of no benefit. This would cut the worst-case error by at least half: before, the entire error range consisted in moving from $1/2$ to a larger value, now we move equally much between $\eta/R_{s',M'}$ and $1/2$ and between $1/2$ and x. Formally

$$\frac{2\eta}{R_{s',M'}(2\eta)} - \frac{1}{2} = \frac{1}{2} - \frac{\eta}{R_{s',M'}(\eta)} \iff \frac{2\eta}{R_{s',M'}(2\eta)} + \frac{\eta}{R_{s',M'}(\eta)} = 1.$$

To maximise the difference, we want to minimise $\frac{\eta}{R_{s',M'}(\eta)}$ and maximise $\frac{2\eta}{R_{s',M'}(2\eta)}$, meaning we want to maximise $R_{s',M'}(\eta) = R_{\sigma,M} + \eta$ and minimise $R_{s',M'}(2\eta) = R_{\sigma,M} + 2\eta/\sqrt{d}$, yielding the new condition

$$\frac{2\eta}{R_{\sigma,M} + 2\eta/\sqrt{d}} + \frac{\eta}{R_{\sigma,M} + \eta} = 1.$$

Solving for η with $d = 1$ gives us $\eta = R/\sqrt{2}$. The difference with $1/2$ becomes

$$\epsilon_\eta(1) = \frac{1}{2} - \frac{\eta}{R_{s',M'}(\eta)} = \frac{1}{2} - \frac{R/\sqrt{2}}{R_{\sigma,M} + R/\sqrt{2}} = \frac{1}{2} - \frac{1}{\sqrt{2}+1} \leq 0.0858 \approx 1/12.$$

This is quite a bit better than what we had before.

A.2.2 Choosing s and ϵ within this upper-bound

Note that this upper-bound $\epsilon_\eta(d)$ is independent of ϵ_a and ϵ (i.e of the precision with which we do phase and amplitude estimation), as we've attempted to directly identify the worst-case in terms of $\eta/R_{s',M'}$ (and not its estimation $\tilde{a}$). We should reflect briefly on whether these two error ranges might be able to result in an even worse case. This would have to occur either on the left-side ($< 1/2$) or on the right-side ($> 1/2$).

On the left-side, we would need to find η with $\eta/R_{s',M'}$ that is more than $\epsilon_\eta(d)$ to the left of $1/2$, but still terminating, i.e. still yielding $\tilde{a} > 1/2 - \epsilon_a$. Recall from the previous section that if $\tilde{a} \in [1/2 - \epsilon_a, 1/2 + \epsilon_a + \epsilon]$, we know that the underlying $\eta/R_{s',M'}$ can be any of $[1/2 - 2\epsilon_a - \epsilon, 1/2 + 2\epsilon_a + \epsilon]$. In other words, an $\tilde{a}$ that terminates the algorithm can have an underlying η with $\eta/R_{s',M'}$ as small as $1/2 - 2\epsilon_a - \epsilon$. Thus, if $2\epsilon_a + \epsilon > \epsilon_\eta(d)$, this smallest η becomes the new worst-case: it is strictly further from $1/2$ than $\epsilon_\eta(d)$.

On the right-side, any η will of course hit the stopping condition, as $1/2 < \eta/R_{s',M'}$ and $\tilde{a}$ can at most be ϵ_a below $\eta/R_{s',M'}$, so that even the smallest $\tilde{a}$ still satisfies $\tilde{a} > \eta/R_{s',M'} - \epsilon_a > 1/2 - \epsilon_a$. Having $\eta/R_{s',M'}$ overshoot $1/2$ by more than $\epsilon_\eta(d)$ is then only possible by doubling some η that hasn't hit the stopping condition, and jumping all the way to some η with $1/2 + \epsilon_\eta(d) < \eta/R_{s',M'}$. But of course, we showed above that this is impossible: we will only ever go $\epsilon_\eta(d)$ beyond $1/2$ (or more precisely, we might go further beyond $1/2$, but only when

the previous $\eta/R_{s',M'}$ was closer to 1/2 than $\epsilon_\eta(d)$, so that the algorithm will already terminate with smaller, previous value).

Thus, as long as $2\epsilon_a + \epsilon \leq \epsilon_\eta(d)$, we can say that for the η we output, $\eta/R_{s',M'}$ is at most $\epsilon_\eta(d)$ from 1/2. Note that $\epsilon_\eta(d)$ decreases with d, and thus peaks at $3/2 - \sqrt{2} \approx 0.0858$ for $d = 1$. To prove our upper-bound, is therefore suffices to have $2\epsilon_a + \epsilon \leq 3/2 - \sqrt{2}$. What is the optimal combination of s and ϵ that attains this?

Recall that ϵ_a is determined by the number of precision bits s used in amplitude estimation:

$$\epsilon_a = \frac{2\pi\sqrt{a(1-a)}}{2^s} + (\frac{\pi}{2^s})^2 \leq \frac{\pi}{2^s} + (\frac{\pi}{2^s})^2$$

where we use $\sqrt{a(1-a)} \leq 1/2$ for $a \in (0,1)$. Below, we list the size of this error range for increasing s. Recall that adding a bit to s means doubling the number of queries that amplitude estimation does.

Recall that ϵ is chosen directly by us, and that increasing it by a certain factor increases the number of queries by the inverse factor. Thus, if we set $\epsilon = 1/a$, the number of queries increases by a factor 10 (ignoring here for a moment the effect of the ceiling function, as we ceil the precision for phase estimation to determine the number of bits).

s	Upper bound on ϵ_a		
5	$\pi(32+\pi)/1024$	≤ 0.1079	
6	$\pi(64+\pi)/4096$	≤ 0.05150	$< 1/10$
7	$\pi(128+\pi)/16384$	≤ 0.02515	
8	$\pi(256+\pi)/65536$	≤ 0.01243	
9	$\pi(512+\pi)/262144$	≤ 0.006174	$< 1/100$
10	$\pi(1024+\pi)/1048576$	≤ 0.003078	
11	$\pi(2048+\pi)/4194304$	≤ 0.001537	
12	$\pi(4096+\pi)/16777216$	≤ 0.0007676	$< 1/1000$
13	$\pi(8196+\pi)/67108864$	≤ 0.0003837	
14	$\pi(16384+\pi)/268435456$	≤ 0.000191784	
15	$\pi(32768+\pi)/1073741824$	≤ 0.000095883	$< 1/10000$

Table A.1: The maximum inaccurcay ϵ_a of amplitude estimation for increasing number s of precision bits s.

What is the cheapest configuration of s and ϵ that yields $2\epsilon_a + \epsilon \leq \epsilon_\eta(1) = 3/2 - \sqrt{2}$? We note that from $s = 7$ on, $\epsilon_a < 2\epsilon_\eta(1)$, allowing the error to be small enough. How large can we pick ϵ to still satisfy our constraint? We have $\epsilon_a = \frac{\pi(128+\pi)}{16384} \approx 0.02515$ so that

$$2\epsilon_a + \epsilon = 2\frac{\pi(128+\pi)}{16384} + \epsilon \leq \epsilon_\eta(1) = 3/2 - \sqrt{2}$$

which implies we can choose $\epsilon = 3/2 - \sqrt{2} - 2\frac{\pi(128+\pi)}{16384} \approx 0.03549$ implying a slowdown of $1/\epsilon \approx 28.17$. If we instead choose $s = 8$ we have $\epsilon_a = \frac{\pi(256+\pi)}{65536} \approx 0.01243$ so that

$$2\epsilon_a + \epsilon = 2\frac{\pi(256+\pi)}{65536} + \epsilon \leq \epsilon_\eta(1) = 3/2 - \sqrt{2}$$

which implies we can choose $\epsilon = 3/2 - \sqrt{2} - 2\frac{\pi(256+\pi)}{65536} \approx 0.06093$ implying a slowdown of $1/\epsilon \approx 16.41$.

The slowdown due to incrementing s is doubling from $2^7 - 1 = 127$ to $2^8 - 1 = 255$. Thus moving to $s = 8$ increases the number of repetitions due to s by 128 and decreases the number of repetitions due to ϵ by ≈ 11.76. It is clear that increasing s further will make this even worse, so that staying at $s = 7$ and $\epsilon = 3/2 - \sqrt{2} - 2\frac{\pi(128+\pi)}{16384} \approx 0.03549$ is optimal.

A.2.3 Upper-bound on the inaccuracy of Algorithm 6

Recall that $R_{s',M'}(c)$ refers to the effective resistance of the graph where we set $x = c$. Recall from Proposition A.3.1 that $R_{s',M'}(c) \in [R_{s',M} + c/|M|, R_{\sigma,M'} + c]$. The worst-case would be the largest possible x that gives rise to a $\tilde{a}$ that just fails to hit the stopping condition, i.e. $\tilde{a} = \eta/(2R_{s',M'}(\eta)) + \epsilon_\eta(d)$, so that after doubling x, we hit the stopping condition with the largest possible x. Since we draw $\tilde{a}$ from $[\eta/R_{s',M'}(x) - \epsilon_\eta(d), \eta/R_{s',M'}(x) + \epsilon_\eta(d)]$, the largest x satisfying our condition would be when we draw from the right of this interval, so that $\eta/R_{s',M'}(x) = \eta/2R_{s',M'}(\eta)$. The error becomes

$$\epsilon_b = \frac{\eta}{2R_{s',M'}(\eta)} - \frac{\eta}{R_{s',M'}(2x)}.$$

To maximise this, we see that we should maximise the left term and minimise the right term, so that we should minimise the left term's denominator, and maximise the right term's denominator:

$$\epsilon_b = \frac{\eta}{2R_{s',M'}(\eta)} - \frac{\eta}{R_{s',M'}(2x)} = \frac{\eta}{R_{s',M} + x/|M|} - \frac{\eta}{R_{s',M} + 2x}.$$

We now need to solve x, and the larger x is, the larger the difference. We know $R_{s',M'}(x) = 2R_{s',M'}(\eta)$ so that $R_{s',M} + x/|M| = 2R_{s',M'}(\eta) = 2R_{s',M} + 2\eta/c$, for some $c \in [1, |M|]$. Rewriting gives $x = |M|(2(R_{s',M} + \eta/c) - R_{s',M}) = |M|(R_{s',M} + \eta/c)$. We see that choosing $c = 1$ maximises x. We also verify here our previous choice to minimise $R_{s',M'}(x)$ to $R_{s',M} + x/|M|$, as this now increases the value of x even further. Plugging in the solution for x, and recalling that $R_{s',M} \approx 2\eta$, gives

$$\epsilon_b = \frac{\eta}{R_{s',M} + x/|M|} - \frac{\eta}{R_{s',M} + 2x} = \frac{1}{5} - \frac{1}{6|M| + 2}.$$

The error then increases with M, tending to a maximum of $1/5$. For $M = 10$ the error becomes $1/5 - 1/(62) = 57/310 \approx 0.1839$. For $M = 1$ the error becomes $1/5 - 1/8 = 3/40 = 0.075$.

Let us now express the error between $R_{s',M'}(2x)$ and $2R_{s',M'}(\eta)$ directly. We have

$$\frac{\eta}{2R_{s',M'}(\eta)} - \frac{\eta}{R_{s',M'}(2x)} \leq \frac{1}{5} \iff R_{s',M'}(2x) - 2R_{s',M'}(\eta) \leq \frac{2R_{s',M'}(\eta)R_{s',M'}(2x)}{5\eta}.$$

We can expand these again

$$\frac{2R_{s',M'}(\eta)R_{s',M'}(2x)}{5\eta} = \frac{2(R_{s',M} + \eta)(R_{s',M} + 2x)}{5\eta} = \frac{12}{5}\eta(3|M| + 1),$$

where in the last step we used our solution for x and $R_{s',M} \approx 2\eta$. This is quite large. Say we have 10 marked elements, we then have $\epsilon_b \approx 74.4\eta$. This is very large: recall that $R_{s',M} \approx 2\eta$ and $R_{s',M'}(2x) \leq R_{s',M} + 2x$

A.3 Flow

Proposition A.3.1. Let M and M' be sets of vertices such that each $k \in M$ is connected to exactly one $k' \in M'$ via a single edge kk', and not in any other way. Let the weight of each kk' be equal to $1/x$ for some x. The energy of f is maximised to x in case the flow is fully concentrated at one edge, and the energy of f is minimised to $x/|M|$ the flow is uniformly spread over all edges.

Proof. Recall that the energy of a flow is given by

$$\sum_{e \in E} \frac{f(e)^2}{w(e)}.$$

When the flow is concentrated at a single edge e the total energy is $f(e)^2/w(e) = 1^2/w(e) = 1/w(e)$. If we would shift flow to some other edge, the energy would become $f(e)^2/w(e) + f(e')^2/w(e') = (f(e)^2 + f(e')^2)/w(e)$. But since $0 < f(e) < 1$ and $0 < f(f(e') < 1$, we know that $f(e)^2 < f(e)$ and $f(e')^2 < f(e)$. This then implies that $(f(e)^2 + f(e')^2)/w(e) < (f(e) + f(e')/w(e) = 1/w(e)$, where in the last step we used the fact that the flow sums to 1, and the flow is fully concentrated at these two edges. It thus follows that spreading flow over two edges gives a strictly smaller energy that keeping the flow concentrated at a single edge.

For the second claim, note that when the flow is uniformly spread over all edges, the energy contributed by each edge is

$$\frac{(1/|M|)^2}{1/x} = \left(\frac{x}{|M|}\right)^2.$$

Say we move c flow from one edge to another. The new energy at the edge losing c flow becomes

$$\left(\frac{1}{|M|} - c\right)^2 = x\left(\left(\frac{1}{|M|}\right)^2 - \frac{2}{|M|} + c^2\right),$$

so that is loses a total of

$$x\left(c^2 - \frac{2}{|M|}\right)$$

energy. Conversely, the new energy at the edge gaining c flow becomes

$$\left(\frac{1}{|M|} + c\right)^2 = x\left(\left(\frac{1}{|M|}\right)^2 + \frac{2}{|M|} + c^2\right),$$

so that is gains a total of

$$x\left(\frac{2}{|M|} + c^2\right).$$

The net gain in energy is then

$$x\left(c^2 - \frac{2}{|M|}\right) + x\left(\frac{2}{|M|} + c^2\right) = 2x \cdot c^2.$$

Thus, any an adjustment in flow strictly increases the total energy. □