



SIGACT News Complexity Theory Column 123

Ben Lee Volk
Efi Arazi School of Computer Science
Reichman University
Herzliya, Israel



Introduction to Complexity Theory Column 123

We close off 2024 with a fantastic guest column by Stacey Jeffery, on the surprising intricacies regarding composition of quantum algorithms. Stacey also did great work in providing illuminating intuitive explanations and figures, so trust me — you don't want to miss it!

What's in store for 2025? We expect to publish guest columns by Igor Carboni Oliveira (on Meta-Mathematics of Computational Complexity Theory), Noam Mazor (topic TBA), Elena Grigorescu (on Locally Decodable Codes for Insertions and Deletions) and Mitali Bafna (on High-dimensional expanders and their applications to PCPs and codes), so stay tuned!

Guest Column: Composing Quantum Algorithms¹

*Stacey Jeffery*²



Abstract

Composition is something we take for granted in classical algorithms design, and in particular, we take it as a basic axiom that composing “efficient” algorithms should result in an “efficient” algorithm – even using this intuition to justify our definition of “efficient.” Composing quantum algorithms is a much more subtle affair than composing classical algorithms. It has long been known that zero-error quantum algorithms *do not* compose, but it turns out that, using the right algorithmic lens, bounded-error quantum algorithms do. In fact, in the bounded-error setting, quantum algorithms can even avoid the log factor needed in composing bounded-error randomized algorithms that comes from amplifying the success probability via majority voting. In this article, we try to give some intuition for these results: why composing quantum algorithms is tricky, particularly in the zero-error setting, but why it nonetheless works *better* than classical composition in the bounded-error setting.

1 Introduction

Algorithms designers and programmers make ubiquitous use of subroutines. Not only does this allow code to be easily reused, perhaps even in subroutine libraries used across platforms, it adds structure to a program, or algorithm, that makes the whole thing easier to analyze. For algorithms designers, the most important reason to use subroutines is probably that by black-boxing the algorithms designed by your ingenious colleagues, you can use them as building blocks within your own new algorithms.

Composition is a fundamental idea in complexity theory as well. One of the motivations for using polynomials as the class of growth functions considered “efficient” is that we feel intuitively that whenever we compose “efficient” algorithms, the resulting algorithm should also be “efficient” [AB09]. Perhaps one reason we take for granted that this should be true is the ease with which we can compose and analyze classical algorithms. If an algorithm \mathcal{A} makes Q queries to a subroutine \mathcal{B} with time complexity $T(\mathcal{B})$, and L additional operations, then its complexity is:

$$T(\mathcal{A}) = Q \cdot T(\mathcal{B}) + L,$$

as any first-year computer science student could tell you. This allows us to say things about the complexity of *composed* functions. Let $f : \{0, 1\}^n \rightarrow \{0, 1\}$ be the *composed function* $f = g \circ h$,

¹© S. Jeffery, 2024.

²QuSoft, CWI & University of Amsterdam, Amsterdam, the Netherlands. jeffery@cwi.nl. Supported by ERC (ASC-Q) 101040624, NWO OCENW.Klein.061, NWA-ORC 1389.20.241, and the CIFAR QIS program.

where $g : \{0,1\}^m \rightarrow \{0,1\}$ for some $m = m(n) \leq n$, and $h : \{0,1\}^{n/m} \rightarrow \{0,1\}$, defined on $x \in \{0,1\}^n$ by

$$f(x) = f(x^{(1)}, \dots, x^{(m)}) = g(h(x^{(1)}), \dots, h(x^{(m)})),$$

where each $x^{(i)}$ is a n/m -bit string. Let $\mathbf{D}(f)$ be the *deterministic query complexity* of f , also called the *decision tree complexity*, which is the minimum number of queries to the bits of the input needed to decide f (on the worst-case input). Then we can immediately see that

$$\mathbf{D}(g \circ h) \leq \mathbf{D}(g) \cdot \mathbf{D}(h),$$

by simply composing query-optimal deterministic algorithms for g and h (that is, every time the algorithm for g makes a query to the input, replace it with a subroutine call to h on the appropriate part of the input).

A similarly easy composition applies to the *expected* running time of *Las Vegas* algorithms, which are randomized algorithms that always output the correct answer, but might run forever: In a Las Vegas algorithm \mathcal{A} , for any fixed input x (which we generally leave implicit), the running time $T(\mathcal{A})$ of the algorithm is a random variable, and while its expected value (over the random choices made by the algorithm) is generally finite, the maximum value it may obtain could be unbounded. If \mathcal{A} is a Las Vegas algorithm that makes Q queries to a Las Vegas subroutine \mathcal{B} , and L additional operations, then³

$$\mathbb{E}[T(\mathcal{A})] = \mathbb{E}[Q] \cdot \mathbb{E}[T(\mathcal{B})] + \mathbb{E}[L].$$

Let $\mathbf{R}_0(f)$ denote the zero-error randomized query complexity of f , which is the minimum expected number of input queries needed by any Las Vegas algorithm for f (on the worst case input). Then we can easily see that

$$\mathbf{R}_0(g \circ h) \leq \mathbf{R}_0(g) \cdot \mathbf{R}_0(h). \quad (1)$$

This naive composition doesn't quite work if we use randomized algorithms that have some probability of error. Suppose $\tilde{\mathcal{B}}$ is a *bounded-error* (or Monte Carlo) algorithm, meaning that it is a randomized algorithm that outputs the correct answer with probability at least $2/3$ on every input, and, in contrast to Las Vegas algorithms, for every input x , there is some finite number of steps after which it is guaranteed to have halted – we call this the running time, and it is not a random variable. Note that Monte Carlo algorithms are generally easier to come by than Las Vegas algorithms, since you can turn any Las Vegas algorithm with worst-case expected running time T into a Monte Carlo algorithm with (deterministic) running time $O(T)$ by simply stopping the Las Vegas algorithm after $3T$ steps. By Markov's inequality, the probability the Las Vegas algorithm has not yet found the correct answer is at most $1/3$.

While $2/3$ is generally considered an acceptable correctness probability, in the context of a subroutine that is called Q times by an algorithm \mathcal{A} that expects the *right* answer, this is insufficient. Assuming $Q \gg 1$, with overwhelming probability, one of the calls to $\tilde{\mathcal{B}}$ will return the wrong answer, in which case, we no longer have any guarantees on the correctness of \mathcal{A} . To ensure that with probability at least $2/3$, all subroutine calls return the correct answer, we need the probability of error on each subroutine call to be $\ll \frac{1}{Q}$, so we can apply a union bound. The standard way to achieve this is as follows: every time \mathcal{A} wants to call the subroutine, we will make about $\log Q$

³We make the reasonable assumption that the number of queries, Q , is independent of the time the queries take, which would be true, for example, if \mathcal{A} were designed with each subroutine query treated as a black box whose cost is always unit.

parallel calls to $\tilde{\mathcal{B}}$, and return the most commonly seen answer (the majority)⁴. Then, by a binomial tail bound, each of the Q queries made by \mathcal{A} is correct except with probability $\ll \frac{1}{Q}$, as needed. If the resulting algorithm is called $\tilde{\mathcal{A}}$, we have:

$$T(\tilde{\mathcal{A}}) = O\left(Q \cdot T(\tilde{\mathcal{B}}) \log Q + L\right). \quad (2)$$

Letting $\mathbf{R}_2(f)$ denote the *bounded-error randomized query complexity* of f , which is the minimum query complexity of a bounded-error randomized algorithm that decides f , we then have:

$$\mathbf{R}_2(g \circ h) \leq O(\mathbf{R}_2(g) \cdot \mathbf{R}_2(h) \log m). \quad (3)$$

This inequality is tight: there exist composed functions for which this log factor is necessary [BB19].

In complexity-theoretic terms, this gives us, for deterministic, zero-error, and bounded-error algorithms respectively:

$$\mathbf{P}^{\mathbf{P}} = \mathbf{P}, \quad \mathbf{BPP}^{\mathbf{BPP}} = \mathbf{BPP}, \quad \text{and} \quad \mathbf{ZPP}^{\mathbf{ZPP}} = \mathbf{ZPP},$$

(all of which relativize), meaning that each of these three classes is *self-low*. These complexity-theoretic statements are, of course, quite granular compared to the finer-grained complexities of composition described above, but they roughly state that each of the three types of algorithms – efficient deterministic, efficient Monte Carlo, and efficient Las Vegas – are “closed under composition”, as one would intuitively hope.

All of this is quite elementary, and I don’t mean to be a bore. My point in saying all of this is to set up a contrast with composition in *quantum* algorithms, which is *not* so simple. The first indication of this, to my knowledge, was a result by Buhrman and de Wolf in 2002 showing that a quantum analogue of (1) does *not* hold [BdW03]. They showed an oracle relative to which

$$\mathbf{ZQP}^{\mathbf{ZQP}} \neq \mathbf{ZQP}.$$

One reason that this is surprising is that *bounded-error* quantum query complexity is known to compose nicely. There is nothing that prevents a quantum algorithm from calling a subroutine, and an analogue of (3) for quantum algorithms is quite straightforward. But more than that, it is possible to show an even stronger statement, *without* log factors:

$$\mathbf{Q}_2(g \circ h) \leq O(\mathbf{Q}_2(g) \cdot \mathbf{Q}_2(h)), \quad (4)$$

where $\mathbf{Q}_2(f)$ is the minimum quantum query complexity of any bounded-error quantum algorithm for f [Rei09].

We will return to this way in which quantum composition works *better* than classical shortly. For now, we focus on the *difficulty* of composing quantum algorithms, which stems from (at least) two issues:

1. Whereas classical algorithms may call a distribution of different subroutines, the quantum analogue is a *superposition* of different subroutines. Unlike a classical distribution, which can be visualized as a tree, quantum branches of a superposition cannot be considered as separate

⁴Note that majority voting only works if the desired behaviour of $\tilde{\mathcal{B}}$ is to output a deterministic bit or string, as opposed to a distribution, or an element from a set of many possible correct answers.

non-interacting branches forever. We generally visualize them as coming back together for the next operation – indeed, unlike randomness, it is possible to evolve a superposition supported on many states back to a point function supported on a single state. So, naively, one must wait for all branches of the superposition to terminate before the next operation can be applied.

2. The quantum analogue of probability is *amplitude*, and it can be negative. Nonzero amplitude on an accepting state may later be cancelled by negative amplitude, in contrast to nonzero probability on an accepting state, which, in a zero-error classical algorithm, is definitive evidence that “accept” is the correct outcome.

The second issue is severe, and is why zero-error quantum algorithms do not compose, which we discuss more in Section 3. The first issue turns out to be surmountable, if we simply look at quantum algorithms the right way.

To illustrate the first issue, imagine having (classical) subroutines $\mathcal{B}_1, \dots, \mathcal{B}_N$ called by an algorithm \mathcal{A} , and let Q_i be the number of calls to \mathcal{B}_i . Then clearly

$$\mathbb{E}[T(\mathcal{A})] = \sum_{i=1}^N \mathbb{E}[Q_i] \mathbb{E}[T(\mathcal{B}_i)] + \mathbb{E}[L], \quad (5)$$

where, as usual, L is the number of additional operations used by \mathcal{A} . For simplicity, suppose the total number of subroutine queries made by \mathcal{A} , Q , is deterministic, and let $p_{j,i}$ be the probability that the j -th subroutine call is to \mathcal{B}_i . Then we can rewrite this:

$$\mathbb{E}[T(\mathcal{A})] = \sum_{i=1}^N \sum_{j=1}^Q p_{j,i} \mathbb{E}[T(\mathcal{B}_i)] + \mathbb{E}[L]. \quad (6)$$

An analogous statement for quantum algorithms is not obvious, and even in the case where the running times $T(\mathcal{B}_i)$ are deterministic (which does simplify things for quantum algorithms), naive composition yields a statement analogous to:

$$\mathbb{E}[T(\mathcal{A})] = Q \max_{i \in [N]} T(\mathcal{B}_i) + \mathbb{E}[L]. \quad (7)$$

This is because, for the most part, quantum algorithms have been viewed in the *quantum circuit model*, which treats them in analogy to classical deterministic algorithms, where each operation is something to be done once, at a fixed time. If the j -th subroutine call is something that happens once, at a fixed time, then it certainly can’t happen before the $(j-1)$ -th subroutine call has terminated (in all branches of superposition), and so the algorithm must wait $\max_{i \in [N]} T(\mathcal{B}_i)$ time steps, assuming each subroutine has a non-zero probability of being called in the $(j-1)$ -th query.

In contrast, randomized algorithms are often viewed as graph-like structures, with probability flowing through them, and so a particular operation might be applied at some step with some probability, and at a later step with some other probability. Taking a similar view of quantum algorithms turns out to make composition much more straightforward. While we can’t hope to prove a quantum statement analogous to (5) for \mathcal{A} a Las Vegas algorithm⁵, due to the result of Buhrman and de Wolf [BdW03], we do have the next best thing: a quantum analogue of (6) for \mathcal{A} a *bounded-error* quantum algorithm [Jef22]. This quantum analogue does not use naive quantum composition

⁵In fact, *quantum Las Vegas complexity*, is better defined in a more natively quantum way, as in [BY23], on which [BJY24] is partially based.

to obtain \mathcal{A} , but a more involved way of composing quantum algorithms, viewing them as quantum analogues of random walks. The results of [Jef22] apply only to subroutines that compute a single bit, but in [BJY24] we generalize this to quantum subroutines that compute any *unitary map* (the kind of map quantum computations are capable of), using a new quantum algorithmic model that generalizes quantum random walks, called *transducers*. In Section 2, we discuss in more detail why quantum composition is different from classical composition, but nonetheless works, if we are satisfied with getting a bounded-error algorithm. This is done using pictures, rather than equations, so it is easy to follow on an intuitive level, but does not contain enough detail for a rigorous understanding. For that, the reader may refer to [Jef22] or [BJY24].

I'd like to return your attention to (4), where we seem to magically save a log factor. Recall that the log factor repetition in composed classical bounded-error algorithms comes from repeating each subroutine call $\log Q$ times, in order to decrease the error of each subroutine query to around $\frac{1}{Q}$ via majority voting. Are quantum algorithms somehow able to avoid this repetition?

For a while I was doubtful that avoiding the log-fold repetition was possible. The result in (4) is proven non-constructively, so it could be that any algorithm for $f = g \circ h$ achieving the optimal query complexity $O(\mathbf{Q}_2(g)\mathbf{Q}_2(h))$ does have some kind of log repetition, it's simply recycling queries, somewhat like randomness can be recycled in majority voting to avoid a log-factor overhead in the random bits needed in error-reduced algorithms [IZ89].

However, it turns out quantum algorithms really can avoid repeating subroutines log-many times: we were able to show that if \mathcal{A} is a bounded-error quantum algorithm that makes calls to a subroutine, whose desired functionality is implemented with bounded error by another quantum algorithm $\tilde{\mathcal{B}}$, then there is a quantum algorithm $\tilde{\mathcal{A}}$ that implements the desired behaviour of \mathcal{A} in complexity [BJY24]:

$$T(\tilde{\mathcal{A}}) = O\left(Q \cdot T(\tilde{\mathcal{B}}) + L\right),$$

where, as usual, L is the number of additional operations made by \mathcal{A} , giving a quantum analogue of (2), but without the log factor. A similar log-factor-free statement applies when there are many different subroutines implemented with bounded error, in which case we get a complexity analogous to (6), up to constant factors.

In order to prove this, we make use of our new formalism for quantum algorithms, the aforementioned *transducers*. If an algorithm has some error with respect to its desired behaviour, it will map to a *perturbed* (with respect to the desired behaviour) transducer. Just as an algorithm's error can be reduced to any ε at a $O(\log \frac{1}{\varepsilon})$ overhead using majority voting, a transducer can be *purified* to a transducer with arbitrarily small perturbation δ , but unlike majority vote, purification results in a $O(1)$ multiplicative overhead, *independent of δ* . Whenever we compile transducers back into algorithms, a constant error ε is introduced, resulting in a bounded-error algorithm, so this does not give us a way to turn a bounded-error quantum algorithm into a $o(1)$ -error quantum algorithm with just $O(1)$ overhead. However, it does give a way to compose without log factors, by purifying all subroutines to have arbitrarily small perturbation, so that when the transducers are composed to get a new transducer, the cumulative perturbation is still $o(1)$, and then turning back into a bounded-error quantum algorithm.

I'm not really a complexity theorist – though I am friends with several – so I am not entirely sure where they stand on log factors, but it seems they are not too bothered by them – after all, they're not even polynomial! To be honest, as an algorithms person, me neither, for the most part.⁶

⁶They can pose a rather more severe problem when composing algorithms to non-constant depth.

In analyzing algorithms, I’m wont to hide log factors – even polylog factors – in my asymptotics, using the notation $\tilde{O}(\cdot)$. However, I think it is still quite striking that these log factors need not appear in composed quantum algorithms. I think it may give us some insight into the power of quantum computers, although I don’t yet know what that insight is, and won’t be telling you about it here. But in Section 4, I will discuss this result more, and give some idea of where it comes from using a toy example.

Note on the model of computation: When we talk about classical computation, we assume the word RAM model, in which random access memory reads and writes (to our working memory, and perhaps a memory storing a description of the program to be executed) have unit cost, as do basic operations, such as addition, of words of a size large enough to address this memory. Our quantum model will be the quantum analogue of this. The quantum word RAM model is less standard than its classical counterpart, partially because implementing a quantum random access gate might be much more difficult than quantum gates acting only on one or two qubits (quantum bits). We will not concern ourselves with the question of how realistic this model of quantum computation is in the near-term, as our main interest here is theoretical comparison between classical and quantum computation, so we give the same powers to both models. Ref. [BJY24] gives some weaker composition results that also apply in the strict quantum circuit model, in which random access gates are not consider a basic operation.

2 Quantum vs. Randomized Algorithms

The goal of this section is to illustrate the difficulty of composing quantum algorithms, and specifically, to convey the following:

1. Composing quantum algorithms to achieve a complexity that is a quantum analogue of (6) is not obvious.
2. It is nonetheless possible, with the caveat that the composed algorithm will have bounded error.

We also hope to lay the foundations for seeing why zero-error composition is *not* possible, which we discuss more in Section 3. For simplicity, throughout this section, the reader may assume that the quantum subroutines have no error, and \mathcal{B}_i terminates after a fixed time $T(\mathcal{B}_i)$, though this time may vary in i . Towards these goals, in this section we describe visualizations of randomized and quantum algorithms that allow us to compare them at an intuitive level without the use of math⁷.

Randomized algorithms, as trees A randomized computation can be visualized as a tree (see Figure 1). There is an *entrance* at the root, and outgoing *terminals* at each leaf, each of which is either *accepting* or *rejecting*. The internal tree nodes, shown as rectangles, represent coin flips, and some extra nodes on the edges might represent deterministic computations. We use dashed lines to represent a computation that we model as an oracle call, either querying the input directly, or perhaps meant to be instantiated by a subroutine.

Of course, we sometimes visualize randomized computations as another type of graph, like a DAG, or more general random walk. However, as long as we don’t care about the amount of

⁷Math is still very much recommended for actually proving things. For details, see [Jef22, BJV24].

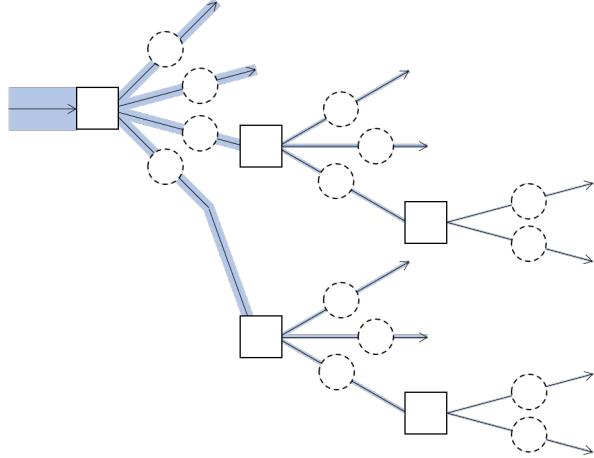


Figure 1: A run of a randomized algorithms, visualized as probability flowing through a tree. The incoming arrow is the entrance, and outgoing arrows are terminals.

memory the algorithm uses, we can always model it as a tree, indicating that we remember every coin flip ever made (even if our future computations might not depend fully on them). So the tree picture is without loss of generality, as far as time complexity is concerned.

Importantly, in our visualizations, the node you are in will not encode the full state of the algorithm, but simply where you are in the computation. We assume a “walker” moving through the tree carries along some extra information as well, based on queries made and computations performed so far. As such a tree’s structure may be adaptive, depending on the input via results of computations in the dashed circles. To visualize a run of the computation on a particular input, we can imagine that we have a unit of probability on the entrance, and as we run the algorithm, it flows through the tree, from left to right, splitting when it gets to an internal tree node (evenly, assuming coin flips are unbiased), until it gets to a terminal. It can be helpful to think of this probability as water flowing through the graph, whose edges are like pipes, possibly carrying with it some information about the algorithm’s current state.⁸ The amount of water already at the terminals after ℓ steps – equivalently, the amount of water at terminals with distance at most ℓ from the entrance – is precisely the probability that the algorithm has halted after at most ℓ steps. Figure 1 also shows this *probability flow*, which includes the probability on the edges at *all* steps of the algorithm, rather than only at a particular step.⁹

It is quite intuitive from Figure 1 that if we instantiate the dashed circles with programs of varying length – for simplicity, imagine deterministic programs, which are just line graphs – we get a complexity (expected distance from entrance to root) as in (6).

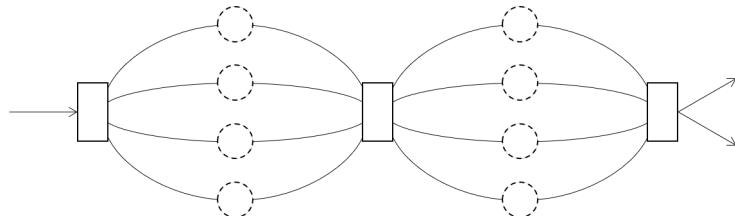
Quantum algorithms are not trees We can view quantum computations in a somewhat similar picture, but we replace the concept of randomness with its quantum analogue, *superposition*, and

⁸If you want to take this metaphor to an obscene level, you can think of the flow as consisting of different types of liquids that don’t mix, like water and oil, whose volumes add up to the total flow. The type of liquid encodes the additional information.

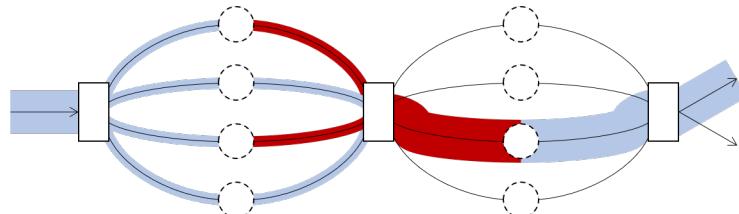
⁹So if you add up all probabilities shown in the probability flow, you will not get 1, but rather, you will get the expected running time of the algorithm.

probability with its quantum analogue *amplitude*. A quantum state is a vector, not unlike a probability distribution, but a quantum state is ℓ_2 -normalized instead of ℓ_1 -normalized like a probability distribution. The squared norms of the amplitudes, which sum to 1, are actually probabilities.

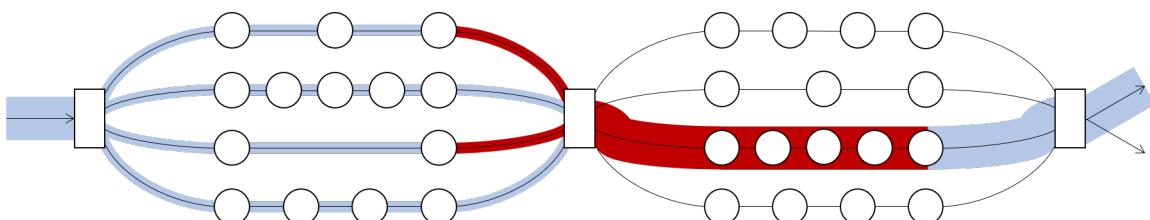
Unlike probability, amplitude can be negative (or even complex), and – actually because of this – in contrast to a random variable, a superposition has no entropy. It's possible to move from a “classical deterministic” state, such as 0^n , into a superposition of many states, and then back to a “classical deterministic” state. So we cannot generally visualize a quantum algorithm as a tree. Instead, it looks more like the diagram below¹⁰:



Each of the rectangles represents some quantum operation, and as above, the dashed circle represents a query to the input (or perhaps a subroutine). There are is an accepting terminal, and a rejecting terminal. In this visualization, any Q -query quantum algorithm looks the same, so this basic picture tells us very little about the program's structure. To visualize a run of the algorithm, we can imagine its amplitude flowing through the graph, as we did with probabilities.



When we show *amplitude flows*, the strength of the flow will be visualized as the *square* of the amplitude, so they all add up to 1 (on any entrance-terminals cut), whereas the colour will tell us the sign: blue for positive, and red for negative¹¹. Visualizing a quantum algorithm this way is, admittedly, much less helpful and intuitive than visualizing a randomized algorithm as a tree, but it does allow us to think about quantum composition. Imagine you want to instantiate the dashed circles with calls to different subroutines. A deterministic subroutine is just a line, so we get something like the following:



¹⁰For readers used to quantum circuit diagrams: the lines do *not* represent different qubits, but rather, different states, so in an n -qubit system, there are 2^n of them.

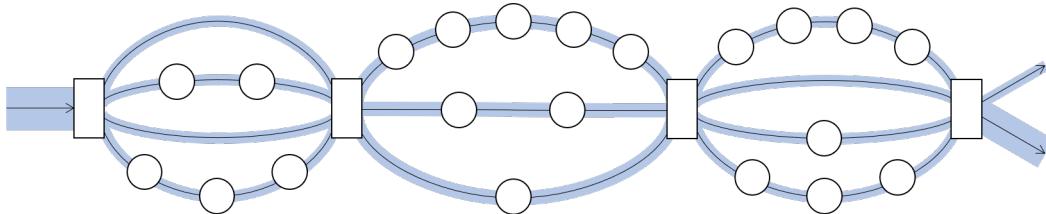
¹¹In general, the amplitude could take any direction $e^{-i\theta}$ on the unit circle, but we will restrict to positive, (+1) and negative (-1).

Each circle above is a subroutine step. The number of subroutine steps varies in different branches of the superposition, just as the number of subroutine steps might vary in different tree branches in a randomized algorithm. What is the complexity of this composed quantum algorithm? To run this algorithm, first we apply the first quantum operation, represented by the first rectangle. Before we apply the second, it seems like we need to complete all subroutine calls – consisting of the operations represented by circles – because the second rectangle depends on all the inputs, so naively, we have to wait the maximum path length. This is certainly true if we think of the second operation as something we’re going to apply once, at some fixed time, as we do in the most common model of quantum algorithms, the *quantum circuit model*. This gives complexity:

$$T(\mathcal{A}) = Q \max_{i \in [N]} T(\mathcal{B}_i) + L, \quad (8)$$

which, compared to (6), is terrible, as we’ve replaced an average with a maximum. This illustrates that it is not straightforward to have quantum subroutines that take different amounts of time in different branches of the superposition. It seems like we need to wait for the slowest branch of the superposition to terminate before we can move on to the next step, and until a couple of years ago, no better expression than (8) was known.

This issue doesn’t come up in a tree, obviously, since paths never come together again. However, as previously mentioned, we often do visualize randomized algorithms as something other than a tree. What is the expected time to get from the entrance to one of the terminals in the following DAG (all edge directions are left-to-right), representing a randomized computation where we forget whatever randomness has been used so far?



It is obviously the expected path length. As “liquid” moves through the graph, it eventually all ends up at the terminals. It doesn’t matter if you let liquid through the rectangles as it arrives, or wait for it all to arrive before distributing it out the other side, it will all end up at the terminals, carrying the same information either way, because probability can do nothing but add up.

It turns out that this intuition also holds (with somewhat more difficulty in making it precise enough to turn into a proof) in quantum algorithms. If we think of the quantum algorithm in analogy to a random process – actually, as the quantum analogue of a random walk – we find that it’s fine if we send some amplitude through a box before the rest of the amplitude has caught up: it will all add up to the same thing on the terminals in the end, no matter what speed different parts of it get there. Thus, if we combine the outer quantum algorithm, consisting of just the boxes and the dashed circles representing queries, with the quantum subroutines instantiating those queries – in a particular way inspired by quantum random walks – everything will add up correctly to the right state on the terminals after the final rectangle. And if we just wait until *most* of the amplitude has made it through the final rectangle, we will end up with a state that is *mostly* correct, giving

a bounded-error quantum algorithm \mathcal{A} with complexity:

$$T(\mathcal{A}) = O \left(\sum_{j=1}^Q \sum_{i=1}^N q_{j,i} T(\mathcal{B}_i) + L \right), \quad (9)$$

where, Q is the number of subroutine calls, and $q_{j,i}$ is the squared norm of the amplitude on the i -th superposition branch right before the j -th subroutine query. This was shown (with polylog factor overhead) for subroutines that compute a single bit in [Jef22], and later generalized to arbitrary (unitary) quantum subroutines in [BJY24]. We can even replace the complexities $T(\mathcal{B}_i)$ with expected complexities (we have not talked about how quantum algorithms can have a running time that is a random variable, but it is possible) to get a quantum analogue of (6).

We have been assuming, for simplicity that the subroutines have no error. If we instead have bounded-error subroutines, then we can use majority voting (assuming the desired behaviour of the subroutines is to output a deterministic string) to obtain the complexity in (9), but with logarithmic overhead. However, using *purifiers*, we can even get this complexity expression with no log factors even when the subroutines have bounded error. We discuss this more in Section 4.

The reason quantum composition does not work in the zero-error case, that is, if we want the composed algorithm \mathcal{A} to have zero error, is that we would need to wait for all amplitude to get to the end of the graph if we want to have no probability of error. In the next section, we will explore this subtle difference between quantum and classical composition.

3 Impossibility of Zero-error quantum Composition

In [BdW03], Buhrman and de Wolf show that there is an oracle A such that

$$\text{ZQP}^{\text{ZQP}^A} \neq \text{ZQP}^A,$$

by exhibiting a composed $f = g \circ h$ for which $\mathbf{Q}_0(g) = 1$, $\mathbf{Q}_0(h) = O(1)$, and $\mathbf{Q}_0(f) \geq \frac{m}{2} + 1$. We describe this composed function, and try to give some intuition why its zero-error quantum query complexity is not also constant.

Let $|x|$ denote the Hamming weight of a binary string x . Let g be the promise problem on the set $\{x \in \{0,1\}^m : |x| \in \{0, m/2\}\}$ for even m , defined:

$$g(x) = \begin{cases} 1 & \text{if } |x| = 0 \\ 0 & \text{if } |x| = m/2. \end{cases}$$

This function decides if a string is *constant* 0^m ; or *balanced*, meaning it has the same number of 0s and 1s. It is well known in the quantum algorithms literature, because there is a quantum algorithm due to Deutsch and Jozsa [DJ92] – one of the very first quantum algorithms in fact – that outputs the correct answer using a single query to the input. So in particular, $\mathbf{Q}_0(g) = 1$ (in fact, we have the stronger statement $\mathbf{Q}_E(g) = 1$, where \mathbf{Q}_E is the *exact quantum query complexity*, since 1 is not merely the expected complexity, but an upper bound on the complexity). In contrast, if the input is constant, then a classical algorithm needs to see $m/2 + 1$ bits to be certain that they are all the same, so $\mathbf{R}_0(g) = m/2 + 1$.

The Deutsch-Jozsa algorithm, which decides g exactly with a single quantum query, relies precisely and crucially on the promised structure of the problem, which we can illustrate in our

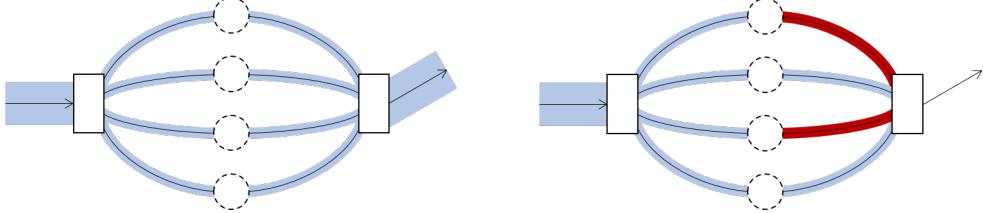


Figure 2: On the left, we see a run of the algorithm on a constant input, and on the right, a run of the algorithm on a balanced input. In both cases, we first do a quantum Fourier transform (first box) to split one unit of amplitude uniformly into m branches. In branch $i \in [m]$, we query x_i (the dashed circle), changing the sign from positive (blue) to negative (red) if $x_i = 1$. Finally, we invert the Fourier transform (last box), sending the sum of the amplitudes on all branches onto the accepting terminal. When x is constant (left-hand side), all paths stay blue (positive), so they add up to a unit of amplitude going out the accepting terminal. When x is balanced (right-hand side), half the paths become red (negative) and cancel with the blue paths, so they add up to 0 amplitude on the accepting terminal. Note that there are other implicit terminals (not shown), all of which are rejecting.

intuitive graph picture, in Figure 2. The algorithm first does something called a *quantum Fourier transform*, which maps $0^{\log m}$ to a uniform superposition over all $\log m$ -bit strings – a bit like flipping $\log m$ (quantum) coins – representing each $i \in [m]$. Next, in each branch of the superposition, the algorithm queries the i -th bit. It does a pretty quantum thing with that bit: if the bit is 1, it changes the sign of the amplitude from positive to negative. That's shown as the colour changing from blue to red. Finally, it does something called an inverse Fourier transform, and the only thing we need to know about that is that the amplitude on $0^{\log m}$ coming out of that operation – which is the accepting state/terminal (the rejecting terminal is not pictured, but all other states are rejecting) – is the sum of all incoming amplitudes¹². So one of two things happens:

Constant (accepting) Case: all the bits are 0, meaning all paths are blue, so they add up to unit amplitude (corresponding to probability 1) coming out the accepting terminal; or

Balanced (rejecting) Case: half the bits are 0 and half the bits are 1, meaning there are as many blue paths as red paths, and so they cancel, resulting in 0 amplitude (corresponding to probability 0) coming out the accepting terminal.

Note that if the Hamming weight were not exactly $m/2$ in the rejecting case, we would not get the perfect cancellation that makes this an exact, and thus a zero-error, algorithm.

The inner function h defining $f = g \circ h$ will also be a promise problem, and we will define it by its pre-images:

$$\begin{aligned} h^{-1}(1) &= \{0^m x^R : x^R \in \{0,1\}^m, |x^R| \geq m/2\} \\ h^{-1}(0) &= \{x^L 0^m : x^L \in \{0,1\}^m, |x^L| \geq m/2\}. \end{aligned}$$

¹²This is actually not completely correct, but it gives the right mental picture if you're not following too closely. The actual amplitude on the accepting terminal is $1/\sqrt{m}$ times the sum of the m incoming amplitudes, each of which is either $1/\sqrt{m}$ (blue) or $-1/\sqrt{m}$ (red). The square roots are because quantum states are ℓ_2 -normalized.

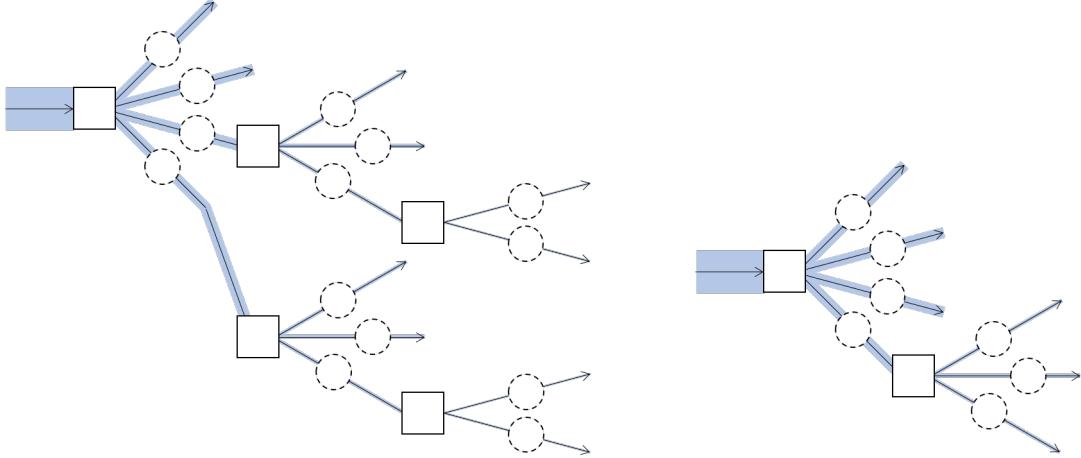


Figure 3: Here we see the probability flow for two runs of a randomized algorithm for h on different inputs: one with $|x| = m/2$ (left-hand side); and one with $|x| = 3m/4$ (right-hand side). The dashed circles represent calls to a small subroutine that queries x_j and x_{m+j} in the j -th branch, and terminates if one of them is 1, in either an accepting terminal (if $x_{m+j} = 1$) or a rejecting terminal (if $x_j = 1$). In the left-hand image, which is the worst case, half the probability is already on terminals after one such step, and for the remaining half, a new $j' \in [m] \setminus \{j\}$ is chosen. The longest paths from the entrance to a terminal in the worst case are $m/2 + 1$ steps, but the total probability weight on these is very small.

So we are promised that one half of the string is all-0s, and the other half has relatively many 1s, and we must decide which half is which. Even a classical algorithm can solve this with zero error in expected constant time: Alternatively query a random as-of-yet-unqueried bit x_j on the left-half and, the corresponding x_{j+m} on the right half until you see a 1, at which point, you're done. In the tree picture, this looks like Figure 3.

If you're maximally unlucky, you will query all the 0s in the half with 1s before you query a 1, meaning you will spend $2(m/2 + 1) = m + 2$ queries, but this only happens with probability $\Theta(2^{-m})$. Since every two queries finds a 1 with probability at least $1/2$, the expected number of queries before a 1 is found is $O(1)$. Thus $\mathbf{Q}_0(h) \leq \mathbf{R}_0(h) = O(1)$.

We have seen that $\mathbf{Q}_0(g)$ and $\mathbf{Q}_0(h)$ are both $O(1)$, so, in light of the fact that $\mathbf{R}_0(g \circ h) \leq \mathbf{R}_0(g) \cdot \mathbf{R}_0(h)$, we might expect $\mathbf{Q}_0(g \circ h)$ to be $O(1)$, but in fact, we have [BdW03]:

$$\mathbf{Q}_0(g \circ h) \geq m/2 + 1.$$

The classical result $\mathbf{R}_0(g \circ h) \leq \mathbf{R}_0(g) \cdot \mathbf{R}_0(h)$ is based on the observation that we can simply run an optimal Las Vegas algorithm for g , and every time the algorithm tries to query the input, replace the query with a call to the optimal Las Vegas algorithm for h (on the appropriate block of the input). Let us try to understand what goes wrong with quantum composition, by considering running the single-query Deutsch-Jozsa algorithm for g , but replacing the query with a call to our algorithm for h (which we can turn into a quantum algorithm without much difficulty).

To see the issue, consider an input $x = (x^{(1)}, x^{(2)}, x^{(3)}, x^{(4)})$ to $g \circ h$ (so $m = 4$) that induces a balanced input to g , meaning that exactly half of the strings $x^{(1)}, x^{(2)}, x^{(3)}$, and $x^{(4)}$ are 1-inputs to h . Each part, $x^{(i)}$, of x is a $2m$ -bit string with two parts, $x^{(i,L)}$ and $x^{(i,R)}$, one of which has weight

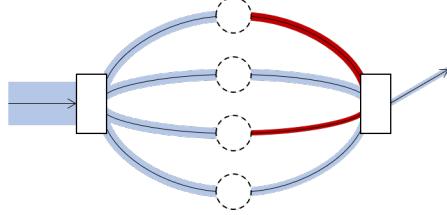


Figure 4: This image depicts the probability flow through the Deutsch-Jozsa algorithm from Figure 2, if we instantiate the queries with the subroutine for h from Figure 3, and pause the subroutine after one step. We are assuming the different inputs to h have different weights, all at least $m/2$, resulting in different amplitudes coming out of the subroutine after one step, all at least half. However, since these amplitudes are not *the same*, even though two are positive and two are negative, they do not perfectly cancel, so there is non-zero amplitude on the accepting terminal.

0, the other of which has weight at least $m/2$. Suppose further that x has the following structure:

$$x = \begin{array}{c|cc|cc|cc|cc} & 0^m & x^{(1,R)} & x^{(2,L)} & 0^m & 0^m & x^{(3,R)} & x^{(4,L)} & 0^m \\ \text{weight} & 0 & \frac{3m}{4} & m & 0 & 0 & \frac{m}{2} & \frac{3m}{4} & 0 \end{array}$$

This satisfies the promise, because all non-zero strings have weight at least $m/2$.

Consider what happens after we've run one step of the algorithm for h – a step being two queries, one for each half of the input to h . This is depicted in Figure 4. In each of the four branches of the superposition, the subroutine for h has terminated with probability at least half, which we depict by some amplitude coming out of the subroutines (the remaining amplitude, not shown, is still inside the subroutine). The size of these amplitudes depends on the weight of the non-zero string in the input to h . In the top branch of the superposition, the input to h is $x^{(1)} = 0^m x^{(1,R)}$. Since $|x^{(1,R)}| = 3m/4$, with probability $3/4$, we find a 1 in the first step, and conclude that $h(x^{(1)}) = 1$, since the one is in $x^{(1,R)}$, the right half. We thus change the amplitude to negative, shown as red – since the subroutine has zero error, only red amplitude will ever exit the subroutine call in this branch. In the second branch, since $|x^{(2,L)}| = m$, we will find a 1 in the first step with certainty, so all amplitude has already exited the subroutine after one step, and it is positive (blue). In the third branch, since $|x^{(3,R)}| = m/2$, half the amplitude exits after one step, and in the fourth branch, since $|x^{(4,R)}| = 3m/4$, $3/4$ of the amplitude exits after one step.

The issue is that the amount of positive and negative amplitude is not the same, and so these amplitudes do not perfectly cancel, meaning there is a nonzero amplitude on the accepting terminal, even though accepting would be incorrect for this input. If we waited for all amplitudes to exit the subroutine, then we would have the same amount coming out of each, and these would cancel, leaving 0 amplitude on the accepting terminal, but if we stop early conditioned on the subroutine being finished after one step, we will have some non-zero probability of outputting the wrong answer.

Note that this cannot happen in a classical algorithm – once you have any non-zero probability on a terminal, you're safe to accept or reject accordingly, because there's no chance that later probability will arrive and cancel it out, so that it would have been zero, had you waited.

The difference between a classical zero-error algorithm and a quantum zero-error algorithm is that in the classical picture, once we have some amount of probability on an accepting terminal, it means, with certainty, that 1 is the correct answer, and once we have some nonzero probability

on a rejecting terminal, it means, with certainty, that 0 is the correct answer. This is related to the fact that a randomized algorithm can always be viewed as a tree: once you get to a terminal (leaf) by some path, there is no other path that could come and change things. This is in contrast to quantum algorithms, where even if at some point you have nonzero amplitude on an accepting terminal, it doesn't mean the answer is 1 – it could be that if you wait, negative amplitude will come and cancel the positive amplitude so that the resulting amplitude on the accepting terminal by the end of the algorithm is 0.

4 Transducers, Purifiers, and no more log Factors

I have already mentioned that by looking at quantum algorithms as quantum random walks, we can get some nice composition results. However, the results obtained this way are limited to composing subroutines that compute classical functions, whereas a quantum subroutine might potentially map a quantum state to another quantum state, through any *unitary* linear map. For this more general type of composition, we need a more general model, called *transducers* [BJY24]. Moreover, composing via transducers is how we can achieve composition of bounded-error quantum algorithms without log factors. That's what we will talk about in this section.

Like a quantum algorithm, a transducer has an associated unitary action, and a complexity. I will not give details about what a transducer is, but it has the following properties that make it a useful abstraction for quantum algorithms:

1. There is a mapping that takes any quantum algorithm implementing a unitary map U in complexity T , and compiles it into a transducer for U with complexity $O(T)$.
2. There is a mapping that takes any transducer for U with complexity T , and compiles it into a quantum circuit implementing U with *bounded error* in complexity $O(T)$.
3. Transducers compose nicely: You can combine a transducer for some outer algorithm that makes oracle calls, with transducers for some subroutines, to get a transducer for the composed functionality. The complexity will have a nice expression, like that in (9).

Often we want a quantum algorithm that decides a function, $f : \{0, 1\}^n \rightarrow \{0, 1\}$, rather than implementing some arbitrary unitary. We say a unitary U_x parametrized by an input x decides f if it maps the quantum state that is a point function on the all-0s string, 0^m for some m (a natural starting state) to the quantum state that is a point function on the string $0^{m-1}f(x)$. A quantum algorithm decides f if it implements a unitary U that decides f .

Often we have quantum algorithms that don't perfectly implement the desired unitary map, U , but rather, implement it up to bounded error. If we try to map this to a transducer, it will also only implement U up to some error. A δ -*perturbed* transducer for U implements a map that is δ -close to U in some sense of closeness. When $\delta = 0$, we say the transducer is *perfect* (wrt some U – often a U that decides a function f).

When we compose transducers, the perturbations add up, similar to how errors add in algorithmic composition, so we need the perturbations to be sufficiently small so that even when they add up, we can bound them well below 1. If all of our subroutines decide functions, then we can use majority voting to decrease their error sufficiently. This incurs log factors: D -round majority voting has a multiplicative overhead of D , but reduces the error to $2^{-\Theta(D)}$, so that when we transform the

subroutines into transducers, the small perturbations add up to a total perturbation well below 1, and so we can turn this into a bounded-error quantum algorithm.

However, as stated, we can avoid the log factor overhead. It turns out that there is an analogue of majority voting for transducers that decide a function, which reduces their perturbations to $2^{-\Theta(D)}$ for any D , with just a $O(1)$ multiplicative overhead on the complexity. This construction is called a purifier, and we will shortly give a purifier for a simplified case, to illustrate how this process works. First, it is worth emphasizing that no matter how much we reduce perturbations in transducers, when we compile them back into algorithms, there will be a small constant error. Thus, there is no way to reduce the error of an algorithm for free by turning it into a transducer, purifying it, and then turning it back into an algorithm. This trick is just powerful enough to let us avoid the log factor overhead in composing bounded-error quantum algorithms.

A toy problem To give some idea of how purification works, we will consider the special case of a quantum algorithm that outputs the quantum analogue of a biased coin that gives 0 with some probability $p_0 = p_0(x)$, depending implicitly on some input x :

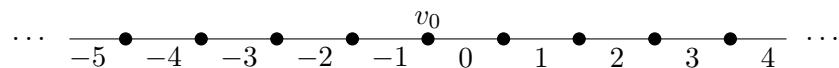
$$\sqrt{p_0} \mathbf{e}_0 + \sqrt{1 - p_0} \mathbf{e}_1. \quad (10)$$

This is a 2-dimensional vector, and there are squareroots over the probabilities because quantum states are ℓ_2 -normalized. We could call such a state a biased quantum coin flip. We imagine that there is some constant $\varepsilon \in [0, 1/2)$ such that one of the following two cases holds, and we want to decide which one:

$$\begin{aligned} \text{Rejecting Case: } & p_0 \geq 1 - \varepsilon \\ \text{Accepting Case: } & p_0 \leq \varepsilon. \end{aligned} \quad (11)$$

This setting does not fully capture the setting of bounded-error quantum algorithms, as we are assuming the algorithm's output is in a known two-dimensional space, but it is the simplest to analyze. But before we think about this quantum problem, let us think a little bit about the classical version of this problem.

A classical walk on a line If you were given a (classical) biased coin with the promise that one of the conditions in (11) holds, and you want to know which one, your instinct would likely be to flip the coin many times, and take a majority vote. One way to model a majority vote is as a weighted random walk on a line. You can use the coin to implement a weighted walk on a line where the edges are labelled by the integers:



and the weight of edge ℓ is

$$w_\ell = \left(\frac{1 - p_0}{p_0} \right)^\ell. \quad (12)$$

In such a graph, from vertex v_ℓ , whose outgoing edges are $\ell - 1$ and ℓ , the probability of moving left, q_L , and right q_R , respectively, are:

$$q_L = \frac{\left(\frac{1-p_0}{p_0}\right)^{\ell-1}}{\left(\frac{1-p_0}{p_0}\right)^{\ell-1} + \left(\frac{1-p_0}{p_0}\right)^\ell} = p_0 \quad \text{and} \quad q_R = \frac{\left(\frac{1-p_0}{p_0}\right)^\ell}{\left(\frac{1-p_0}{p_0}\right)^{\ell-1} + \left(\frac{1-p_0}{p_0}\right)^\ell} = 1 - p_0.$$

Thus, you can take steps on this graph by flipping your biased coin, and moving to the left when you see a 0, and right when you see a 1.

If you begin in v_0 , and then use D coin flips to walk for D steps, you will end up to the left of v_0 if you see 0s a majority of the time over your D coin flips, and to the right if you see 1s the majority of the time over your D coin flips. Thus, your position on the line is simply a way to remember how many more 1s than 0s you've seen, so that you can decide, after D steps, what the majority was.

A quantum walk on a line For the quantum analogue, we need some definitions (this will be the most technical part of the article). Let $G = (V, E)$ be a weighted undirected graph, with edge weights $\{w_e\}_{e \in E}$. The *total weight* of G is defined

$$\mathcal{W}(G) = \sum_{e \in E} w_e.$$

Let s and t be distinct vertices in V . A *unit st-flow* on G is a function θ on $V \times V$ such that

1. $\theta(u, v) = -\theta(v, u)$ for all $u, v \in V$;
2. $\theta(u, v) = 0$ whenever $\{u, v\} \notin E$;
3. for all $u \in V \setminus \{s, t\}$, $\sum_v \theta(u, v) = 0$; and
4. $\sum_v \theta(s, v) = \sum_v \theta(v, t) = 1$.

We often think of a flow as water flowing through the graph from s to t : a unit of flow enters at s – we can imagine this happening through an additional *boundary edge* at s , which is an edge that is incident to only one vertex – and exits at t – perhaps also along some boundary edge – and at any other vertex u , the total incoming flow (flow on edges with $\theta(v, u) > 0$) equals the total outgoing flow (flow on edges with $\theta(u, v) > 0$). The “probability flows” in Section 2 are examples of flows from the entrance to the (sometimes more than one) terminal(s). The effective resistance between s and t is defined:

$$\mathcal{R}_{s,t}(G) = \min_{\theta} \sum_{e \in E} \frac{\theta(e)^2}{w_e}$$

where the minimization runs over unit *st*-flows, and if $e = \{u, v\}$, $\theta(e)^2 = \theta(u, v)^2 = \theta(v, u)^2$. The effective resistance is so named because it is the resistance across s and t if an appropriate potential difference is applied, in a network of electrical resistors, one for each edge of G , with conductances given by the edge weights. This beautiful theory is explained in [DS84]. Such electrical quantities have long-known connections to the theory of random walks, one of which is the following. Let

$\mathcal{H}_{s,t}(G)$ be the expected number of steps needed by a random walk on G , starting in s , to reach t for the first time. Then the following remarkable fact is due to [CRR⁺96]:

$$\mathcal{H}_{s,t}(G) + \mathcal{H}_{t,s}(G) = 2\mathcal{W}(G)\mathcal{R}_{s,t}(G). \quad (13)$$

A quantum walk is a kind of quantum algorithm based on a weighted undirected graph. We now describe a particular type of quantum walk, which is a special case of the *electric network quantum walk framework* [Bel13]. Consider a graph G_x (depending in some way on an input x) with distinct vertices $s, t \in V$ that we will refer to as *entrance* and *terminal*. Suppose s is a known vertex, perhaps labelled by 0^k for some k , to which we will always append a dangling boundary edge; and t is an unknown vertex, but we can recognize it, and we want to check if it has a certain property. If t has the property – suppose this happens precisely when $f(x) = 1$ – we will also give t a dangling boundary edge, but otherwise we will not. Note that boundary edges are not really part of the graph G (they are not in E), but we can assign them weights. We let $w_{u,\emptyset}$ be the weight of the boundary edge incident to u , or 0 if there is no such edge. A simple case of this setup is when G is a line, as shown in Figure 5.

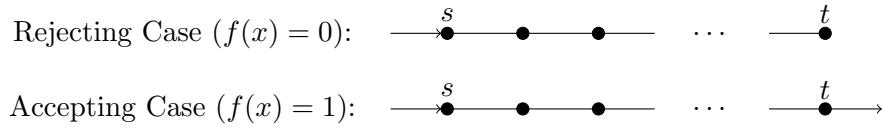


Figure 5: The cases distinguished by a quantum walk algorithm (special case of a line).

A classical random walk can distinguish these two cases in (worst-case) $\max_x \mathcal{H}_{s,t}(G_x)$ expected steps (which depends on the graph, but even when the graph is a line, it depends on the edge weights).

A quantum algorithm can distinguish these two cases in the following number of steps [Bel13]:

$$\sqrt{\max_{x:f(x)=0} \mathcal{W}(G_x) \cdot \max_{x:f(x)=1} \mathcal{R}_{s,t}(G_x)}. \quad (14)$$

An important question is, what is meant here by step? What we mean is the quantum analogue of a classical random walk step, which is to be able to generate, for any vertex u , a superposition with amplitude proportional to $\sqrt{w_{u,v}}$ on the edge going out from u towards v , which we write mathematically as

$$\sum_{v \in V} \sqrt{w_{u,v}} \mathbf{e}_{u,v} + \sqrt{w_{u,\emptyset}} \mathbf{e}_{u,\emptyset}. \quad (15)$$

Above $\mathbf{e}_{u,v}$ is just a vector with a 1 in the (u,v) -position, and 0 everywhere else. A quantum operation that generates superpositions proportional to (15) can be used to implement a quantum operation that we call the *walk operator*, and the walk operator is a transducer that decides between the accepting and rejecting cases from Figure 5 – that is, it decides the function f – with *no perturbation*, and with complexity given by the expression in (14) (see [BJY24, Section 6]). That is, it's a *perfect* transducer.

Let us consider the complexity in (14), and compare¹³ it to the classical complexity

¹³The classical complexity counts the number of classical walk steps, or classical samples of an outgoing edge, whereas the quantum complexity counts the number of quantum walk steps, but for the purposes of this article, we will assume they are approximately the same difficulty to implement.

$\max_x \mathcal{H}_{s,t}(G_x)$. Note that (14) is upper bounded by:

$$\sqrt{\max_x \mathcal{W}(G_x) \cdot \mathcal{R}_{s,t}(G_x)}$$

which is equal to

$$\sqrt{\max_x \frac{1}{2}(H_{s,t}(G_x) + H_{t,s}(G_x))},$$

by (13). This suggests that we get at best a squareroot speedup over the classical complexity. However, we will see that we can do better by maximizing separately over the two product terms.

Simple purification So returning to our toy example, suppose you have a quantum subroutine that outputs a superposition $\sqrt{p_0}\mathbf{e}_0 + \sqrt{1-p_0}\mathbf{e}_1 = \sqrt{p_0(x)}\mathbf{e}_0 + \sqrt{1-p_0(x)}\mathbf{e}_1$, the quantum version of a biased coin flip, satisfying one of the conditions in (11). Let f be the function that takes value $f(x) = 0$ precisely when $p_0(x) \geq 1 - \varepsilon$ (the rejecting case), and $f(x) = 1$ precisely when $p_0(x) \leq \varepsilon$ (the accepting case).

Let G be the line graph of length D , with vertices labelled $s = v_1, \dots, v_D = t$, edges labelled $1, \dots, D-1$, entrance boundary edge (into s) labelled 0; terminal boundary edge (out of t) labelled D ; and edge weights to be specified momentarily. With one call to the subroutine, we can, for any ℓ , generate the superposition

$$\sqrt{p_0}\mathbf{e}_\ell + \sqrt{1-p_0}\mathbf{e}_{\ell+1}, \quad (16)$$

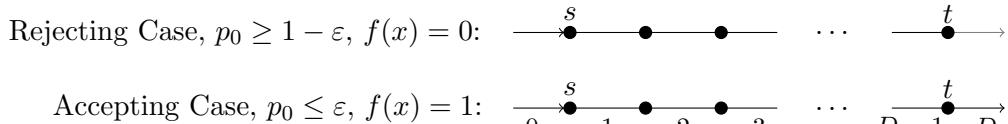
the quantum analog of sampling a bit, and adding ℓ to the sampled bit. Since the state in (16) is proportional to:

$$\left(\frac{1-p_0}{p_0}\right)^{\ell/2} \mathbf{e}_\ell + \left(\frac{1-p_0}{p_0}\right)^{(\ell+1)/2} \mathbf{e}_{\ell+1},$$

this allows us to implement a quantum walk on G with edge weights set to

$$w_\ell = \left(\frac{1-p_0}{p_0}\right)^\ell.$$

Now we want to distinguish between two cases:



In both cases, there is a boundary edge coming out of $t = v_D$, but in the rejecting case, the weight of this edge is

$$w_D = \left(\frac{1-p_0}{p_0}\right)^D \leq \left(\frac{\varepsilon}{1-\varepsilon}\right)^D = 2^{-\Theta(D)},$$

so it's almost like the edge is not there. It can be formalized that while a quantum walk of the form in Figure 5 is a perfect transducer for the associated decision problem (rejecting case vs. accepting case), the walk we are now describing is a *perturbed* transducer for the associated decision problem, with perturbation that can be made arbitrarily small by increasing D .

Let us now see that this transducer has $O(1)$ complexity, independent of D . I haven't defined the complexity of a transducer, but in the case of a quantum walk, it's just the expression in (14). So we need to upper bound the total weight of the graph in the rejecting case, and the effective resistance in the accepting case.

Rejecting case: Suppose $p_0 \geq 1 - \varepsilon$. We have:

$$\mathcal{W}(G_x) = \sum_{e \in E(G)} w_e = \sum_{\ell=1}^{D-1} \left(\frac{1-p_0}{p_0} \right)^\ell \leq \sum_{\ell=1}^{D-1} \left(\frac{\varepsilon}{1-\varepsilon} \right)^\ell \leq \frac{1}{1 - \frac{\varepsilon}{1-\varepsilon}} = O(1). \quad (17)$$

Accepting case: Suppose $p_0 \leq \varepsilon$. Since the graph is a path, there is only one possibility for a flow: $\theta(v_\ell, v_{\ell+1}) = 1$ on all edges ℓ between $s = v_1$ and $t = v_D$. Thus:

$$\mathcal{R}_{s,t}(G_x) \leq \sum_{e \in E(G)} \frac{\theta(e)^2}{w_e} = \sum_{\ell=1}^{D-1} \frac{1}{w_\ell} = \sum_{\ell=1}^{D-1} \left(\frac{p_0}{1-p_0} \right)^\ell \leq \sum_{\ell=1}^{D-1} \left(\frac{\varepsilon}{1-\varepsilon} \right)^\ell = O(1). \quad (18)$$

Complexity Together (17) and (18) imply that

$$\sqrt{\max_{x: p_0(x) \geq 1-\varepsilon} \mathcal{W}(G_x) \cdot \max_{x: p_0(x) \leq \varepsilon} \mathcal{R}_{s,t}(G_x)} = O(1).$$

As I claimed that the left-hand side is the complexity of the perturbed transducer, we see that it is constant, even when D is arbitrarily large.

Note that in the rejecting case, since $p_0 > 1 - p_0$, the effective resistance blows up to $2^{\Theta(D)}$, but fortunately, we only care about the resistance in the accepting case. Similarly, in the accepting case, since $1 - p_0 > p_0$, the total weight blows up to $2^{\Theta(D)}$, but we only care about the total weight in the rejecting case.

5 Final Thoughts

While efficient methods of composition are certainly useful in practice, I believe there is also something we can learn about the power of quantum algorithms relative to classical algorithms by comparing their different composition capabilities. The picture of what that is is not yet fully clear. However, I believe we also have something to learn about the right way of abstracting programs for future quantum computers.

Quantum algorithms are not classical algorithms, and we have probably been trying to force them unnaturally into the shape of classical algorithms for too long. At the same time, the temptation to do so is understandable. We understand classical algorithms pretty well, relative to quantum algorithms, where things seem very mysterious, dramatic speedups are few and far between, and we don't really understand for the most part why fast quantum algorithms are fast (at least, not well enough to produce more fast quantum algorithms). So a model that keeps some classical intuition is desirable. My two cents is that we've been trying to hold onto the wrong classical intuition – that of circuits – and should borrow more heavily from the intuition of randomized algorithms, to understand both the similarities and differences between quantum and classical algorithms.

Transducers give us a way to reason about quantum algorithms that seems more natural for them as *quantum algorithms*, rather than some quantum version of a *classical algorithm*. If we think about the special case of quantum walks, which are a sort of easy-to-visualize transducer, we have seen how a quantum walk lets us model a quantum algorithm, even one with error (which is most of them), as *perfect* or at least *arbitrarily close to perfect* objects that easily compose. I would like to humbly put forth that perhaps transducers are the more quantum model for reasoning about quantum algorithms that we have been missing.

Acknowledgements I would like to thank Aleksandrs Belovs and Ronald de Wolf for helpful comments and suggestions on a draft of this article.

References

- [AB09] Sanjeev Arora and Boaz Barak. *Computational Complexity: A Modern Approach*. Cambridge University Press, 2009.
- [BB19] Eric Blais and Joshua Brody. Optimal separation and strong direct sum for randomized query complexity. In *Proceedings of the 34th IEEE Conference on Computational Complexity (CCC)*, pages 1–17, 2019.
- [Bel13] Aleksandrs Belovs. Quantum walks and electric networks. arXiv: [1302.3143](https://arxiv.org/abs/1302.3143), 2013.
- [BJY24] Aleksandrs Belovs, Stacey Jeffery, and Duyal Yolcu. Taming quantum time complexity. *Quantum*, 8(1444), 2024. arXiv: [2311.15873](https://arxiv.org/abs/2311.15873)
- [BdW03] Harry Buhrman and Ronald de Wolf. Quantum zero-error algorithms cannot be composed. *Information Processing Letters*, 87(2):79–84, 2003. arXiv: [quant-ph/0211029](https://arxiv.org/abs/quant-ph/0211029)
- [BY23] Aleksandrs Belovs and Duyal Yolcu. One-way ticket to Las Vegas and the quantum adversary. arXiv: [2301.02003](https://arxiv.org/abs/2301.02003), 2023.
- [CRR⁺96] Ashok K. Chandra, Prabhakar Raghavan, Walter L. Ruzzo, Roman Smolensky, and Prasoon Tiwari. The electrical resistance of a graph captures its commute and cover times. *Computational Complexity*, 6(4):312–340, 1996.
- [DJ92] David Deutsch and Richard Jozsa. Rapid solutions of problems by quantum computation. *Proceedings of the Royal Society of London A*, 1907:553–558, 1992.
- [DS84] Peter G. Doyle and J. Laurie Snell. *Random walks and electric networks*. Mathematical Association of America, 1984. arXiv: [math/0001057](https://arxiv.org/abs/math/0001057)
- [IZ89] Russell Impagliazzo and David Zuckerman. How to recycle random bits. In *Proceedings of the 30th IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 248–253, 1989.
- [Jef22] Stacey Jeffery. Quantum subroutine composition. arXiv: [2209.14146](https://arxiv.org/abs/2209.14146), 2022.
- [Rei09] Ben W. Reichardt. Span programs and quantum query complexity: The general adversary bound is nearly tight for every boolean function. In *Proceedings of the 50th IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 544–551, 2009.