41076: Methods in Quantum Computing

'Quantum Algorithms' Module

Maria Kieferova based on materials from Min-Hsiu Hsieh
Centre for Quantum Software & Information, Faculty of Engineering and Information Technology, University of Technology Sydney

Abstract

Contents to be covered in these two lectures are

1. Complexity of an algorithm
2. Phase kickback
3. Hadamard transform
4. Fourier Transform Algorithm
5. Phase Estimation
6. Hamiltonian simulation and ground state preparation
7. Grover search and its optimality

1 What makes a good algorithm

To understand how to analyze and design quantum algorithms, we first need to talk about general concepts in algorithm design. An algorithm can be thought of as a recipe for a computer to compute a function, i.e. compute an output for each input from a set of allowed inputs. We might require that the algorithm always produces the correct output. We say that an algorithm is **sound** if it never returns a wrong solution, say an incorrectly sorted list, a wrong solution to arithmetic operations or a sub-optimal solution. An algorithm is **complete** if it gives an answer for any input. An incomplete algorithm for cat/dog classification might not provide a label if the input is neither a cat nor a dog picture.

There are notable types of algorithms that are not complete or sound:

- Many **probabilistic**/quantum algorithms give the correct answer with a high probability (say, in at least 2/3 cases) but might fail. The probability of success can be boosted by repeated runs. Examples included adiabatic phases estimation, Grover search, adiabatic algorithms (with long evolution time), random walk search ...
- **Approximation algorithms** are useful for optimization problems. Optimization problems require finding maximum/minimum or a function on a given domain. An approximation algorithm does not aim to achieve the best solution but produces a solution that is "close". An example for is Goemans-Williamson algorithm that is guaranteed to achieve approximation

ratio (defined as $\frac{\text{algorithm's score}}{\text{optimal score}}$) of 0.868 for MAX-CUT. Some algorithms come with rigorous guarantees for the quality of the approximation, others, such as simulated annealing, are guaranteed to produce the optimal solution if run indefinitely.

- Some algorithms perform well only on a subset of inputs. An example would be machine algorithms, for example, ML approaches to k-means clustering, that would be inefficient on general problems but perform well on problems with some structure. A formal way to restrict the set of inputs is to define a **promise problem** which guarantees additional properties of the input (as in the Deutsch-Jozsa algorithm).

Heuristic algorithms are a broad class of algorithms that trade speed for completeness, optimality or precision that might be based on rigorous underlying theory or rule-of-thumb insights.

To analyze an algorithm, we consider the resources an algorithm needs. We call these considerations **computational complexity** The most common consideration is the time the algorithm needs to compute the output. The time is proportional to the number of operations the computer needs to make during computation. We consider how the time needed grows with the **size of the input** . This can be the number of bits of a number, the size of an array or a data set or the number of particles in a quantum system. When talking about the dependence of the time on the size of the input, we call this measure the **time complexity** or often only **complexity**.

We sometimes consider are space needed for computation known as space complexity. Optimizing algorithms for space is needed when one is dealing with limited memory and a limited number of qubits. If we have a computational model that has the ability to execute multiple operations in parallel, such as a node with many processors/GPU-s, we might consider the depth of an algorithm instead. In practice, one might consider the dollar cost of computation which would depend on the computational time as well as particularities of a machine used.

Let say that we agreed what is the metric that matters to us. How would we then evaluate the performance of our algorithms? There are several approaches

- In theoretical computer science, the most common approach is computing the **asymptotic complexity**. This is an upper bound on the amount of time one would need as a function of input size in the limit of input size going to infinity. Computing the tight bound is generally complicated but a good estimate in the big-O notation can be obtained for many algorithms.

- When an algorithm is too complicated to analyze rigorously or its performance can vary, it is common to estimate its **performance on benchmarks**. An example would be standard machine-learning or SAT competition. It is important to choose benchmarks similar to the problems of practical importance and similar size. Choosing a benchmark that is too small or simple can give a false sense of success that would not generalize to larger instances.

- Argue with everyone at conferences and on Twitter that your algorithm is the best. This is common but not recommended.

Once we know the complexity of our algorithm, we can ask how good it is. The strongest type of evidence would be an argument that it is not possible to construct an asymptotically better algorithm, possibly subject to some complexity-theoretic assumptions. An example of such algorithms is $O(n \log n)$ complexity sorts and Grover search. For these examples, the performance of the algorithms matches the theoretical lower bound on what could be possible. An argument that is still strong is showing that your algorithm is the fastest existing algorithm. Such an example

notation	name	example
$O(1)$	constant	computing $(-1)^n$
$O(\log x)$	logarithmic	binary search
$O(x)$	linear	classical unstructured search
$O(x^2)$	quadratic	bubble sort, elementary-school multiplication, Dijkstra
$O(x^c)$	polynomial	all of the above, linear programming, primality testing, shortest path on a graph
$O(2^x)$	exponential	travelling salesman, generalized checkers
$O(x!)$	factorial	brute force try all possibilities algorithm

Figure 1: Common functions characterizing asymptotic complexity.

is Shor's factoring algorithm which is asymptotically faster (polynomial vs. exponential). Since factoring has been studied for centuries and Shor's algorithm is the first efficient algorithm this constitutes a major achievement. On the other hand, the QAOA algorithm has achieved the best approximation ratio for the Max-3XOR problem, only to be outperformed a few weeks later. Of course, you might not care about having the best algorithm according to the community. You might only care about solving a particular problem, say computing properties of a complex molecule or devising a super-secret strategy for trading derivatives and you only care that your algorithm delivers a solution up to your (and your boss's standard).

1.1 Big-O notation

Perhaps the most important measure of an algorithm is how its complexity grows as the size of the input. Unfortunately, estimating the exact resources is almost always impossible - the exact number of gates will depend on the gate set physical properties of a chip. To be able to devise and compare algorithms that can run on many different chips, we focus only on the type of the growth rate expressed using the big-O notation that characterizes the behaviour of complexity for the limit $x \to \infty$.

Let $f(x) : \mathcal{R} \to \mathcal{R}$ be the exact complexity of an algorithm. The one can write

$$f(x) = O(g(x)) \tag{1}$$

if there exist numbers $x_0 \in \mathcal{R}$ and $M \in \mathcal{R}_+$ such that

$$|f(x)| \leq Mg(x) \tag{2}$$

for all $x > x_0$. When estimating the big-O complexity, it is useful to keep in mind the following properties:

$$f_1(x) = O(g_1(x)) \text{ and } f_2(x) = O(g_2(x)) \implies f_1(x)f_2(x) = O(g_1(x)g_2(x)) \tag{3}$$
$$f_1(x) = O(g_1(x)) \implies f_1(x)f_2(x) = O(g_1(x)f_2(x)) \tag{4}$$
$$f_1(x) = O(g_1(x)) \text{ and } f_2(x) = O(g_2(x)) \implies f_1(x) + f_2(x) = O(\max(g_1(x), g_2(x))) \tag{5}$$
$$f_1(x) = O(g_1(x)) \implies const. \cdot f_1(x) = O(g_1(x)) \tag{6}$$
$$f_1(x) = O(g_1(x)) \implies const. + f_1(x) = O(g_1(x)) \tag{7}$$

Exercise 1. *Practise the big-O formalism.*

Figure 2: Reversible oracle for a function f.

- *Is* $2^{n+1} = O(2^n)$*?*
- *Is* $2^{2n} = O(2^n)$*?*
- *Compute the asymptotic upper bound on* $2^{n+1} + 2^{2n}$*.*

We say that an algorithm is efficient if its complexity is polynomial in input size as opposed to exponential in any parameter. The class of all problems solvable in polynomial time on a classical computer is known as **P** and **BQP** on a quantum computer.

1.2 Oracles

Apart from gates, it is possible to include oracles (sometimes also called black boxes) in computation. An oracle is an abstract operation that can perform a given computation in a unit of time. We refer to the number of queries to the oracle as the query complexity. There are several reasons why one wants to include oracles in an algorithm. An oracle can represent a separate computational primitive whose implementation can vary. An example of such oracles are functions that compute matrix elements of a Hamiltonian in Hamiltonian simulation.

An oracle can be given as a part of the definition of the problem. In this case, one can have oracular access to a function, and our goal is to determine some properties of this function as in Deutsch-Joza or Grover's algorithms. Finally, oracles are often used in the theory of computational complexity to quantify the difficulty of tasks. One can construct a hierarchy of computational difficulty with respect to more and more powerful oracles.

2 Thinking about quantum algorithms

We often (but not always) think about quantum algorithms in terms of quantum circuits. Broadly speaking, a quantum algorithm is a description of how to build a quantum circuit that produces the desired output from the input for each size of the input. Thinking in terms of quantum circuits is often tedious for more advanced algorithms and people often describe quantum algorithms in terms of frameworks such as quantum walks or simulating sparse matrices. These concepts will be explored with more depth in UTS class Quantum algorithms.

If we are given a quantum circuit, we can easily compute its various cost metrics such as the number of operations and depth. Calculating an asymptotic complexity of a quantum algorithm is similar to calculating the complexity of classical algorithms.

We learned in lecture one that any classical computation can be realized reversibly using Toffoli gates. Since Toffoli gates can be (in principle) implemented on quantum computers, it means that any classical computation can be implemented on a quantum computer as a quantum algorithm. However, doing so would not produce any speedup - it would be likely slower (by a constant factor) a more expensive. However, this highlights the point that an **existence of a quantum algorithm does not imply a speedup**, which refutes the common misconception that quantum computing

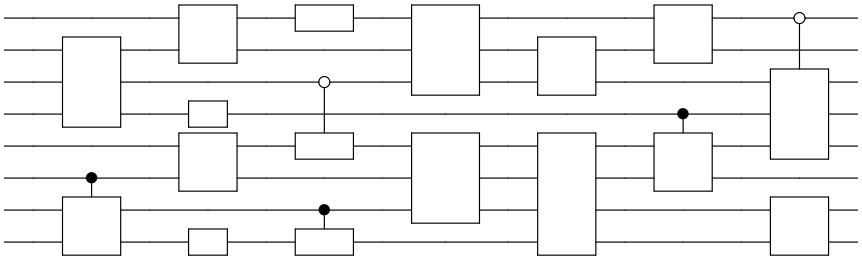


Figure 3: A quantum circuit consists of wires (qubits, represented as horizontal lines) and gates (unitary operations, represented as boxes). The space cost of this circuit is 8, the time cost is 17, and the depth is 7.

can exponentially speed up any computation. We have examples of algorithms, for example sorting, that provably do not allow asymptotic quantum speedup.

3 Phase kickback

Exercise 2. *Compute:*

- *CNOT* $|b\rangle\frac{|0\rangle-|1\rangle}{\sqrt{2}}$ *for* $b \in \{0,1\}$. *Hint: It will be useful to write* $1=(-1)^0$ *and* $-1=(-1)^1$.
- *CNOT* $\frac{|0\rangle+|1\rangle}{\sqrt{2}}\frac{|0\rangle-|1\rangle}{\sqrt{2}}$.

We can now see that if the target qubit is an eigenstate of X, CNOT will affect the control qubit instead of the target one. A phase kickback is a clever trick that extends this observation from a CNOT to an arbitrarily controlled gate. Let us take the oracle from 2 and apply it on $|y\rangle = \frac{|0\rangle-|1\rangle}{\sqrt{2}}$. We will get:

$$U_f|x\rangle\frac{|0\rangle-|1\rangle}{\sqrt{2}} = \frac{U_f|x\rangle|0\rangle - U_f|x\rangle|1\rangle}{\sqrt{2}} \tag{8}$$

$$= \frac{|x\rangle|0\oplus f(x)\rangle - |x\rangle|1\oplus f(x)\rangle}{\sqrt{2}}. \tag{9}$$

Without knowing the value of x, we still know that $f(x)$ will be either 0 or 1.

Let us consider what happens for the options. For $f(x)=0$ we get

$$\frac{|x\rangle|0\oplus 0\rangle - |x\rangle|1\oplus 0\rangle}{\sqrt{2}} = \frac{|0\rangle-|1\rangle}{\sqrt{2}} \tag{10}$$

and similarly $f(x)=1$

$$\frac{|x\rangle|0\oplus 1\rangle - |x\rangle|1\oplus 1\rangle}{\sqrt{2}} = -\frac{|0\rangle-|1\rangle}{\sqrt{2}} \tag{11}$$

will result in a global phase. We can both outcomes in a compact way

$$(-1)^{f(x)}|x\rangle\frac{|0\rangle-|1\rangle}{\sqrt{2}} \tag{12}$$

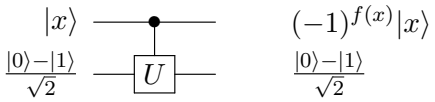


Figure 4: Phase kickback

.

This is known as the phase-kickback trick - it allows us to turn the output of the function f into a (global) phase. To see why it can be useful, consider $|x\rangle = a|0\rangle + b|1\rangle$ to be an arbitrary superposition. Using (12) we can compute

$$U(a|0\rangle + b|1\rangle)\frac{|0\rangle - |1\rangle}{\sqrt{2}} = (-1)^{f(0)}a|0\rangle\frac{|0\rangle - |1\rangle}{\sqrt{2}} + (-1)^{f(1)}b|1\rangle\frac{|0\rangle - |1\rangle}{\sqrt{2}} \tag{13}$$

$$= \Big((-1)^{f(0)}a|0\rangle + (-1)^{f(1)}b|1\rangle\Big)\left(\frac{|0\rangle - |1\rangle}{\sqrt{2}}\right). \tag{14}$$

The action of the oracle is now entirely encoded in the phase of the first qubit and there is no entanglement between registers.

An application of the phase kickback is Deutsch's algorithm.

Exercise 3 (Deutsch's problem)**.** *Consider an unknown function $f(x)$ given through a coherent oracle. Decide whether the function is constant $f(0) = f(1)$ or balanced $f(0) \neq f(1)$.*

- *Use the phase kickback trick for $x = 0$ and $x = 1$ on Deutsch's problem. What is the outcome for a constant and a balance function?*
- *Now consider $|0\rangle = \frac{|0\rangle+|1\rangle}{\sqrt{2}}$. What gate do you need to apply on the control register to read out the solution in computational basis?*

4 Hadamard transform

We saw that application of Hadamard was crucial in the previous algorithm. Applying Hadamard on evry qubit is known as **Hadamard transform**. In the simplest case, we can apply Hadamards on a number of qubits in each in state $|0\rangle$.

$$(H|0\rangle)^{\otimes n} = \left(\frac{|0\rangle + |1\rangle}{\sqrt{2}}\right)^{\otimes n} = 2^{-n/2}\sum_{i=0}^{2^n-1}|i\rangle. \tag{15}$$

Exercise 4. *Expand $\left(\frac{|0\rangle+|1\rangle}{\sqrt{2}}\right)^{\otimes n}$ to convince yourself that*

$$\left(\frac{|0\rangle + |1\rangle}{\sqrt{2}}\right)^{\otimes n} = 2^{-n/2}\sum_{i=0}^{2^n-1}|i\rangle. \tag{16}$$

How would you formally prove it?

We see that applying Hadamards will create a uniform superposition over all strings. What happens when we apply Hadamard on an arbitrary input? First, notice that we can write an action of a single Hadamard as

$$H|x_i\rangle = \frac{|0\rangle + (-1)^x|1\rangle}{\sqrt{2}} = \sum_{y=0}^{1} \frac{(-1)^{x_i \cdot y}|y\rangle}{\sqrt{2}} \tag{17}$$

where $x_i \in \{0,1\}$. Let us now take $x = x_0 x_1 \dots x_{n-1}$ and apply Hadamard on this state. We obtain

$$H^{\otimes n}|x\rangle = \frac{1}{\sqrt{2^n}} \sum_y (-1)^{x \cdot y}|y\rangle, \tag{18}$$

where $x \cdot y = \sum_i x_i y_i$.

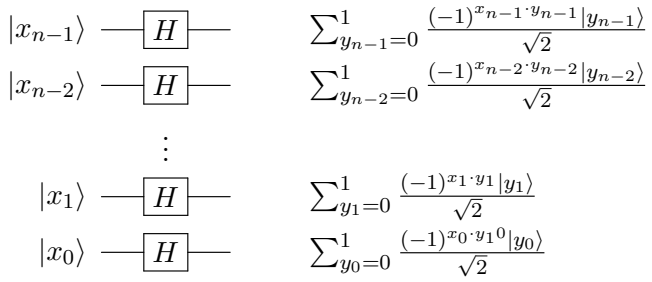


Figure 5: Hadamard transform

Exercise 5. *Show that Hadamard transform is self-inverse, i.e. $H^{\otimes n} H^{\otimes n} = \mathbb{I}^{\otimes n}$. While this might be obvious for each individual qubit, it is worth expanding the expression.*

An application of phase kickback and Hadamard transform is the Deutsch-Jozsa algorithm.

Problem 6 (Deutsch-Jozsa)**.** *Assume an oracle that implements a function $f : \{0,1\}^n \to \{0,1\}$ such that we are promised*

1. *(constant) all inputs give the same output, or*
2. *(balanced) half the inputs give '0' and the other half give '1'.*

The goal is to find out whether $f(x)$ is constant or balanced.

Before going to the quantum algorithm for solving this problem, let us first look at the possible classical solutions. In fact, the classical strategy is very simple. If we want 100% accuracy, in the worse case, one has to *query* at least $\frac{N}{2} + 1$ bits in $\boldsymbol{x}$.

It seemed quite counter-intuitive in the beginning that there is a quantum algorithm, proposed by Deutsch and Jozsa, which can produce the correct answer with just a single use of quantum oracle (i.e., quantum unitary). The quantum oracle queries the bit string $\boldsymbol{x}$ only once; hence the Deutsch-Jozsa algorithm has the exponential saving, compared with the classical strategy, in the number of queries to $\boldsymbol{x}$.

A quantum algorithm for Deutsch-Jozsa problem expands on Deutsch's algorithm by replacing a single Hadamard with the Hadamard transform.

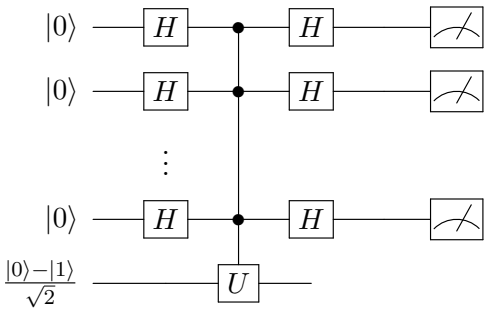


Figure 6: Deutsch-Jozsa algorithm

We will again use a quantum oracle

$$O_{\boldsymbol{x}} : |i\rangle|b\rangle \to |i\rangle \otimes |b \oplus x_i\rangle \tag{19}$$

where $i \in [N]$, $b \in \mathbb{Z}_2$ and $\oplus$ is the binary addition.

Let us see what the algorithm in 6 does step-by-step.

Starting with zeros on the first n registers and $\frac{|0\rangle-|1\rangle}{\sqrt{2}}$ on the last qubit, we apply Hadamards on the first n qubits and end up with a uniform superposition over all strings

$$H^{\otimes n}|0\rangle^{\otimes n}\frac{|0\rangle-|1\rangle}{\sqrt{2}} = 2^{-n/2}\sum_{i=0}^{2^n-1}|i\rangle\frac{|0\rangle-|1\rangle}{\sqrt{2}}. \tag{20}$$

Next, we apply the oracle

$$U2^{-n/2}\sum_{i=0}^{2^n-1}|i\rangle\frac{|0\rangle-|1\rangle}{\sqrt{2}} = 2^{-n/2}\sum_{i=0}^{2^n-1}|i\rangle\frac{|0\oplus f(i)\rangle-|1\oplus f(i)\rangle}{\sqrt{2}} = 2^{-n/2}\sum_{i=0}^{2^n-1}(-1)^{f(i)}|i\rangle\frac{|0\rangle-|1\rangle}{\sqrt{2}} \tag{21}$$

where we realized the insight from previous section that the oracle will perform a phase shift. After the second Hadamard transform we get

$$2^{-n/2}H^{\otimes n}\sum_{i=0}^{2^n-1}(-1)^{f(i)}|i\rangle\frac{|0\rangle-|1\rangle}{\sqrt{2}} = 2^{-n}\sum_{j=0}^{2^n-1}\sum_{i=0}^{2^n-1}(-1)^{f(i)}(-1)^{i\cdot j}|j\rangle\frac{|0\rangle-|1\rangle}{\sqrt{2}}. \tag{22}$$

The algorithm is concluded by measuring the first n registers. Let us now see what is the resulting state if f is constant or balanced. If f is constant, $f(i) = f$ independent of i. That would simply (22) to

$$2^{-n}(-1)^f\sum_{j=0}^{2^n-1}\sum_{i=0}^{2^n-1}(-1)^{i\cdot j}|j\rangle\frac{|0\rangle-|1\rangle}{\sqrt{2}} = (-1)^f|0\rangle^{\otimes n}\frac{|0\rangle-|1\rangle}{\sqrt{2}}, \tag{23}$$

because the oracle will only contribute an overall phase and Hadamard transform is self inverse. Thus, if the function is constant, we will measure all zeros. If the function is balanced, we can divide the inputs into those with output 0 and the remaining half with output 1

$$2^{-n}\left(\sum_{i,f(i)=0}(-1)^0\sum_{j=0}^{2^n-1}(-1)^{i\cdot j}|j\rangle + \sum_{i,f(i)=1}\sum_{j=0}^{2^n-1}(-1)^1(-1)^{i\cdot j}|j\rangle\right)\frac{|0\rangle-|1\rangle}{\sqrt{2}}. \tag{24}$$

We can see that this state has 0 overlap with $|0\rangle^{\otimes n}$. Thus, if we measure all zeros, the function is constant and otherwise, it is balanced.

Exercise 7. *Show that the output from a balanced function will be orthogonal to all the zero state.*

Before ending this section, I would like to emphasize that if we allow some small error probability in deciding whether $\boldsymbol{x}$ is constant or balance in the classical setting, the quantum advantage of the Deutsch-Jozsa algorithm will disappear completely.

5 Quantum Fourier Transform

The discrete Fourier transform of a set $\{x_0, \cdots, x_{N-1}\}$ of N elements is defined as

$$X_k = \frac{1}{\sqrt{N}} \sum_{j=0}^{N-1} x_j e^{\frac{i2\pi}{N} jk}. \tag{25}$$

In the following, we will denote $\omega_N := e^{\frac{i2\pi}{N}}$ the N-th root of unity. Let $U_{\rm F}$ be the square matrix whose (i,j)-th element is $\frac{1}{\sqrt{N}}\omega_N^{ij}$.

Exercise 8. *Show that $U_{\rm F}$ is unitary.*

The definition of Fourier transform in Eq. (25) can be extended to the quantum setting

$$|\Psi_k\rangle := U_{\rm F}|k\rangle = \frac{1}{\sqrt{N}} \sum_{j=0}^{N-1} e^{\frac{i2\pi}{N} jk} |j\rangle. \tag{26}$$

It is crucial to note that the state $|\Psi_k\rangle$ is a product state when $N = 2^n$ and can be written as

$$\frac{1}{\sqrt{N}} \sum_{j=0}^{N-1} e^{i2\pi jk/2^n} |j\rangle = \bigotimes_{\ell=1}^{n} \frac{1}{\sqrt{2}} \Big(|0\rangle + e^{i2\pi k/2^\ell} |1\rangle \Big),$$

.

To show that, consider the binary representation of $k \equiv (k_1, \cdots, k_n) \in [N]$, where k_1 is the most significant bit, i.e.,

$$k = k_1 2^{n-1} + k_2 2^{n-2} + \cdots + k_n 2^0, \tag{27}$$

and we write

$$k/2^n = 0.k_1 k_2 \cdots k_n = \sum_{\ell=1}^{n} k_\ell 2^{-\ell}.$$

Take for example, $k = 5 = (1, 0, 1)$ and $n = 3$, therefore $5/8 = 0.101$. Thus

$$\begin{aligned}
|\Psi_k\rangle &= \frac{1}{\sqrt{N}} \sum_{j=0}^{N-1} e^{i2\pi jk/2^n} |j\rangle \\
&= \frac{1}{\sqrt{2^n}} \sum_{j_1=0}^{1} \cdots \sum_{j_n=0}^{1} e^{i2\pi \left(\sum_{\ell=1}^{n} j_\ell/2^{-\ell}\right)k} |j_1, j_2, \cdots, j_n\rangle \\
&= \frac{1}{\sqrt{2^n}} \sum_{j_1=0}^{1} \cdots \sum_{j_n=0}^{1} \bigotimes_{\ell=1}^{n} e^{i2\pi j_\ell k/2^{-\ell}} |j_\ell\rangle \\
&= \frac{1}{\sqrt{2^n}} \bigotimes_{\ell=1}^{n} \left[\sum_{j_\ell=0}^{1} e^{i2\pi j_\ell k/2^{-\ell}} |j_\ell\rangle \right] \\
&= \bigotimes_{\ell=1}^{n} \frac{1}{\sqrt{2}} \Big(|0\rangle + e^{i2\pi k/2^\ell} |1\rangle \Big), \\
&:= \bigotimes_{\ell=1}^{n} |\Phi_\ell\rangle, \qquad (28)
\end{aligned}$$

where in the last line we denote

$$|\Phi_\ell\rangle := \frac{1}{\sqrt{2}} \Big(|0\rangle + e^{i2\pi k/2^\ell} |1\rangle \Big), \qquad (29)$$

and $k/2^\ell = 0.k_{n-\ell-1} \cdots k_n$, the ℓ least significant bits of k because the first $n - \ell$ most significant bits of k have no effect on the value ($e^{i2\pi m} = 1$ for $m \in \mathbb{N}$).

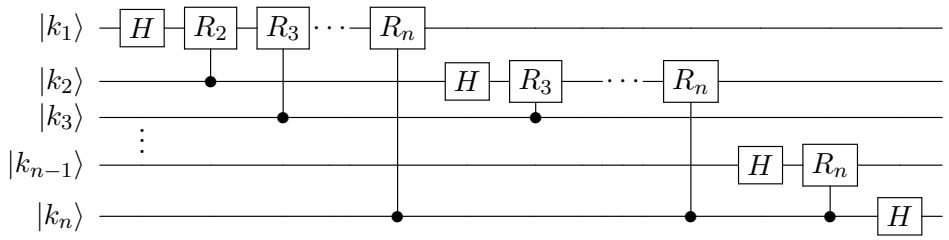


Figure 7: Circuit for quantum Fourier transform.

The implementation of Eq. (28) is given in Figure 7, where

$$R_m = \begin{pmatrix} 1 & 0 \\ 0 & e^{i2\pi/2^m} \end{pmatrix}. \qquad (30)$$

At the first step, the quantum state is transformed into

$$|k_1\rangle \otimes \cdots \otimes |k_n\rangle \to \frac{1}{\sqrt{2}} \Big(|0\rangle + e^{i2\pi 0.k_1} |1\rangle \Big) \otimes |k_2\rangle \otimes \cdots \otimes |k_n\rangle, \qquad (31)$$

because

$$e^{i2\pi 0.k_1} = e^{i2\pi \frac{k_1}{2}} = \begin{cases} -1 & \text{when } k_1 = 1 \\ 1 & \text{when } k_1 = 0 \end{cases}. \qquad (32)$$

Next, we have

$$\frac{1}{\sqrt{2}}\Big(|0\rangle + e^{i2\pi 0.k_1k_2}|1\rangle\Big) \otimes |k_2\rangle \otimes \cdots \otimes |k_n\rangle, \tag{33}$$

because when $k_2 = 0$, the state in Eq. (33) is the same as Eq. (31), and when $k_2 = 1$, a phase of $e^{i2\pi/2^2}$ is applied. Following the same derivation, the state on the first qubit will become

$$\frac{1}{\sqrt{2}}\Big(|0\rangle + e^{i2\pi 0.k_1k_2\cdots k_n}|1\rangle\Big) \otimes |k_2\rangle \otimes \cdots \otimes |k_n\rangle. \tag{34}$$

On the second qubit we get after Hadamard

$$|k_2\rangle \to \frac{1}{\sqrt{2}}\Big(|0\rangle + e^{i2\pi 0.k_2}|1\rangle\Big)$$

and controlled rotations

$$|k_2\rangle \to \frac{1}{\sqrt{2}}\Big(|0\rangle + e^{i2\pi 0.k_2k_3\ldots k_n}|1\rangle\Big)$$

On all the qubits

$$2^{-n/2}\Big(|0\rangle + e^{2\pi i 0.k_n}|1\rangle\Big)\Big(|0\rangle + e^{2\pi i 0.k_{n-1}k_n}|1\rangle\Big)\ldots\Big(|0\rangle + e^{2\pi i 0.k_1k_2\ldots k_n}|1\rangle\Big)$$

We can see that the number of gates in Figure 7 is

$$n + (n-1) + \cdots + 1 = \frac{n(n+1)}{2}, \tag{35}$$

that is exponentially less than the classical fast Fourier transform which requires $O(n2^n)$ gates. (Exercise).

Exercise 9. *Show that on all zero input, the Hadamard transform and quantum Fourier transform produce the same output. Formally* $QFT|0\rangle^{\otimes n} = H^{\otimes n}|0\rangle^{\otimes n}$.

Unlike Hadamard transform, quantum Fourier transform is not self inverse. QFT^{-1} is defined as

$$|\Psi_k\rangle := U_{\mathrm{F}}^{-1}|k\rangle = \frac{1}{\sqrt{N}}\sum_{j=0}^{N-1} e^{-\frac{i2\pi}{N}jk}|j\rangle. \tag{36}$$

Inverse-QFT can be implemented by inverting the circuit, i.e. running it backwards with rotations in the negative direction. Thus, inverse-QFT has the same complexity as QFT.

6 Phase Estimation

In this section, we will introduce the quantum phase estimation protocol. The crucial component of the quantum phase estimation protocol is the quantum Fourier transform.

Problem 10. *Given a unitary* U *and its eigenvector* $|\nu\rangle$*, estimate the corresponding eigenvalue* $\lambda = e^{i2\pi\varphi}$.

Theorem 11. *The quantum phase estimation algorithm, illustrated in Figure 8, can estimate the value of* φ *to the additive error* ε *with high probability, using* $O(\log(\frac{1}{\varepsilon}))$ *qubits and* $O(\frac{1}{\varepsilon})$ *controlled-U operations.*

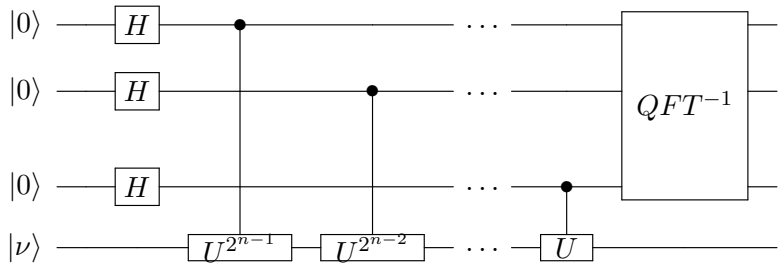


Figure 8: A circuit for phases estimation and eigenvalue estimation. The last block represents the inverse quantum Fourier transform.

Proof. Let $N = 2^n$. After Hadamard transform step, the overall state is

$$|\Psi_{t_1}\rangle = \left(\frac{1}{\sqrt{N}}\sum_{k=0}^{N-1}|k\rangle\right)\otimes|\nu\rangle. \tag{37}$$

Since $U|\nu\rangle = e^{i2\pi\varphi}|\nu\rangle$, we have

$$U^{2^j}|\nu\rangle = e^{i2\pi 2^j\varphi}|\nu\rangle. \tag{38}$$

Each controlled unitary will act as

$$c-U^{2^j}|k_i\rangle|\nu\rangle = e^{i2\pi 2^j k_i\varphi}|\nu\rangle, \tag{39}$$

where we again think of k in its binary representation $k = k_1 2^{n-1} + k_2 2^{n-2} + \cdots + k_n 2^0$. Note that all of the controlled-U operations commute. After application of the controlled unitaries, the state will be

$$|\Psi_{t_2}\rangle = \left(\frac{1}{\sqrt{N}}\sum_{k=0}^{N-1}e^{i2\pi\varphi Nk}|k\rangle\right)\otimes|\nu\rangle. \tag{40}$$

Lastly we apply the inverse quantum Fourier transform here denoted as U_F^{-1}

$$|\Psi_{t_3}\rangle = (U_{\mathrm{F}}^{-1}\otimes I)|\Psi_{t_2}\rangle = \left(\frac{1}{N}\sum_{y=0}^{N-1}\sum_{k=0}^{N-1}e^{\frac{i2\pi k}{N}(N\varphi-y)}|y\rangle\right)\otimes|\nu\rangle. \tag{41}$$

Finally, the probability of obtaining the outcome $a \in [N]$ is

$$\begin{aligned}
\Pr\{\text{Outcome } a\} &= \left|\langle a|\frac{1}{N}\sum_{y=0}^{N-1}\sum_{k=0}^{N-1}e^{\frac{i2\pi k}{N}(N\varphi-y)}|y\rangle\right|^2 &(42)\\
&= \left|\frac{1}{N}\sum_{k=0}^{N-1}e^{\frac{i2\pi k}{N}(N\varphi-a)}\right|^2 &(43)\\
&= \begin{cases}1 & N\varphi = a\\ \left|\frac{1}{N}\sum_{k=0}^{N-1}e^{\frac{i2\pi k}{N}\delta}\right|^2 & N\varphi - a = \delta \neq 0\end{cases}. &(44)
\end{aligned}$$

For the ideal case where $N\varphi = a$ is an integer, the estimation is exact.

When $N\varphi - a = \delta \neq 0$, we can show that the circuit will produce the correct outcome with high probability. □

7 Order finding and Shor's algorithm

Order finding is an application of QFT and phase estimation that is the crucial step of Shor's algorithm. We say that r is an order of a modulo N if it is the smallest positive integer that satisfies

$$a^r = 1(\text{mod } N). \tag{45}$$

where N is an integer.

Exercise 12. *Find the order of* 4 *modulo* 7*, i.e. find* r *such that* $4^r = 1(mod 7)$.

Generally, computing the order is difficult for large N - we do not have a classical algorithm that can perform order finding in time polynomial in the number of bits of N. In fact, factoring can be reduced into order finding. Let N be a large integer that we wish to find factors of. Assume that N is not even or of the form $n^k = N$ – these cases are easy to check and would allow us to find factors easily.

1. Randomly pick $1 < a < N$.
2. Use Euclid's algorithms to find the greatest common divisor $gcd(a, N)$. If it is larger than 1, we found a factor of N and stop.
3. Compute the period r such that $a^r = 1(\text{mod } N)$.
4. If r is odd or $a^{r/2} = N - 1(\text{mod } N)$, go back restart from step 1.
5. Otherwise, both $gcd(a^{r-1}\pm, N)$ give nontrivial factors of N.

Exercise 13. *Try this algorithm with* $N = 15$ *and* $a = 7$.

Shor's algorithm performs the steps above with order finding being the only quantum part of the algorithm. Order finding uses the circuit in Fig. 8 with

$$|z\rangle c - U^j|x\rangle = |z\rangle|a^{z \cdot j} x(\text{mod } N)\rangle. \tag{46}$$

This oracle can be implemented using modular exponentiation and each application has complexity $O(\log N^2)$. Since we need to apply it N times, the overall complexity of applying controlled-U operations is $O(\log^3 N)$. The other steps of the algorithm are the Hadamard transform with complexity $O(\log N)$ (we need $\log N$ qubit to encode N) and quantum Fourier transform with complexity $O(\log^2 N$. Thus, the overall complexity of the order-finding algorithm, as well as the quantum part of Shor's algorithm, is $O(\log^3 N)$.

8 Hamiltonian simulation

In this section we use H to denote Hamiltonians and Had for Hadamard!

Given an initial state of a system $|\psi(0)\rangle$ and a Hamiltonian H, our goal is to simulate the time evolution $|\psi(0)\rangle \to e^{-iHt}|\psi(0)\rangle$. The goal of Hamiltonian simulation is to design a circuit U consisting of gates and oracles that approximates the time evolution up to an error ϵ such that

$$\left\|U - e^{-iHt}\right\|_2 < \epsilon, \tag{47}$$

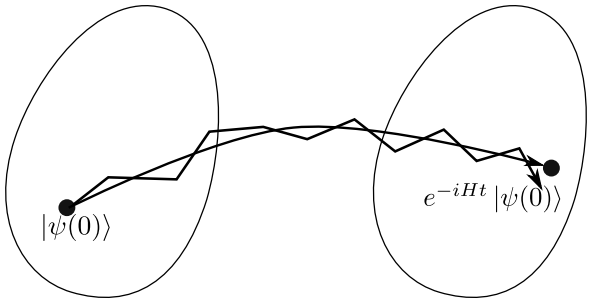


Figure 9: Hamiltonian simulation approximates the time evolution by a series of digital operations.

where $\|\cdot\|_2$ is the spectral norm.

Quantum systems are fundamentally difficult to simulate; the dynamics of quantum systems is a BQP-hard (or BQP complete for Hamiltonians with natural restrictions). Except for a few special cases, the complexity of the best known classical algorithms grows exponentially with the number of qubits. As such, simulation of quantum dynamics is a field where quantum computers can quickly outperform classical ones. In fact, the time evolution of quantum systems was the original application for quantum computers suggested by Feynman.

In the simplest scenario, we assume that the simulated Hamiltonians are given in the form $H = \sum_{j=1}^{m} H_j$, where each H_j is sufficiently simple such that $e^{-iH_j t}$ can be implemented directly for arbitrary t. A common case is when these H_j are Paulis or local Hamiltonians.

The simplest quantum simulation algorithms rely on the Lie-Trotter formula

$$\lim_{r\to\infty}\left(e^{A/r}e^{B/r}\right)^r = \lim_{r\to\infty}\left((1+\frac{A}{r})(1+\frac{B}{r})\right)^r \tag{48}$$

$$= \lim_{r\to\infty}\left(1+\frac{A+B}{r}+\frac{AB}{r^2}\right)^r \tag{49}$$

$$= \lim_{r\to\infty}\left(1+\frac{A+B}{r}\right)^r \tag{50}$$

$$= e^{A+B} \tag{51}$$

Since the individual terms in the Hamiltonian typically do not commute, decomposing an exponential of a sum into a finite sum of exponentials will lead to errors.

Exercise: Prove that $\left\|e^{t(A+B)} - \left(e^{At/r}e^{Bt/r}\right)^r\right\| \in \mathcal{O}\left(\frac{t^2}{r}\right)$ for $\|A\|, \|B\| \leq 1$.

Next, we can recursively that for $H = \sum_{j=1}^{m} H_j$, one can decompose the evolution with respect to H into the evolution with respect to each H_j as

$$\widetilde{U} = \left(e^{-iH_1 t/r}e^{-iH_2 t/r}\ldots e^{-iH_m t/r}\right)^r + \mathcal{O}(\|H\|t^2/r). \tag{52}$$

Thus, if we are willing to tolerate error at most ϵ, we need to perform $\mathcal{O}(\frac{\|H\|t^2}{\epsilon})$ operations.

Up to now, we assumed that we know how to implement each $e^{-iH_j t}$ directly. Let us now show how to implement them in some simple cases.

In the simplest case, H_j is a Pauli Z acting on the jth qubit. The Hamiltonian evolution is then a Z-rotation on the jth qubit.

$$e^{-itZ} = e^{-it}|0\rangle\langle 0| + e^{it}|1\rangle\langle 1| \tag{53}$$

Next, we show how to simulate a tensor product of Zs

$$H = Z_1 \otimes Z_2 \otimes \cdots \otimes Z_n. \tag{54}$$

We first prove the following identity. For an arbitrary unitary U:

$$\begin{aligned} Ue^{-iHt}U^\dagger &= U\sum_{k=0}^{\infty}\frac{(-iHt)^k}{k!}U^\dagger && (55)\\ &= UU^\dagger - iUHU^\dagger t + i^2UH(U^\dagger U)Ht^2U^\dagger \\ &\quad - i^3UH(U^\dagger U)H(U^\dagger U)HU^\dagger t^2 + \ldots && (56)\\ &= \sum_{k=0}^{\infty}\frac{(-iUHU^\dagger t)^k}{k!} && (57)\\ &= e^{-iUHU^\dagger t}. && (58) \end{aligned}$$

We can implement $e^{-iZ_1\otimes\cdots\otimes Z_n t}$ using the circuit in Fig. 10

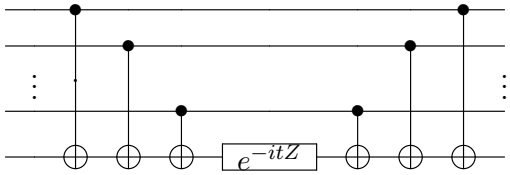


Figure 10: Simulating a tensor product of Pauli Zs.

The correctness of the circuit can be showed through induction. We already proved the first step in (53). In the inductive step, we conjugate an existing circuit by CNOTs, see Fig. 10. The top wire corresponds to the very last register in our notation. We can rewrite $CNOTe^{-itZ_1\otimes Z_2\otimes\cdots\otimes Z_{n-1}}CNOT$ as:

$$\begin{aligned} &(|0\rangle\langle 0|\otimes\mathbb{I} + |1\rangle\langle 1|\otimes X)\left(\mathbb{I}\otimes e^{-itZ_1\otimes\cdots\otimes Z_{n-1}}\right)(|0\rangle\langle 0|\otimes\mathbb{I} + |1\rangle\langle 1|\otimes X)\\ =&|0\rangle\langle 0|\otimes e^{-itZ_1\otimes\cdots\otimes Z_{n-1}} + |1\rangle\langle 1|Xe^{-itZ_1\otimes\cdots\otimes Z_{n-1}}X\otimes\\ =&e^{-itZ_1\otimes\cdots\otimes Z_{n-1}}\otimes|0\rangle\langle 0| + e^{itZ_1\otimes\cdots\otimes Z_{n-1}}\otimes|1\rangle\langle 1|\\ =&e^{-itZ_1\otimes Z_2\otimes\cdots\otimes Z_{n-1}\otimes Z_n} \end{aligned}$$

Simulating other Paulis is possible by changing the basis. Recall that $e^{-itUHU^\dagger} = Ue^{-itH}U^\dagger$. We can then use the identities

$$X = \text{Had } Z \text{ Had} \tag{59}$$

$$Y = S^\dagger\text{Had } Z \text{ Had } S \tag{60}$$

We used Had instead of H to denote the Hadamard gate since we reserved the symbol H for Hamiltonians.

This allows us to simulate evolution according to any Pauli. For example, we can simulate the evolution according to the $H = X \otimes Y \otimes Z$ according to the circuit in Fig. 11.

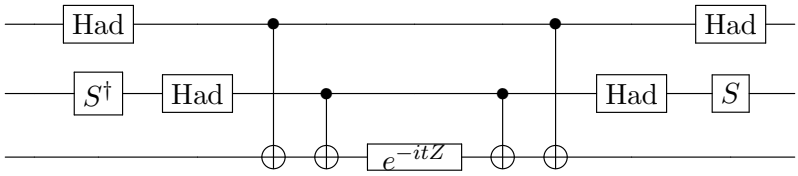


Figure 11: X and Y Paulis can be simulated by a change of basis. The circuit above depicts the simulation according to $e^{-itX\otimes Y\otimes Z}$.

Now we know how to simulate any Hamiltonian that is a sum of Paulis. The complexity of the algorithm for simulating a Hamiltonian $H = \sum_{l=0}^{L-1} P_l$ where P_ns are Paulis on at most n qubits is

$$\mathcal{O}\left(\frac{Lt^2 n}{\epsilon}\right). \tag{61}$$

However, there is still a need for more efficient algorithms. First, the number of terms L needed to decompose a Hamiltonian into a sum of Paulis can be exponentially large. Second, the scaling in terms of t and ϵ is quite poor.

Other Hamiltonian simulation algorithms allow for different decomposition of Hamiltonians. A popular one is decomposing a sparse matrix into 1-sparse matrices which can be then simulated directly. Another one is decomposition into a linear combination of unitaries (LCU) which is a generalization of the Pauli decompositions.

Exercise: Let ρ, σ be density matrices, S the SWAP operator and Tr_p partial trace over the first variable. Show that

$$\mathrm{Tr}_p[e^{-iS\Delta}\rho \otimes \sigma e^{iS\Delta}] = e^{-i\rho\Delta}\sigma e^{i\rho\Delta} + \mathcal{O}(\Delta^2)$$

In these more general cases the Hamiltonian is accessed through an oracle. One type of oracular access is particularly common when the Hamiltonian (in a computational basis) is given by a sparse matrix. We say a Hamiltonian is *row-d-sparse* if each row has at most d non-zero entries. If there is an efficient procedure to locate these entries we moreover say that the Hamiltonian is *row-computable*. In this case, one can efficiently construct oracles

$$O_{loc}|r, k\rangle = |r, k \oplus l\rangle \tag{62}$$

$$O_{val}|r, l, z\rangle = |r, l, z \oplus H_{r,l}\rangle. \tag{63}$$

Oracle O_{loc} locates the position l of the k-th non-zero element in row r. The oracle O_{val} then gives the value of the matrix element $H_{r,l}$. We compute the cost of algorithms in terms of the number of queries to these oracles.

It is possible to construct different oracles. Any Hermitian matrix can be decomposed into a sum of unitaries

$$H = \sum_{l=0}^{L-1} \alpha_l H_l, \tag{64}$$

where for each l, $\alpha_l \geq 0$ and H_l is a unitary matrix $\|H_l\| = 1$. This decomposition can be efficiently implemented for sparse Hamiltonians. The coefficients α_l and unitaries H_l can be accessed through oracles

$$O_\alpha |l, z\rangle = |l, z \oplus \alpha_l\rangle \tag{65}$$

$$O_{V_l} |l, \psi\rangle = V_l |l, \psi\rangle, \tag{66}$$

or, in some cases, described classically.

9 Grover's Algorithm

Problem 14 (The search problem)**.** *For $N = 2^n$, we are given a marked item $\boldsymbol{w} \in \mathbb{Z}_2^N$, and the goal is to locate $\boldsymbol{w}$.*

The classical solution is easy to see. In the worse case, the algorithm has to check all N items in order to find $\boldsymbol{x}$.

In the quantum setting, we are given an oracle U_{G} so that

$$U_{\mathrm{G}} |\boldsymbol{x}\rangle = \begin{cases} -|\boldsymbol{x}\rangle, & \text{if } \boldsymbol{x} = \boldsymbol{w} \\ |\boldsymbol{x}\rangle, & \text{otherwise} \end{cases}. \tag{67}$$

In other words, the Grover's oracle can add a phase to the target element $|\boldsymbol{w}\rangle$. We can see that

$$U_{\mathrm{G}} = I - 2|\boldsymbol{w}\rangle\langle\boldsymbol{w}|. \tag{68}$$

We will also define a *diffusion* operator as follows:

$$U_{\mathrm{d}} = 2|\boldsymbol{s}\rangle\langle\boldsymbol{s}| - I, \tag{69}$$

where

$$|\boldsymbol{s}\rangle = \frac{1}{\sqrt{N}} \sum_{i=0}^{N-1} |i\rangle. \tag{70}$$

Note that the diffusion operator can be easily implemented as follows:

$$U_{\mathrm{d}} = H^{\otimes n} \big(2|0\rangle\langle 0|^{\otimes n} - I\big) H^{\otimes n}. \tag{71}$$

The quantum circuit for the Grover's search algorithm is illustrated in Figure 12, where U_{G} and U_{d} are given in Eqs. (68) and (69), respectively.

The Grover's algorithm has the following steps. As the first step, the algorithm prepares a uniform superposition state:

$$|\Psi_{t_1}\rangle = H^{\otimes n} |0\rangle^{\otimes n} = \left(\frac{1}{\sqrt{N}} \sum_{i=0}^{N-1} |i\rangle \right) \equiv |\boldsymbol{s}\rangle. \tag{72}$$

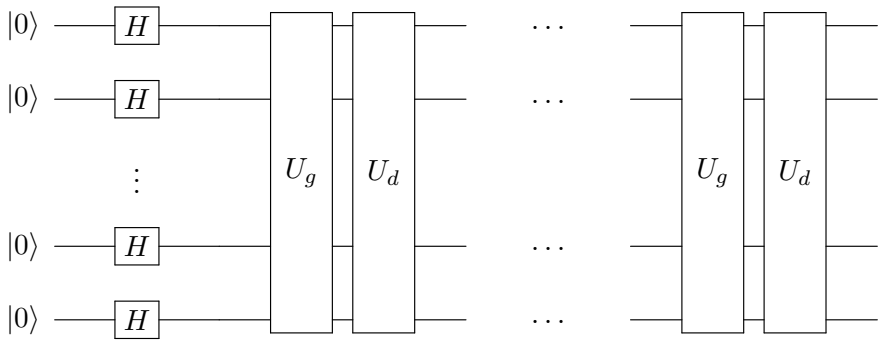


Figure 12: Grover's algorithm

At the next step, the protocol employs the Grover's oracle U_{G}:

$$\begin{aligned} |\Psi_{t_2}\rangle &= U_{\mathrm{G}}|\Psi_{t_1}\rangle \\ &= |\boldsymbol{s}\rangle - \frac{2}{\sqrt{N}}|\boldsymbol{w}\rangle, \end{aligned} \tag{73}$$

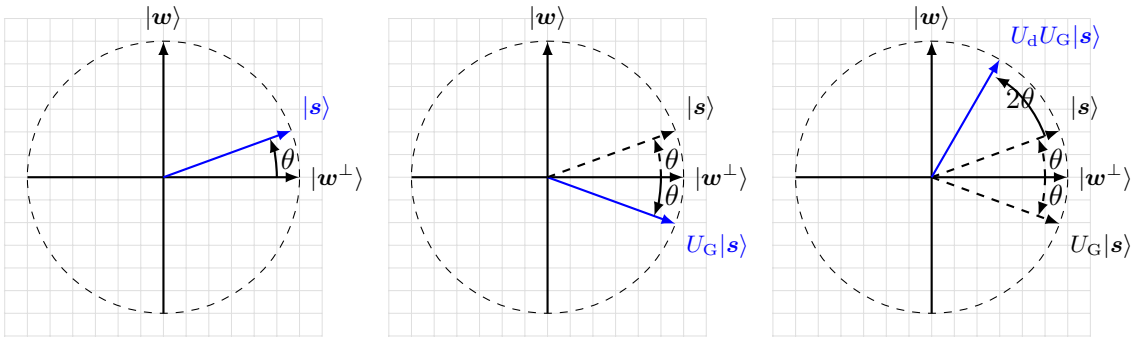
because $\langle \boldsymbol{s}|\boldsymbol{w}\rangle = \frac{1}{\sqrt{N}}$. Then, the protocol employs the diffusion oracle U_{d}:

$$|\Psi_{t_3}\rangle = U_{\mathrm{d}}|\Psi_{t_2}\rangle \tag{74}$$

$$= (2|\boldsymbol{s}\rangle\langle\boldsymbol{s}| - I)\left(|\boldsymbol{s}\rangle - \frac{2}{\sqrt{N}}|\boldsymbol{w}\rangle\right) \tag{75}$$

$$= \left(\frac{N-4}{N}|\boldsymbol{s}\rangle\right) + \frac{2}{\sqrt{N}}|\boldsymbol{w}\rangle. \tag{76}$$

We say that "one iteration" of Grover's search algorithm consists of the employment of U_{G} followed by U_{d}. In the following, we aim to show that (i) the protocol can find the target $\boldsymbol{w}$ with $O(\sqrt{N})$ iterations with successful probability ≈ 1 and (ii) this is optimal given access to quantum computers.



(a) Step t_1 **(b)** Step t_2 **(c)** Step t_3

Figure 13: Geometric illustration of Grover's search procedure

The easiest way to prove statement (i) is by the following geometric argument in Figure 13. Denote

$$|\boldsymbol{w}^\perp\rangle = \frac{1}{\sqrt{N-1}} \sum_{\boldsymbol{x}\neq\boldsymbol{w}} |\boldsymbol{x}\rangle.$$

One can see that the uniform superposition state $|\boldsymbol{s}\rangle$ at the step t_1 can be decomposed into

$$|\boldsymbol{s}\rangle = \sqrt{\frac{1}{N}}|\boldsymbol{w}\rangle + \sqrt{\frac{N-1}{N}}|\boldsymbol{w}^{\perp}\rangle, \tag{77}$$

and the angle θ in Figure 13a corresponds to

$$\sin\theta = \sqrt{\frac{1}{N}}, \qquad \cos\theta = \sqrt{\frac{N-1}{N}}. \tag{78}$$

At the second step, application of U_{G} leads to

$$\begin{aligned} U_{\mathrm{G}}|\boldsymbol{s}\rangle &= -\sqrt{\frac{1}{N}}|\boldsymbol{w}\rangle + \sqrt{\frac{N-1}{N}}|\boldsymbol{w}^{\perp}\rangle \\ &= -\sin\theta|\boldsymbol{w}\rangle + \cos\theta|\boldsymbol{w}^{\perp}\rangle. \end{aligned} \tag{79}$$

Geometrically, the oracle U_{G} reflects the vector $|\boldsymbol{s}\rangle$ along the axis $|\boldsymbol{w}^{\perp}\rangle$ in Figure 13b. Finally, application of U_{d} at the third step t_3 to Eq. (79) is equivalent to reflect the state $U_{\mathrm{G}}|\boldsymbol{s}\rangle$ along the axis $|\boldsymbol{s}\rangle$. Therefore, the application of $U_{\mathrm{d}}U_{\mathrm{G}}$ yields

$$U_{\mathrm{d}}U_{\mathrm{G}}|\boldsymbol{s}\rangle = \sin 3\theta|\boldsymbol{w}\rangle + \cos 3\theta|\boldsymbol{w}^{\perp}\rangle. \tag{80}$$

By induction, after k iterations, we have

$$(U_{\mathrm{d}}U_{\mathrm{G}})^k|\boldsymbol{s}\rangle = \sin(2k+1)\theta|\boldsymbol{w}\rangle + \cos(2k+1)\theta|\boldsymbol{w}^{\perp}\rangle. \tag{81}$$

If we measure after k iterations, the probability of obtaining the target element $\boldsymbol{w}$ is

$$p_k := \Pr\{\boldsymbol{w} \text{ appears}\} = \sin((2k+1)\theta)^2. \tag{82}$$

If we choose $k = \frac{\pi}{4\theta} - \frac{1}{2}$, then the Grover's algorithm will produce the state $|\boldsymbol{w}\rangle$ with certainty because $p_k = 1$. However $k = \frac{\pi}{4\theta} - \frac{1}{2}$ will unlikely be an integer, but we can still show that if $\tilde{k}$ is an integer closest to k and $1 \ll N$, then the failure probability decays proportional to N:

$$\begin{aligned} 1 - p_{\tilde{k}} &= \cos((2\tilde{k}+1)\theta)^2 \\ &= \cos((2k+1)\theta + 2(\tilde{k}-k)\theta)^2 \\ &= \cos(\pi/2 + 2(\tilde{k}-k)\theta)^2 \\ &= \sin(2(\tilde{k}-k)\theta)^2 \\ &\leq \sin(\theta)^2 \\ &= \frac{1}{N}, \end{aligned} \tag{83}$$

where the first inequality follows because $|k - \tilde{k}| \leq 1/2$. Since $\arcsin\theta \geq \theta$, then

$$\tilde{k} \leq \frac{\pi}{4\theta} = \frac{\pi}{4}\sqrt{N}. \tag{84}$$

Hence, we can see that the number of iteration is $O(\sqrt{N})$. While Grover's algorithm is often presented as a search in an unsorted database, using it within such a quantum data structure would wipe up all the speedup. Instead, the speedup from Grover search is most commonly seen to boost the success probability of other algorithms in its modified form known as amplitude amplification.

10 Additional reading

UTS courses Quantum Software and Quantum Algorithms both build on the topics from this lecture. Introduction to algorithms [1] is a wonderful introduction textbook to classical algorithms and data structures. A lot of the explanations here is based on [2]. Lecture notes on quantum algorithms from Andrew Childs (available online) cover quantum algorithms with a lot more detail.

References

[1] Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, and Clifford Stein, *Introduction to algorithms*, MIT press, 2009.

[2] Phillip Kaye, Raymond Laflamme, Michele Mosca, et al., *An introduction to quantum computing*, Oxford University Press on Demand, 2007.