UNIVERSIDAD
DE GRANADA

Facultad de Ciencias
E.T.S. de Ingenierías Informática y de Telecomunicación

Doble Grado en Ingeniería Informática y Matemáticas

TRABAJO DE FIN DE GRADO

Modelos de computación cuánticos

Presentado por: Pablo Baeyens Fernández

Tutor: Serafín Moral Callejón

Curso académico 2018–2019

Contents

Introduction

In the 1920s, quantum physics sparked a revolution in the field of physics, which prompted the creation of all kinds of technologies. A decade later, Alan Turing first proposed the *Turing machine model*, giving birth to the field of theoretical computer science.

It was not until the 1980s that the idea of quantum computation, combining these two disciplines, was first proposed, where they were first suggested as a way to efficiently simulate physical phenomena (Feynman (1982)).

This combination of the principles of quantum physics with the ideas of computation allowed for a more general model of computers that entailed a radical paradigm shift from the previous way of doing computation. It was in this decade when the first theoretical results and algorithms were first discovered.

On the one hand, algorithms such as Shor's algorithm (Shor (1994)) or Grover's algorithm (Grover (1996)), which solved problems asymptotically faster than any probabilistic algorithm known at the time generated an intense research program into the development of quantum algorithms and the research of quantum systems' properties.

On the other hand, results such as the *no-cloning theorem*, first stated in 1982 (Wootters and Zurek (1982)), or the optimality of Grover's algorithm hinted at the limits of quantum computation and how its capacities were harder to exploit than expected.

Today, quantum computation is a young yet thriving field that has seen its first experimental success with real quantum computers and that has both a more theoretical side from the field of complexity theory and a more applied one when it comes to the design and implementation of quantum algorithms.

This document attempts at giving an overview of some results in quantum computation, as well as implementing some algorithms in a quantum programming language. Some of these algorithms can be simulated while, for others, an assesment of the needed resources can be given.

The first part, encompassed by chapters 1 to 4, is more theoretical and describes the classical, probabilistic and quantum computation model in detail using the circuit model and the tools of complexity theory. It also states and proves some results that allow us to understand the relations between those models.

The second part, chapters 5 to 8, discusses the fundamentals for programming in the Quipper programming language, (Green et al. (2013a)), and describes several key quan-

tum algorithms and subroutines, giving a formal justification of their effectiveness as well as a description of their circuit families using Quipper. The code is also available in the attached document and can be generated as an executable to try out some of the algorithms and obtain a graphical representation of others.

For the development of this document practical and theoretical results from the areas of geometry and linear algebra, computation models, probability and functional programming have been needed.

The main sources used for the writing of this document were Nielsen and Chuang (2010), Kaye, Laflamme, and Mosca (2007), Arora and Barak (2009) and Green et al. (2013a). The sources used for each section and chapter are quoted within the text.

Main goals and results achieved

The initial goals of this bachelors' thesis were

1. to establish a theoretical and mathematical study of quantum models and quantum complexity classes, as well as their relation to classical and probabilistic models,
2. to study the quantum programming languages available, their capabilities, and to choose one of them and
3. to describe, develop some quantum algorithms in this programming language, as well as to make a theoretical study of these.

The first two were completely successful. For the latter, the Quipper programming language was chosen both because of the closeness of its underlying model as well as its succintness and expressiveness within the functional programming realm.

The third goal was partially successful. Several quantum algorithms were effectively implemented in the Quipper programming language, yet some of them could not be executed since the simulation was unfeasible for the available computational resources.

To make up for this, in these algorithms a graphical depiction of these algorithms as well as an assesment of the necessary resources for their execution were stated.

For the achievement of these goals I have needed to make use of results relating to: Hilbert spaces and geometry, computational models, probability and statistics and functional programming and software design.

Summary

Brief summary

In this document, we develop the theoretical fundamentals of quantum computation and put some of these ideas into practice.

In the first part, a theoretical outlook to quantum and classical computation is given. First, the Hilbert space formalism for quantum physics, the Turing machine and the uniform circuit model are described, giving an unified framework for describing computation classically, probabilistically and quantumly. The basic properties of these models are studied.

Then, several key complexity classes are described, such as $\mathsf{P}, \mathsf{NP}, \mathsf{BPP}, \mathsf{BQP}$ and QMA. The main properties of the complexity classes as well as the known unrelativized relations between these classes are described and summarized.

In the second part, the purely functional quantum programming language Quipper is described, together with the necessary functional programming concepts drawn from Haskell needed, such as monads or generalized typeclasses.

Finally, three main quantum algorithms together with their subroutines are described: the Deutsch-Jozsa algorithm, Shor's algorithm and Grover's algorithm. Their formal justification is given and they are described in the Quipper programming language. If possible, a simulation of their execution is given.

Keywords: Theoretical computer science, quantum computation, quantum algorithms, functional programming, search problems

Resumen extendido en español

Si bien el campo de la computación cuántica es altamente interdisciplinar y necesita en todos sus aspectos de informática y matemáticas, a grandes rasgos puede distinguirse una parte con un enfoque más teórico y **matemático** en los capítulos **1** a **4** y una parte con una orientación más aplicada e **informática** en los capítulos **5** a **8**.

La parte matemática describe el modelo de estados puros para la computación cuántica y utiliza el modelo de familias de circuitos uniformes para describir de forma unifi-

cada diversos resultados de complejidad que permiten hacer un análisis teórico de las capacidades de los algoritmos cuánticos. En concreto:

Capítulo 1: En este capítulo se procede al desarrollo matemático de los principios de la mecánica cuántica. En el modelo de física cuántica de *estados puros* en dimensión finita, un *qubit* es el espacio proyectivo asociado a un espacio de Hilbert de 2 dimensiones con una base computacional ortonormal fijada. Estos qubits u otros espacios de estados pueden combinarse en un espacio de estados compuesto mediante el producto tensorial de los espacios de los subestados.

Las operaciones válidas que transforman estos espacios de estados son las operaciones *unitarias*: aplicaciones lineales que en dimensión finita se caracterizan porque su inversa viene dada por su transpuesta conjugada. Estas operaciones son la base de la computación cuántica, y se conocen como *puertas cuánticas*. Algunas puertas cuánticas relevantes para la computación cuántica son la puerta de Hadamard, las puertas de cambio de fase o la puerta NOT controlada. Podemos dar su descripción en términos de matrices unitarias.

No todas las operaciones clásicas pueden generalizarse de forma satisfactoria al ámbito cuántico; el *teorema de no-clonado* impide ciertas operaciones en el ámbito cuántico que sí son factibles de forma clásica. Además, la operación de *medición* nos da para cada estado cuántico una distribución de probabilidad sobre los posibles estados de la base computacional.

Capítulo 2: En este capítulo se describe y estudia el modelo clásico de computación. Uno de los enfoques fundamentales de la informática teórica es el estudio de los *problemas de decisión* y su relación con el concepto de los lenguajes formados por palabras de un alfabeto dado. Las transformaciones válidas en el ámbito clásico tradicionalmente se describen con el *modelo de máquina de Turing clásica*, pero, para tener una descripción unificada con el caso cuántico, recurrimos a los circuitos. El *modelo de circuitos uniforme*, que puede verse como la descripción de un morfismo en una categoría monoidal en términos de un conjunto de morfismos básicos llamados *puertas*, puede demostrarse como equivalente al modelo de máquina de Turing clásica en términos de computabilidad y eficiencia computacional.

A partir de estos dos modelos podemos definir *clases de complejidad* deterministas y no deterministas como $\mathsf{P}, \mathsf{P/poly}, \mathsf{NP}$ y PSPACE: conjuntos de problemas de decisión que son resolubles mediante la utilización de una cierta cantidad de recursos en un cierto tipo de máquina. El uso polinomial del tiempo puede verse como un uso eficiente de recursos y por tanto estas clases pueden darnos información sobre si un cierto problema es o no eficiente.

Las relaciones entre estas clases, como que $\mathsf{P} \subseteq \mathsf{NP} \subseteq \mathsf{PSPACE}$ nos aportan cotas inferiores y superiores de la dificultad de resolver ciertos problemas. Otras relaciones, como el problema de P vs. NP permanecen sin resolver y nos ayudarían a comprender estos problemas. En concreto, la clase NP puede verse como una clase de problemas para

los cuales puede demostrarse su resolución para un cierto caso dando un *certificado*.

Capítulo 3: En este capítulo se describe y estudia el modelo probabilístico de computación. Análogamente al modelo descrito en el capítulo 1 para la computación cuántica, puede definirse un modelo de computación probabilística compatible con el formalismo del modelo de circuitos uniformes, utilizando en este caso espacios vectoriales reales como espacio de estados. Las aplicaciones válidas en este caso son un subconjunto de las *aplicaciones estocásticas* (aquellas cuyas entradas pueden calcularse de forma eficiente). Además, estas aplicaciones pueden descomponerse siempre en una parte clásica y un generador de bits aleatorios con un sesgo dado. A partir de estos podemos definir algoritmos probabilísticos que resuelvan un cierto problema.

Para incrementar la probabilidad de éxito de uno de estos algoritmos pueden utilizarse las *cotas de Chernoff*. Estos algoritmos junto con los resultados anteriores permiten definir clases de problemas resolubles de forma eficiente en el caso probabilístico como BPP, en el cual los algoritmos deben tener una probabilidad de éxito acotada. Aunque se cree que los algoritmos probabilísticos no proporcionan una ventaja asintótica en la práctica, damos un *ejemplo de lenguaje* BPP para el cual no se conoce un algoritmo clásico eficiente. *La clase* PP no requiere esta cota en la probabilidad, y la inclusión $\mathsf{NP} \subseteq \mathsf{PP}$ justifica que esta no se establezca como una clase cuyos problemas puedan resolverse de forma eficiente. Además, podemos generalizar la clase NP a la clase MA. Estas clases dan lugar a una jerarquía de clases clásicas y probabilísticas que nos proporciona información sobre las capacidades de ambos modelos. Existen diversos problemas abiertos en este campo.

Capítulo 4: En este capítulo se describe y estudia el modelo cuántico de computación. La computación cuántica puede describirse en términos de familias de circuitos cuánticos uniformes, cuyas puertas son aplicaciones unitarias junto con la operación de preparación de qubits y la posibilidad de ignorar el resultado de un qubit.

Existen conjuntos finitos de puertas cuánticas (*conjuntos universales de puertas cuánticas*) que permiten aproximar cualquier otra puerta. El *teorema de Solovay-Kitaev*, para el que esbozamos la demostración, nos indica que además esta aproximación puede hacerse de forma eficiente. El modelo cuántico generaliza al modelo clásico y probabilístico mediante el uso de la computación reversible: un proceso por el cual podemos transformar cualquier operación clásica en una operación cuántica que la generaliza en un cierto sentido.

La computación cuántica no proporciona ventajas en términos de computabilidad, pero sí lo hace potencialmente en términos de complejidad. Análogas a las clases BPP, PP y MA podemos definir las clases BQP, PQP y QMA. El resultado $\mathsf{PQP} = \mathsf{PP}$ nos permite establecer la relación $\mathsf{BQP} \subseteq \mathsf{QMA} \subseteq \mathsf{PP}$, que nos da una cota superior clásica a las bondades de la computación cuántica. Esta cota es débil en el sentido de que creemos que $\mathsf{BQP} \neq \mathsf{QMA}$. Puede establecerse así una jerarquía completa de clases definidas en los capítulos 2 a 4.

La parte informática desarrolla varios algoritmos concretos en el modelo definido en los capítulos 1 a 4, tanto de forma téorica como de forma aplicada en el lenguaje Quipper. Trata los siguientes contenidos:

Capítulo 5: En este capítulo se discuten brevemente los lenguajes de programación cuánticos y se justifica la elección del lenguaje de programación funcional puro Quipper, embebido en Haskell. Quipper necesita de varios conceptos de la programación funcional tomados de Haskell, como es el caso de las mónadas, un concepto tomado de la teoría de categorías que nos permite tratar de forma pura con contextos computacionales.

Además, Quipper hace uso de las clases de tipos generalizadas de Haskell y de sus capacidades para metaprogramación en tiempo de compilación. El modelo subyacente a Quipper es un modelo mixto, que combina los circuitos clásicos y los circuitos cuánticos en un único lenguaje de descripción. Este lenguaje distingue entre parámetros y valores; los primeros son conocidos a la hora de generar un circuito mientras que los segundos se conocen sólo durante su ejecución.

El lenguaje se ha integrado como paquete en el sistema `stack`, en el que se han implementado la descripción de los circuitos de capítulos posteriores. En este capítulo describimos también la lectura de funciones clásicas y su transformación a circuitos cuánticos.

Capítulo 6: En primer lugar, en este capítulo se describe el modelo simplificado de complejidad de consultas (*query complexity*) en el que se realizará el análisis de eficiencia de los algoritmos presentados. Este modelo mide la complejidad de un algoritmo en función del número de consultas que hace a una cadena de caracteres dada mediante un *oráculo*.

Bajo este enfoque puede discutirse el problema de Deutsch, uno de los primeros problemas para los cuales se describió un algoritmo cuántico (el algoritmo de Deutsch-Jozsa) que superaba en el modelo de complejidad de consultas la eficiente de cualquier algoritmo clásico.

En la práctica la ventaja es sólo respecto del caso clásico, puesto que en el caso probabilístico es asintóticamente equivalente al caso cuántico en función del conjunto de puertas cuánticas utilizado. Este algoritmo se ha implementado en Quipper y puede encontrarse en el código adjunto.

Capítulo 7: En este capítulo se describe el algoritmo de Shor. El problema de la factorización de enteros es un problema muy relevante de forma práctica, ya que su resolución limita severamente las capacidades de algunos sistemas de encriptación actuales. Su problema de decisión asociado está en NP pero no se conoce ningún algoritmo polinómico clásico o probabilístico. El algoritmo de Shor resuelve este problema en tiempo cuántico polinómico, colocándolo en BQP.

Para hacerlo necesitamos la transformada cuántica de Fourier: un algoritmo que aplica la transformada discreta de Fourier al vector de coordenadas de un estado cuántico. Es más eficiente que cualquier algoritmo clásico conocido pero su aplicabilidad es limitada ya que no podemos recuperar fácilmente la salida; tendríamos que realizar mediciones que no nos dan estas coordenadas (*amplitudes*). Puede describirse fácilmente en Quipper de forma recursiva.

Una de las aplicaciones más importantes es la resolución del problema de estimación de los autovalores de un operador unitario, conocido como el *algoritmo de estimación de fase cuántico*. Este algoritmo puede describirse también de forma sencilla en Quipper, utilizando operadores que combinan operaciones monádicas.

Este algoritmo puede aplicarse en el problema de hallar el orden de un elemento del grupo de las unidades de los enteros módulo un N dado, que puede implementarse en Quipper gracias a sus capacidades para lidiar con enteros de forma cuántica.

Finalmente, el problema de factorización de enteros puede reducirse de forma probabilística al de hallar el orden de una unidad, con lo que podemos definir finalmente el algoritmo de Shor. La parte cuántica de este algoritmo no puede ejecutarse de forma factible en la práctica, por lo que discutimos la implementación de la estimación de recursos para obtener la factorización e implementamos la parte clásica.

Capítulo 8: En este último capítulo se discute desde la perspectiva de la complejidad de consultas el problema de búsqueda no estructurada en un conjunto de palabras. El algoritmo más eficiente en el caso clásico y probabilístico es lineal, pero en el caso cuántico podemos encontrar un algoritmo que proporciona una ventaja cuadrática: el *algoritmo de Grover*.

Este algoritmo tiene una interpretación geométrica como la aplicación de un giro en un plano del espacio de estados que trata el algoritmo, lo que nos permite justificar su funcionamiento. Puede implementarse de forma sencilla en Quipper, mediante la lectura de oráculos descrita anteriormente y la aplicación de operaciones de combinación monádica. Además, este algoritmo es óptimo asintóticamente en el modelo de complejidad de consultas.

Para la ejecución del algoritmo necesitamos saber el número de soluciones, que se puede obtener también de forma eficiente con el *algoritmo de conteo cuántico*. Este algoritmo es una aplicación directa del algoritmo de estimación de fase. Los requerimientos en términos de la dimensión del espacio de estados impiden su simulación directa, pero podemos estimar el número de recursos necesarios.

Palabras clave: informática teórica, computación cuántica, algoritmos cuánticos, programación funcional, problemas de búsqueda

---

Además de este texto, se incluye el código desarrollado para la simulación de los algoritmos de Grover y Deutsch-Jozsa y la estimación de recursos de los algoritmos de

Shor y de conteo cuántico, así como la generación de los diagramas presentes en el texto. Una descripción del código, sus tests y documentación puede hallarse en el apéndice suplementario, así como en los ficheros `LEEME.md` (en español) y `README.md` (en inglés).

Part I.

Quantum complexity theory

1. A model of quantum mechanics

The study of quantum computers requires us to state the basic principles and mathematical structures this physical theory is based on. The physical systems considered in the theory of quantum computation are finite dimensional, yet in the general case physical systems are described by a system with possible infinite dimensions. In this section we describe five principles that describe quantum mechanical systems at the appropriate level for the development of the theory of quantum computation. We follow the presentation given at (Nielsen and Chuang (2010), chap. 2) and (Lipton and Regan (2014), chap. 1).

1.1. State space and bra-ket notation

A concrete state of an isolated physical system is described by a **state vector** that belongs to a **state space**. In the case of quantum mechanics these are described by (projective) Hilbert spaces. The assumption of isolation is not met in practice, yet error correction techniques can be efficiently applied to make up for this disconnect between theory and practice, so that the model we will describe can produce useful practical results (Nielsen and Chuang (2010), section 10.6).

Definition 1.1: *A (complex) **pre-Hilbert space** is a pair $(H, \langle\cdot|\cdot\rangle : H^2 \to \mathbb{C})$ such that*

1. *H is a complex vector space,*
2. *$\langle\cdot|\cdot\rangle$ (the* product*) is* sesquilinear*, that is, if $u, v \in H$ and $\alpha, \beta \in \mathbb{C}$, $\langle u|v\rangle = \overline{\langle v|u\rangle}$ and*

$$\langle \alpha u + \beta v|w\rangle = \overline{\alpha}\langle u|w\rangle + \overline{\beta}\langle v|w\rangle\,,$$

3. *$\langle\cdot|\cdot\rangle$ is* definite positive*, that is, if $v \in H$, $\langle v|v\rangle \geq 0$ and $\langle v|v\rangle = 0 \iff v = 0$.*

*An **orthonormal basis** of H is a basis $\{u_i\}_{i\in I}$ of H as a vector space such that $\langle u_i|u_j\rangle = \delta_{ij}$.*

*The product of a pre-Hilbert space induces a norm over H given by $\|v\| = \sqrt{\langle v|v\rangle}$. A **Hilbert space** is a pre-Hilbert space such that $(H, \|\cdot\|)$ is complete.*

Every finite dimensional normed space over $\mathbb{C}$ is complete, therefore if H is a finite dimension pre-Hilbert space then H is a Hilbert space. We will focus on this scenario when dealing with quantum computers; all the Hilbert spaces considered will be finite dimensional. Vectors in quantum computer science are usually written as a *ket* between $|$ and $\rangle$.

The canonical finite dimensional Hilbert space of dimension N is $\mathbb{C}^N$ with product given by

$$\langle u|v\rangle = \sum_{j=1}^{N} \overline{u_j} v_j \quad \text{where } u = (u_1, \ldots, u_N), v = (v_1, \ldots, v_N)$$

An orthonormal basis of $\mathbb{C}^N$ is the computational basis of $\mathbb{C}^N$ as a vector space $B = \{|e_i\rangle\}_{i=1,\ldots,N}$. Every finite dimensional Hilbert space of dimension N is isomorphic to $\mathbb{C}^N$, by taking its coordinates over an orthonormal basis, thus we will focus on these spaces.

An alternative way of considering this inner product is by the use of the *adjoint*.

Definition 1.2: *Let $A \in \mathcal{M}_{N\times N}(\mathbb{C})$. The **adjoint** or **conjugate transpose** of A, $A^\dagger$, is given by*

$$(A^\dagger)_{ij} = \overline{A}_{ji}.$$

Given a vector $|v\rangle \in \mathbb{C}^N$ we define $\langle v| = |v\rangle^\dagger$ (by considering $|v\rangle$ as a matrix). That way $\langle u| \cdot |v\rangle = \langle u|v\rangle$, which justifies the notation.

Definition 1.3: *The **projective Hilbert space** $P(H)$ of a Hilbert space H is the quotient of $H\backslash\{0\}$ under the relation*

$$x \sim y \iff \exists \lambda \in \mathbb{C}\backslash\{0\} : \; x = \lambda y$$

Thus, each element of a Hilbert space is a subspace of dimension 1 called a *ray*.

Principle 1: *The state space of an isolated (quantum) physical system is the projective space associated with a complex separable Hilbert space (also called state space).*

Thus, the state vectors of an isolated physical system are *rays* on a Hilbert space. We will identify a ray with a unit vector that generates it. This unit vector is unique up to a constant of the form $e^{i\theta}$ with $\theta \in \mathbb{R}$.

A **qubit** is an isolated physical system with state space $\mathbb{P}\mathbb{C}^2$ in which we fix an orthonormal basis $|0\rangle, |1\rangle$ (called the *computational basis*). The term is also used to refer to an element of this state space; using the identification of rays with unit vectors a qubit can also refer to a vector

$$|\psi\rangle = \alpha|0\rangle + \beta|1\rangle \quad \text{such that } \||\psi\rangle\| = |\alpha|^2 + |\beta|^2 = 1$$

The coefficients α, β are called amplitudes:

Definition 1.4 (Amplitudes)**:** *The* amplitudes *of a quantum state are its coordinates with respect to the computational basis.*

The coordinate vector is the amplitude vector *of the state.*

1.2. Composite systems

A physical system may be composed of several separate subsystems. The relationship between the state spaces of the subsystems and the composite system is given by the tensor product, which is justified by the following result:

Proposition 1.1: *Let H_1, H_2 be finite dimensional Hilbert spaces with orthonormal basis $B_1 = \{u_i\}_{i\in I}, B_2 = \{v_j\}_{j\in J}$ respectively. Then $H_1 \otimes H_2$ is a finite dimensional Hilbert space with inner product given by the linear extension of*

$$\langle u \otimes v | u' \otimes v' \rangle = \langle u|u'\rangle \langle v|v'\rangle \tag{1.1}$$

and $B_1 \otimes B_2 = \{u_i \otimes v_j\}_{(i,j)\in I\times J}$ is an orthonormal basis.

Proof.

By definition $H_1 \otimes H_2$ is a complex vector space with $\dim(H_1 \otimes H_2) = \dim(H_1)\dim(H_2)$ and $B_1 \otimes B_2$ is a basis of it. Furthermore

$$\langle u_i \otimes v_j | u_l \otimes v_k \rangle = \langle u_i|u_l\rangle \langle v_j|v_k\rangle = \delta_{il}\delta_{jk} \tag{1.2}$$

which is equal to one if and only if $i = l$ and $j = k$ and it is zero otherwise.

It suffices to prove that the product is definite positive. Let $v \in H_1 \otimes H_2$. Since $B_1 \otimes B_2$ there exists unique $\alpha_{ij} \in \mathbb{C}$ such that

$$v = \sum_{i,j} \alpha_{ij}(u_i \otimes v_j)\,.$$

and by linearity

$$\langle v|v\rangle = \sum_{i,j,k,l} \alpha_{il}\alpha_{jk}\langle u_i \otimes v_j | u_l \otimes v_k\rangle = \sum_{i,j} \alpha_{ij}^2 \geq 0\,,$$

Furthermore $\langle v|v\rangle = 0 \iff \sum_{i,j}\alpha_{ij}^2 = 0 \iff \alpha_{ij} = 0\ \forall i,j$.

Therefore $H_1 \otimes H_2$ is a Hilbert space with the previously defined Hilbert product and by eq. 1.2 we know that $B_1 \otimes B_2$ is therefore an orthonormal basis.

□

Therefore we can write:

Principle 2: *The state space of a composite system is the tensor product of the state spaces of the subsystems.*

The state vector of a composite system is the tensor product of the state vectors of the subsystems.

We will write $|\phi\psi\rangle := |\phi\rangle\,|\psi\rangle := |\phi\rangle \otimes |\psi\rangle$. As in Principle 1, we take the projective Hilbert space and work with the rays. By eq. 1.1 the tensor product of two unit vectors is a unit vector and thus we can take the tensor product of the representatives of two rays as a unit norm representative of the tensor product of the rays.

We will mostly use composite systems made out of qubits. The tensor product of N qubits has a state space of 2^n dimension. We fix as a basis, the tensor product of the computational basis of each qubit

$$|a_1 \dots a_n\rangle \qquad \text{where } a_i \in \{0,1\}$$

The basis is ordered in lexicographic order.

In what follows, we will use letters from the Latin alphabet as in $|x\rangle, |y\rangle, |z\rangle, |a\rangle \dots$, to write elements from the computational basis and letters from the Greek alphabet as in $|\phi\rangle, |\psi\rangle$, to denote a general element from a composite system.

Furthermore, given $0 \leq i < 2^N$ we may write $|i\rangle = |a_1 \dots a_N\rangle$ where $a_1 \dots a_N$ is the representation of i in binary.

Example 1.1 (2 qubit system)**:** *Let Q_A, Q_B be two qubits with orthonormal basis $|0\rangle_A, |1\rangle_A$ and $|0\rangle_B, |1\rangle_B$ respectively. An orthonormal basis of the composite system $Q_A \otimes Q_B$ is*

$$(|0\rangle_A \otimes |0\rangle_B, |0\rangle_A \otimes |1\rangle_B, |1\rangle_A \otimes |0\rangle_B, |1\rangle_A \otimes |1\rangle_B),$$

which we write $(|00\rangle, |01\rangle, |10\rangle, |11\rangle)$ or $(|0\rangle, |1\rangle, |2\rangle, |3\rangle)$.

1.3. Quantum operations

Following the classical model, the theory of quantum computation considers time as a discrete variable, thus we are only interested in the evolution of quantum systems in discrete steps. Two kinds of operations are possible within this framework: *unitary* operations, which are reversible and keep the system in a quantum state and *measurements* which are irreversible and non-deterministic and return a classical bit.

Given an operator $A : H \to H'$ and $|\psi\rangle \in H$ we write $A\,|\psi\rangle := A(|\psi\rangle)$. We will identify a linear operator with its matrix over the fixed basis.

Operators form a normed vector space given by the following proposition

Proposition 1.2: *Let X, Y be two normed (complex) spaces and let $L(X,Y)$ be the set of linear maps $A : X \to Y$. Then $(L(X,Y), \|\cdot\|)$ is a normed (complex) space, with $\|\cdot\|$ given by*

$$\|A\| = \sup\{\|A\,|x\rangle\| \;:\; |x\rangle \in X, \||x\rangle\| = 1\}.$$

The operators that can be used in quantum mechanics are unitary.

Definition 1.5: *Let H be a Hilbert space, a **unitary operator** $U : H \to H$ is a continuous linear operator such that*

1. *U is surjective and*
2. *U is an isometry, that is, for all $|\phi\rangle, |\psi\rangle \in H$, $\langle\phi|\psi\rangle = \langle U\phi|U\psi\rangle$.*

If $\||\psi\rangle\| = 1$ then $\|U|\psi\rangle\| = \sqrt{\langle U\psi|U\psi\rangle} = \sqrt{\langle\psi|\psi\rangle} = \||\psi\rangle\| = 1$, therefore a unitary operator takes unit vectors into unit vectors. This justifies the identification of unit vectors with rays.

In finite dimensional Hilbert spaces this concept admits a simple characterization on the matrix associated with the operator.

Proposition 1.3: *Let $U : \mathbb{C}^N \to \mathbb{C}^N$ be a linear operator, then*

$$U \text{ is unitary} \iff U \text{ is invertible and } U^{-1} = U^\dagger .$$

Proof.

$\Rightarrow$) Let U be unitary. We consider its matrix representation with respect to $\{e_i\}_{i\in I}$.

$$(U^\dagger U)_{ij} = \langle e_i| U^\dagger U |e_j\rangle = |e_i\rangle^\dagger U^\dagger U |e_j\rangle = \langle Ue_i|Ue_j\rangle = \langle e_i|e_j\rangle = \delta_{ij}$$

where we have used $(AB)^\dagger = B^\dagger A^\dagger$. Therefore U is invertible and $U^\dagger U = I$.

$\Leftarrow$) Since U is invertible it is surjective. Furthermore, let $|\phi\rangle, |\psi\rangle \in \mathbb{C}^N$. Then

$$\langle U\phi|U\psi\rangle = \langle\phi| U^\dagger U |\psi\rangle = \langle\phi| I |\psi\rangle = \langle\phi|\psi\rangle$$

□

Quantum operations are represented by unitary operators, as stated in the following principle:

Principle 3: *The evolution of the state of a quantum system from time t_1 to time $t_2 > t_1$ is given by a unitary transformation U, that is*

$$|\psi(t_2)\rangle = U |\psi(t_1)\rangle$$

If we apply a unitary operation on each subsystem the resulting operation is also a unitary operation. This is given by the following well-known proposition:

Proposition 1.4: *Let H_1, H_2 be finite dimensional Hilbert spaces with orthonormal basis $B_1 = \{u_i\}_{i\in I}, B_2 = \{v_j\}_{j\in J}$ respectively and let $U_i : H_i \to H_i$ $(i = 1, 2)$ be unitary operators.*

Then $U^{(1)} \otimes U^{(2)} : H_1 \otimes H_2 \to H_1 \otimes H_2$ given by

$$U^{(1)} \otimes U^{(2)} |u_i\rangle \otimes |v_j\rangle = U^{(1)} |u_i\rangle \otimes U^{(2)} |v_j\rangle \quad \forall i \in I, j \in J,$$

is a unitary operator.

Furthermore, if $U^{(1)} = (u_{ij}^{(1)})$*, its matrix expression wrt. the basis* $B_1 \otimes B_2$ *is given by the* Kronecker product,

$$U^{(1)} \otimes U^{(2)} = \begin{pmatrix} u_{11}^{(1)} U^{(2)} & \cdots & u_{1n}^{(1)} U^{(2)} \\ \vdots & \ddots & \vdots \\ u_{m1}^{(1)} U^{(2)} & \cdots & u_{mn}^{(1)} U^{(2)} \end{pmatrix}.$$

Lastly we state the measurement operation for finite dimensional systems.

Principle 4: *A measurement of a quantum system of dimension* $N = 2^n$ *with state*

$$|\psi\rangle = \sum_{i=0}^{N-1} \alpha_i |i\rangle$$

is a discrete random variable X *such that*

$$P(X = |i\rangle) = |\alpha_i|^2 \qquad (i = 1, \ldots, N)$$

Since we take $\||\psi\rangle\| = 1$ this means

$$\sum_{i=0}^{N-1} |\alpha_i|^2 = 1$$

Furthermore, $|\alpha_i|^2 \geq 0$, so the random variable defined at Principle 4 is well-defined.

We can restrict ourselves to measurements on the computational basis, but since the change of basis matrix for any other orthonormal basis is unitary we can in practice measure with respect to any orthonormal basis by applying an appropriate unitary operation before the measurement.

1.3.1. Some useful quantum operations

Proposition 1.5: *The following operators given by their matrices wrt. the computational basis are unitary:*

Identity *The identity operator (for any number of qubits) is unitary,*
Hadamard *The* Hadamard gate *is the (1 qubit) unitary operation given by*

$$H = \frac{1}{\sqrt{2}} \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix}, \qquad H|0\rangle = \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle) \quad H|1\rangle = \frac{1}{\sqrt{2}}(|0\rangle - |1\rangle).$$

It is its own inverse.

NOT *The* NOT gate *(or* X gate*) is the (1 qubit) unitary operation given by*

$$X = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}, \qquad X|0\rangle = |1\rangle \quad X|1\rangle = |0\rangle .$$

On classical states it performs the logical NOT operation.

SWAP *A* SWAP gate *is the (2 qubit) unitary operation given by*

$$\text{SWAP} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}.$$

It swaps the position of two qubits: $\text{SWAP}|i\rangle|j\rangle = |j\rangle|i\rangle$.

Phase change *Let* $\theta \in [0, 2\pi]$*. The* θ-phase change gate *is the (1 qubit) unitary operation given by*

$$R_\theta = \begin{pmatrix} 1 & 0 \\ 0 & e^{i\theta} \end{pmatrix}, \qquad R_\theta|0\rangle = 0, \quad R_\theta|1\rangle = e^{i\theta}|1\rangle .$$

Controlled U***-gate*** *Given an* n *qubit unitary operation* U*, the controlled* U *gate is an* $n+1$ *qubit operation given by*

$$C_U = \left(\begin{array}{c|c} I_N & 0 \\ \hline 0 & U \end{array}\right), \qquad C_U|a\rangle|x\rangle = |a\rangle U^a|x\rangle ,$$

that is, $C_U|0\rangle|x\rangle = |0\rangle|x\rangle$ *and* $C_U|1\rangle|x\rangle = |1\rangle U|x\rangle$*. It serves as a conditional operation.*

CNOT *The controlled not operation is the operation* $\text{CNOT} = C_{\text{NOT}}$*. It maps* $\text{CNOT}|x,y\rangle = |x, x \oplus y\rangle$*, where* $\oplus : \mathbb{B}^2 \to \mathbb{B}$ *is addition modulo 2. Since it is a controlled gate, its matrix expression is*

$$\text{CNOT} = \left(\begin{array}{c|c} I_2 & 0 \\ \hline 0 & X \end{array}\right),$$

where X *is the* NOT *gate matrix.*

Toffoli gate *The* Toffoli gate *(or* CCNOT gate*) is the 3 qubit gate that maps* $\text{CCNOT}|x,y,z\rangle = |x,y,z \oplus (x \cdot y)\rangle$*. Its matrix expression is*

$$\text{CCNOT} = \left(\begin{array}{c|c} I_6 & 0 \\ \hline 0 & X \end{array}\right),$$

where X *is the* NOT *gate matrix.*

Furthermore, it is easy to show that cloning is not a unitary operation. This is the famous *no-cloning theorem*.

Proposition 1.6 (No-cloning theorem)**:** *There is no unitary map* $U : Q^{\otimes 2n} \to Q^{\otimes 2n}$ *and state* $|x_0\rangle \in Q^{\otimes n}$ *that for an arbitrary state* $|\psi\rangle \in Q^{\otimes n}$ *maps*

$$U |\psi\rangle |x_0\rangle = |\psi\rangle |\psi\rangle .$$

Proof.

(Nielsen and Chuang (2010), Box 12.1)

Suppose there exists such unitary map U. Let $|\psi\rangle , |\phi\rangle \in Q^{\otimes n}$.

By Definition 1.5 we have

$$\begin{aligned} \langle\psi|\phi\rangle &= \langle\psi|\phi\rangle\langle x_0|x_0\rangle = \langle |\psi\rangle |x_0\rangle \, | \, |\phi\rangle |x_0\rangle \rangle \\ &= \langle U |\psi\rangle |x_0\rangle \, | U |\phi\rangle |x_0\rangle \rangle = \langle |\psi\rangle |\psi\rangle \, | \, |\phi\rangle |\phi\rangle \rangle = \langle\psi|\phi\rangle^2, \end{aligned}$$

therefore $\langle\psi|\phi\rangle \in \{0, 1\}$ are either equal or orthogonal, which is a contradiction, since we assumed they were arbitrary.

□

We can however clone states from an orthogonal basis, and we will make use of this fact for quantum simulation of classical operations.

2. Classical computation models

2.1. What is a problem?

Computability and complexity theory attempt to answer questions regarding how to efficiently solve real-world problems. To formalize the notion of problem, we need to encode the inputs and outputs in an uniform way as words over an alphabet.

Definition 2.1: *An* alphabet *is a finite set. Its elements are called* symbols*.*

A word $w = w_1 \dots w_n$ *over an alphabet A is a finite sequence of symbols over that alphabet. Its length is $|w| := n$. The set of all words over A is denoted $A^\star$.*

Given two words $u = u_1 \dots u_n, v = v_1 \dots v_m \in A^\star$ its concatenation is the word

$$\cdot(u, v) := uv := u_1 \dots u_n w_1 \dots w_m \in A^\star.$$

$(A^\star, \cdot)$ is a monoid.

A language *over an alphabet A is a subset $L \subseteq A^\star$.*

Using these notions one can define a variety of concepts that correspond to different kinds of problems. In this text we will mostly focus on decision problems and promise problems.

Definition 2.2: *A* promise problem *over A is a pair of disjoint languages $(L_{\text{YES}}, L_{\text{NO}})$ over A. The* promise *of a promise problem is the set $L_{\text{YES}} \cup L_{\text{NO}}$.*

A decision problem *is a promise problem $(L_{\text{YES}}, L_{\text{NO}})$ such that $L_{\text{YES}} \cup L_{\text{NO}} = \mathbb{B}^*$. The associated language to a decision problem is L_{YES}.*

In a promise problem we are promised that the input will belong to the promise, and we must decide whether it is a YES-instance or a NO-instance. These kinds of problems do not capture the full general notion of problems such as calculating a function. However, most real-world problems can be translated into a decision problem that captures the complexity of solving it.

Inputs are usually stated in terms of mathematical structures such as integers, graphs, Turing machines or (finitely generated) groups. The encoding as binary words needs to be done in a per-object basis; it will be implicitly assumed what the actual encoding is if it is clear from context.

2.2. Computation models and computability

2.2.1. Turing machines

The classical notion of computability is formalized in the field of computational complexity via the notion of Turing machines. Although Turing machines can be further generalized, for our purposes we will focus on single-tape Turing machines with tape alphabet $\mathcal{T} := \{0, 1, \square\}$ that includes the *blank symbol* $\square$.

Definition 2.3: *A* Turing machine *is a 4-tuple* $M = (Q, \delta, q_0, q_F)$ *such that*

1. Q*, the set of* states*, is a finite set,*
2. $\delta : Q \times \mathcal{T} \to Q \times \mathcal{T} \times \{\mathrm{L}, \mathrm{R}\}$ *is the* transition function*,*
3. $q_0 \in Q$ *is the* initial state*,*
4. $q_F \in Q$ *is the* final state*.*

A configuration *of* M *is a triplet* (q, v, u) *such that* $q \in Q$ *and* $v, u \in \mathcal{T}^*$*. It is* final *if* $q = q_F$*.*

Informally, a Turing machine has an internal state, an infinitely long tape with cells that initially have the blank symbol $\square$ and a head positioned on top of one of the cells. At each step, the machine reads the symbol on the current position and depending on the internal state and the symbol

1. transitions to a new internal state,
2. writes a new symbol on the current position and
3. moves the head to the **L**eft or **R**ight.

We can define two relations between configurations that will allow us to formalize this notion. The idea is that a configuration represents the cells to the left and to the right of the head except for the infinite amount of blank cells at each side.

Definition 2.4: *Given two configurations* C, C' *of the same machine, we say* $C \vdash C'$ *if, if* $C = (q, v, u)$ *then* $C' = (q', v', u')$ *such that* $\delta(q, u_1) = (q', b, D)$ *with* $D \in \{L, R\}$ *and*

a) if $D = L$ *then if* $|v| = n, |u| = m$ *we have* $v' := v_1 \cdots v_{n-1}$ *and* $u' := v_n b u$ *in all cases except*

   i) if $n = 0$ *then* $v' := \varepsilon, u' := \square bu$
   ii) if $m = 1, b = \square$ *then* $v' = v_1 \cdots v_{n-1}, u' := \varepsilon$
   iii) if both conditions are held simultaneously then $v' = \varepsilon, u' = \square$*.*

b) if $D = R$ *then if* $|v| = n, |u| = m$ *we have* $v' := vb$ *and* $u' := u_2 \cdots u_m$ *in all cases except*

   i) if $m = 1$ *then* $v' := vb, u' := \square$
   ii) if $n = 0, b = \square$ *then* $v' = \varepsilon, u' := u_2 \cdots u_m$
   iii) if both conditions are held simultaneously then $v' = \varepsilon, u' = \square$*.*

Given two configurations C, C' of the same machine, we say $C \vdash^* C'$ if there exists a finite sequence $C = C_1 \vdash \cdots \vdash C_n = C'$. The Turing machine then takes n steps *to get from C to C'.*

A configuration may be related with at most one other configuration. We may assume that final configurations are not related to any other configuration. Additionally, we say the Turing machine uses a cell of memory when a symbol different than the initial one has been written for the first time. Finally we have

Definition 2.5: *A Turing machine $M = (Q, \delta, q_0, q_F)$* accepts *$u \in \mathbb{B}^*$ if there exists a final configuration C such that $(q_0, \varepsilon, u) \vdash^* C$.*

A Turing machine $M = (Q, \delta, q_0, q_F)$ computes *$f : \mathbb{B}^* \to \mathbb{B}^*$ if for every word u we have $(q_0, \varepsilon, u) \vdash^* (q_F, v_1, v_2)$ with $f(u) = v_1 v_2$.*

A Turing machine M accepts a language *L if for every word $u \in L$, M accepts u.*

If, when starting with a word on the tape at some point a Turing machine reaches a non-final configuration with no other related configuration, we say the Turing machine *rejects* its input. When it is clear from context, we will also denote by M the partial function computed by the Turing machine M.

2.2.2. The circuit model

The circuit model provides an alternative computation model that will be useful in proving the relation between classic and quantum computation models. We adapt the definition from (Vollmer (1999), chap. 1).

We will use the term circuit in different contexts in this document. In general, there is an implicit base state space (for example $\mathbb{B}, \mathbb{R}$, a random bit or a qubit), that can be composed with a certain product (usually the Cartesian product or the tensor product).[1] Both will be clear from context.

Using these basic pieces, a circuit is made of gates from a certain set (*basis*). A *gate* is a function $f : A^n \to A^m$ from a product of the base state space A to another product. We say that the gate has n inputs and m outputs.

Definition 2.6: *A* circuit *with n inputs $\{x_1, \ldots, x_n\}$ and m outputs $\{y_1, \ldots, y_m\}$ with respect to basis $\mathcal{B}$ is a 3-tuple $C = (G, \beta, \mathcal{B})$ such that*

1. *$G = (V, E)$ is a finite directed acyclic graph,*
2. *$\beta : V \to \mathcal{B} \cup \{x_1, \ldots, x_n\} \cup \{y_1, \ldots, y_m\}$ is a function such that*
   a) *the in-degree and out-degree of a vertex match the number of inputs and outputs of its image by β,*
   b) *each input or output is the image of β in at most one node.*

[1] In its most general terms what we are essentially describing a morphism in a planar monoidal category via a string diagram (Selinger (2010), thm. 3.1).

The size *of a circuit C is $|C| := |V|$.*

We assume a fixed efficient encoding of a circuit (over a finite basis) as a binary string such that

1. the encoding enc is an injective function,
2. the encoding is efficient; there exists a polynomial $p(n) \geq n$ such that

$$|\operatorname{enc}(C)| = p(|C|)$$

A possibility is the representation of the graph as and adjacency matrix.

In the case of classical circuits we allow the following known logical gates

1. the AND gate ($\wedge : \mathbb{B}^2 \to \mathbb{B}$),
2. the OR gate ($\vee : \mathbb{B}^2 \to \mathbb{B}$) and
3. the NOT gate ($\neg : \mathbb{B} \to \mathbb{B}$).

This is not the smallest possible set of gates; it is a well-known fact that they can be reduce to the singleton set $\{\text{NAND}\}$, where $\text{NAND} : \mathbb{B}^2 \to \mathbb{B}$ is given by

$$\text{NAND}(x, y) = \neg(x \wedge y),$$

but, as we shall see in a later section, this is irrelevant for asymptotic measures of size. It is also important that all gates are symmetric.

Furthermore, we will use three operations that allow us to clone, destroy and create bits, namely,

4. the FANOUT gate, given by $\text{FANOUT}(x) = (x, x)$,
5. the 1-input 0-output DISCARD gate, that ignores the output and
6. the 0-input 1-output ANCILLARY gate that produces a 0 bit.

Thus,

Definition 2.7: *A (classical) circuit is a circuit with respect to the basis*

$$\{\text{AND}, \text{OR}, \text{NOT}, \text{FANOUT}, \text{DESTROY}, \text{ANCILLARY}\}$$

Clearly, a (classical) circuit C with n inputs and m outputs computes a function $C : \mathbb{B}^n \to \mathbb{B}^m$. To allow for functions with an input of arbitrary length we need the concept of a *circuit family*.

Definition 2.8: *A (classical)* circuit family *is a sequence $\mathcal{C} = \{C_n\}_{n \in \mathbb{N}}$ such that for every $n \in \mathbb{N}$ C_n is a circuit with n inputs.*

We will mostly restrict ourselves to the case where every circuit in the family has one output. In this case, a circuit family computes a function $\mathcal{C} : \mathbb{B}^* \to \mathbb{B}$ given by $\mathcal{C}(x) =$

$C_{|x|}(x)$. We say $\mathcal{C}$ decides a language L if

$$\mathcal{C}(x) = \begin{cases} 1 & \text{if } x \in L \\ 0 & \text{otherwise} \end{cases}$$

In what follows we will not describe circuits explicitly as 2-tuples, relying instead on graphical representations or high-level descriptions.

2.2.3. Computability

A Turing machine M can be encoded in a canonical way as a binary string w_M, in such a way that there exists a *universal Turing machine* U that computes $U(w_M, x) = M(x)$ (Arora and Barak (2009), thm. 1.9). Furthermore, we denote by $n(M)$ the number associated with the binary string w_M.

It is clear from the previous paragraph that the set of Turing machines is countable, since the function $M \mapsto n(M)$ is injective. On the other hand, the set of all languages $\mathcal{P}(\mathbb{B}^*)$ is uncountable by Cantor's theorem. Therefore there are languages that are *undecidable*, that is, that are not the language associated with a Turing machine.

The following example showcases one undecidable language that is the language associated with a circuit family. This shows the power difference between the two models of computation.

Proposition 2.1: *Let* $L_{\text{HALT}} = \{1^{n(M)} : M \notin L(M)\}$. *Then*

1. L_{HALT} *is undecidable.*
2. *For any* L *such that every* $x \in L$ *is of the form* $x = 1^n$ *(such as* L_{HALT}*) there exists a circuit family* $\mathcal{C}$ *of linear size such that* $L(\mathcal{C}) = L$.

Proof.

1.

Assume that L_{HALT} is decidable, that is, that there exists a Turing machine M such that $L(M) = L$.

Let M' be a Turing machine such that

> On input x, M' computes $M(1^{n(M')})$. If accepted then reject, otherwise accept.

Then we arrive at a contradiction since

- If $M' \in L(M)$ then by the definition of M, $M' \in L(M')$. But in that case, $L(M') = \varnothing$, thus $M' \notin L(M')$.
- If $M' \notin L(M)$ then by the definition of M, $M' \notin L(M')$. But in that case, $L(M') = \mathbb{B}^*$, thus $M' \in L(M')$.

Therefore M does not exist.

2.

Let $n \in \mathbb{N}$. Let C_n be a linear size circuit that computes

a) the constant function o if $1^n \notin L$ or
b) the function $C_n(x_1 \dots x_n) = x_1 \wedge \dots \wedge x_n$ otherwise.

The latter circuit can be computed by building a tree of $\wedge$s of logarithmic height. The family $\{C_n\}_{n\in\mathbb{N}}$ then decides L.

□

2.3. Complexity

2.3.1. Deterministic complexity

There are different approaches as to how to measure the complexity of a given algorithm, here we will focus on two such notions: the worst-case time and space complexity for Turing machines and the number of gates per input size in the case of circuits.

Definition 2.9: *Let $g \in \mathbb{N}^{\mathbb{N}}$.*

$$O(g) = \left\{ f \in \mathbb{N}^{\mathbb{N}} \ : \ \exists N, C \in \mathbb{N} : \quad f(n) \leq Cg(n) \quad \forall n > N \right\}$$

Definition 2.10: *Let $f : \mathbb{N} \to \mathbb{N}$, $L \subseteq \mathbb{B}^*$. Then:*

1. *$L \in \mathrm{TIME}(f(n))$ if there exists a Turing Machine M such that $L(M) = L$ and M takes $g(n) \in O(f(n))$ steps to decide any $x \in \mathbb{B}^*$ with $|x| = n$.*
2. *$L \in \mathrm{SPACE}(f(n))$ if there exists a Turing Machine M such that $L(M) = L$ and M writes $g(n) \in O(f(n))$ cells to decide any $x \in \mathbb{B}^*$ with $|x| = n$.*
3. *$L \in \mathrm{SIZE}(f(n))$ if there exists a circuit family $\mathcal{C}$ over the basis B such that $L(\mathcal{C}) = L$ and $|C_n| \in O(f(n))$.*

If $F \subseteq \mathbb{N}^{\mathbb{N}}$ then

$$\mathrm{TIME}(F) = \bigcup_{f(n)\in F} \mathrm{TIME}(f(n))$$

and likewise for SPACE and SIZE.

These classes are potentially dependent on the use of the specific single-tape Turing machine model of Definition 2.5; if a different model is used, such as multi-tape Turing machines or RAM models, the classes might change (van Leeuwen (1990), chap. 1).

2.3.2. Non-deterministic complexity

Analogous to the concept of Turing machine, a prevalent idea in complexity theory is the use of non-deterministic models of computation. These models will be useful in that their associated complexity classes encapsulate several problems of practical importance and they provide upper-bounds for deterministic classes.

Intuitively, a non-deterministic Turing machine can be simultaneously in several configurations and thus have multiple computation paths. Depending on the imposed criteria these computation paths can be interpreted in terms of probability or a more general non-determinism.

Definition 2.11: *A* non-deterministic Turing machine *(NDTM) is a 4-tuple* $M = (Q, \delta, q_0, q_F)$ *such that*

1. Q, *the set of* states, *is a finite set,*
2. $\delta \subseteq (Q \times \mathcal{T}) \times (Q \times \mathcal{T} \times \{\mathrm{L}, \mathrm{R}\})$ *is the* transition relation,
3. $q_0 \in Q$ *is the* initial state,
4. $q_F \in Q$ *is the* final state.

A configuration *of* M *is a triplet* (q, v, u) *such that* $q \in Q$ *and* $v, u \in \mathcal{T}^*$. *It is* final *if* $q = q_F$.

An identical relation to that of Turing machines exists between configurations of NDTM. In this case each configuration may be related to more than one other configuration, which is what gives rise to non-determinism. Using this assumed notions we define when a NDTM accepts a language in an analogous way to Turing machines. The number of steps and memory usage is given by the maximum steps (resp. memory cells) used by any computation path. This gives rise to the classes $\mathrm{NTIME}(f(n))$ and $\mathrm{NSPACE}(f(n))$ for languages that can be decided by a NDTM using $O(f(n))$ time or space respectively.

The possible computation paths on a given execution of a non-deterministic Turing machine can be seen as a tree with vertices labeled by the current configurations. Every tree of computation paths of the same NDTM will have a maximum *branching factor* that can be calculated from the transition relation.

For stating the relations between the different notions of complexity we will make use of the following technical lemma, that we present without proof.

Lemma 2.2 (Oblivious Turing machine)**:** *(Arora and Barak (2009), remark 1.7)*

For any $f(n)$*-time Turing machine* M *there exists a* $f(n) \log f(n)$*-time Turing machine* $\widetilde{M}$ *that decides the same language and such that for every input* $x \in \mathbb{B}^*$ *and* $i \in \mathbb{N}$ *the location of the head of* M *at the* i*th step of execution is a function of* $|x|$ *and* i.

Using this lemma we can prove

Proposition 2.3: *The following inequalities hold for every* $f \in \mathbb{N}^{\mathbb{N}}$

1. $\text{TIME}(f(n)) \subseteq \text{SIZE}(f(n)\log f(n))$,
2. $\text{TIME}(f(n)) \subseteq \text{NTIME}(f(n))$ *and* $\text{SPACE}(f) \subseteq \text{NSPACE}(f)$,
3. $\text{NTIME}(f(n)) \subseteq \text{SPACE}(f(n))$ *ad*
4. *if* $f(n) \geq \log(n)$ *then* $\text{NSPACE}(f(n)) \subseteq \text{TIME}(2^{f(n)})$.

Proof.

1. Let $L \in \text{TIME}(f(n))$ and M a $f(n)$-time Turing Machine. By Lemma 2.2 there exists a $g(n) = f(n)\log f(n)$-time oblivious Turing machine M' that decides the same language.

   On input x, the configuration of M' at step n can be encoded as a constant-size string. Furthermore, the transition from configuration C to configuration C' can be computed with a constant-sized circuit with inputs C and the configurations that were in the same position as configuration C', that, since M' is oblivious, depend only on the size of the input. The composition of these circuits gives rise to a $g(n)$-sized circuit family that computes the same language as M.

2. Clearly, every (deterministic) Turing machine can be transformed into a non-deterministic one by transforming the transition function into a transition relation. The running time and space will be identical and thus we have both inclusions

3. Let $L \in \text{NTIME}(f(n))$ and M the associated NDTM. Let d be the maximum branching factor of M. Any sequence of choices can be written in $O(f(n)\log d)$ space as a sequence of $O(f(n))$ numbers between 0 and d.

   For every possible sequence of choices, try the execution of M and accept if M accepts. If M rejects for any of these sequences, then reject.

4. Let $L \in \text{NSPACE}(f(n))$, $M = (Q, \delta,)$ the associated NDTM. The set of possible configurations of M has size $|Q|2^{f(n)}(n+1) \in O(2^{f(n)+\log(n)}) = O(2^{f(n)})$.

   Checking $x \in L$ is equivalent to checking if there is a path between the initial configuration and a final configuration, the graph of possible configurations, where two configurations are connected if they are related by $\vdash$. This can be done in quadratic time, which proves the result.

□

2.3.3. Polynomial usage of resources

A polynomial amount of resources is informally described as an efficient usage of them. An algorithm can be informally described as *efficient* if it takes a polynomial amount of time to run as a function of its input length and we say it makes an *efficient usage of space* if it uses a polynomial amount of space in its execution.

This definition does not precisely match the everyday definition of "efficient"; for practical purposes an algorithm that takes n^{1000} steps to decide a word of size n is not considered efficient and an algorithm with running time $n^{\log\log n}$ is (Aaronson (2016a), sec. 1.2.1).

Nonetheless, the interpretation of the following classes as containing efficiently decidable languages makes sense since these are the minimal classes closed under composition, concatenation and union that contain linear-time decidable languages (or respectively space or size) (Cobham (1965)). Additionally, these classes are robust under small changes in the computational models used to define them (e.g. by using multi-tape machines or RAM models) (van Leeuwen (1990), chap. 1).

Definition 2.12: *Let* $\mathrm{poly}(n) = \left\{P \in \mathbb{N}^{\mathbb{N}} \ : \ P \text{ is a polynomial}\right\}$. *Then*

- $\mathsf{P} := \mathrm{TIME}(\mathrm{poly}(n))$,
- $\mathsf{NP} := \mathrm{NTIME}(\mathrm{poly}(n))$,
- $\mathsf{PSPACE} := \mathrm{SPACE}(\mathrm{poly}(n))$,
- $\mathsf{NPSPACE} := \mathrm{NSPACE}(\mathrm{poly}(n))$ *and*
- $\mathsf{P/poly} := \mathrm{SIZE}(\mathrm{poly}(n))$.

Clearly, by Proposition 2.3, $\mathsf{P} \subseteq \mathsf{NP} \subseteq \mathsf{PSPACE} \subseteq \mathsf{NPSPACE}$. The first inequality is conjectured to be strict and is the most important problem in the field (Aaronson (2016a) sec. 2.2.5), given the wide range of practical problems that are in NP. The last inequality is in fact an equality given by *Savitch's theorem* (Arora and Barak (2009), thm. 4.14).

As seen in Proposition 2.1 P/poly contains languages that are uncomputable by the classic notion of computation, thus $\mathsf{P} \subsetneq \mathsf{P/poly}$, and some restriction is needed to use the circuit model for stating complexity results.

The following definition and result gives an alternative characterization of P in terms of families of circuits.

Definition 2.13: *A circuit family C is said to be* polynomial-time uniform *if there exists a Turing machine M that computes the function $1^n \mapsto \mathrm{enc}\, C_n$ in polynomial time.*

Clearly, a polynomial-time uniform circuit family has polynomial size, since it could not be written as the output otherwise.

Proposition 2.4: *Let L be a language. Then*

1. *$L \in \mathsf{P}$ if and only if L is decidable by some polynomial-time uniform circuit family.*
2. *$L \in \mathsf{P/poly}$ if and only if there exists a Turing machine M and a sequence of words w_n of polynomial size such that $x \in L$ if and only if M accepts $x\square w_{|x|}$.*

Proof.

1.

$\Longrightarrow$) The circuit family described in Proposition 2.3 1 is polynomial-sized if M is polynomial-time. Additionally, it can clearly be computed by a Turing machine simulating the execution of M on 1^n and constructing the circuit as it goes.

$\Leftarrow$) Let $\{C_n\}_{n\in\mathbb{N}}$ be a polynomial-time uniform circuit family that decides L. Then let M be the Turing machine given by the following description

> On input x write $1^{|x|}$ and compute $C_{|x|}$. Finally compute $C_{|x|}(x)$ and accept depending on the outcome.

Clearly $1^{|x|}$ can be written on linear time and, since, $\{C_n\}_{n\in\mathbb{N}}$ is polynomial-time uniform, $C_{|x|}$ can be calculated in polynomial time. Computing $C_{|x|}(x)$ implies computing the outcome at each vertex which can be done in constant time. The whole computation takes polynomial time since the circuit family has polynomial size.

2.

$\Longrightarrow$) Since $L \in \mathsf{P}/\mathrm{poly}$, take as the sequence of words w_n the encoding of the polynomial-size circuit family C_n that decides L. Since C_n has polynomial size, w_n has polynomial size.

Then, let M be the Turing machine that on input $x\square w_n$ output $C_n(x)$. As in the proof of **1**, one can do this in polynomial time.

$\Leftarrow$) By Proposition 2.3 1 we know that there exists a polynomial-size sequence of circuits C'_n such that C'_n has $n + |w_n|$ inputs and $C'_n(x, w_n) = M(x\square w_n)$.

Let C_n be the circuit C'_n with input w_n fixed. C_n has polynomial-size on the variable $m = n + |w_n|$ and thus has polynomial size with respect to n.

□

2.3.4. Polynomial time verifiers

An important notion in the field of computational complexity is the notion of *verifiers*. Potentially, a language might not be able to be decided in polynomial time, yet one can decide membership when supplied with a *proof*. Here we show a characterization of NP as a verifier class.

Proposition 2.5: *Let $L \subseteq \mathbb{B}^*$. Then $L \in \mathsf{NP}$ if and only if there exists a Turing Machine V (*verifier*) that takes polynomial time to execute with respect to the length of its first argument and a polynomial $p(n) \in \mathrm{poly}(n)$ such that*

$x \in L$ if and only there exists $y \in \mathbb{B}^*$ with $|y| \leq p(|x|)$ such that $V(x, y) = 1$.

Proof.

$\Longrightarrow$) Let $M = (Q, \delta, q_0, q_F)$ be a polynomial-time non-deterministic Turing machine for L. As in the proof of Proposition 2.3, any sequence of choices can be written in a polynomial-length word as a sequence of numbers between 0 and the maximum branching factor.

We can then let V be the verifier given by the following description

> On input x, y run M with input x. Whenever a pair (q, a) is related with more than one triplet by δ, choose among the triplets at step i by using the ith number of y. Output 1 if and only M accepts.

Since M runs in polynomial time V runs in polynomial time. Furthermore, if and only if $x \in L$ there exists a set of choices y that leads to an accepting path.

$\Longleftarrow$) Let V be a verifier for L and p its associated polynomial. Then let M be the Turing machine with the following description

> On input x non-deterministically select a string y of length $|y| \leq p(|x|)$. Accept if $V(x, y) = 1$.

The selection of a string is done in polynomial time since writing each symbol takes one step. By hypothesis, V runs in polynomial time on x, thus M is a non-deterministic polynomial-time Turing machine.

□

3. Probabilistic computation models

A simple generalization of classical models of computation is to consider the addition of randomness. The traditional focus is on probabilistic Turing machines, yet here we focus on probabilistic circuits since they are more similar to quantum circuits.

3.1. Circuits

3.1.1. The state space

Probabilistic circuits generalize classical circuits by considering "stochastic" gates, that is, gates that given a valid distribution on the inputs, output a valid distribution on the outputs.

The state space they modify can be represented by a real vector space spanned by a basis $\{|0\rangle, |1\rangle\}$. We call such space R.

Definition 3.1: *A* random bit *is a vector*

$$p = a|0\rangle + b|1\rangle$$

of the real vector space R spanned by a basis $\{|0\rangle, |1\rangle\}$, such that

$$a, b \geq 0 \text{ and } \|p\|_1 = a + b = 1.$$

As was the case on quantum mechanics, to consider the state space of a composite system we tensor the state spaces of the subsystems. Similarly, a state can be *measured*

Definition 3.2: *A measurement of a probabilistic system of dimension $N = 2^n$ with state*

$$p = \sum_{i=0}^{N-1} a_i |i\rangle$$

is a discrete random variable X such that

$$P(X = |i\rangle) = a_i \qquad (i = 1, \ldots, N)$$

Lastly, we state the allowed operations. Even though we will restrict probabilistic circuits to a small set of gates, we will describe here the most general possible operations that

can be made by composing these gates.

Definition 3.3: *Let $n, m \in \mathbb{N}$. A* stochastic gate *is a linear map $f : R^{\otimes n} \to R^{\otimes m}$, represented by a* stochastic matrix*, that is, a matrix such that*

1. *every entry is non-negative and*
2. *every column adds up to 1.*

When considered as a gate, n is the number of ***inputs*** *and m is the number of* ***outputs****.*

Some simple examples of stochastic gates can be given.

Example 3.1:
- *Let $f : \mathbb{B}^n \to \mathbb{B}^m$ be classical gate. A stochastic gate can be formed by considering the linear extension of f, $\widetilde{f} : R^{\otimes n} \to R^{\otimes m}$, that is,*

$$\widetilde{f}\left(\sum_{i=0}^{2^n-1} a_i |i\rangle\right) = \sum_{i=0}^{2^n-1} a_i |f(i)\rangle .$$

  Its matrix is a permutation matrix.
- *The random gate* $\mathrm{RANDOM}(p) : R^{\otimes 0} \to R$ *outputs a random bit, $p|0\rangle + (1-p)|1\rangle$).*
- *If f is a stochastic gate, a controlled version of it can be considered analogously to the quantum case (see Proposition 1.5).*

Every stochastic gate can be seen as transforming from one distribution to another.

Proposition 3.1: *Let $p \in R^{\otimes n}$ such that $\|p\|_1 = 1$ and all entries are non-negative, and let $f : R^{\otimes n} \to R^{\otimes m}$ be a stochastic gate. Then $f(p) \in R^{\otimes m}$ has all entries non-negative.*

Proof.

Let A be the matrix associated with f with respect to the usual basis. Then

$$f(p) = Ap = \left(\sum_{j=1}^{2^n-1} A_{ij} p_j\right)_{i=1,\ldots 2^n-1}$$

Every entry of A and p are non-negative, so $\sum_{i=1}^{2^n-1} A_{ij} v_j \geq 0$ for every i. Lastly,

$$\|Ap\|_1 = \sum_{i=1}^{2^n-1}\sum_{j=1}^{2^m-1} A_{ij} v_j = \sum_{j=1}^{2^m-1} v_j \left(\sum_{i=1}^{2^n-1} A_{ij}\right) = \sum_{j=1}^{2^m-1} v_j = 1.$$

□

3.1.2. Probabilistic circuits

Clearly, we can decompose any stochastic gate into a classical gate and a number of ancillary random bits, that is, RANDOM(p) gates. In practice, not every source of randomness is feasible, since we could encode in the digits an uncomputable function.

Definition 3.4: *A real number $r \in \mathbb{R}$ is a* computable number *if the function that given n outputs the n-th bit of r can be computed.*

r is polynomial time computable *if this function can be computed in time polynomial in n.*

We restrict the numbers on stochastic gates to be computable so as to avoid this situation. This restriction is not too strict though since the set of computable numbers is dense. Provided we follow that restriction, the following result shows us that we can restrict to single source of randomness with a fixed bias.

Proposition 3.2: *Let $p, q \in]0,1[$ be polynomial time computable numbers.*

A random source X with $P[X = 1] = p$ can be simulated by a random source Y with $P[Y = 1] = q$ on expected $O(\frac{1}{q(1-q)})$ time.

Proof.

The proof is adapted from (Arora and Barak (2009), lemma 7.12 and lemma 7.13).

Consider the case where $q = \frac{1}{2}$, that is, we have a uniform distribution. Let $p = \sum_{i=1}^{\infty} p_i 2^{-i}$. Consider the following algorithm

1. Let $i = 0$
2. Increase i
3. Generate a random bit b_i
4. If $b_i < p_i$, stop and output 1. Else, if $b_i > p_i$, stop and output 0.
5. If $b_i = p_i$, go back to step 2.

We reach the i-th run with probability 2^{-i}, thus the probability of outputting 1 is exactly $\sum_{i=1}^{\infty} p_i 2^{-i} = p$.

The expected running time of the algorithm is $\sum i^c 2^{-i}$ for a certain constant c (that depends on the algorithm that computes the bits of p_i). This series is convergent, thus the simulation takes constant time.

Consider now the case where $p = \frac{1}{2}$ and run the following algorithm:

1. Generate two random bits a, b
2. If $a = b$, go back to step 1.
3. Output a.

Conditioned on $a \neq b$, the two outcomes occur with the same probability, $q(1-q)$. Each time two bits are generated, the probability of reaching step 3 is $2q(1-q)$, thus the running time is $O(1/(q(1-q)))$.

By joining the two previously discussed cases we have the desired algorithm.

□

By using this result, we can approximate any random source gate up to accuracy $\varepsilon > 0$ with a $O(\log(1/\varepsilon))$ sized circuit. Thus, it makes sense to consider the following definition of probabilistic circuit:

Definition 3.5: *A probabilistic circuit is a circuit respect to the basis*

$$\{\text{NAND}, \text{FANOUT}, \text{RANDOM}(1/2)\}$$

Here the base state space is R and the product is the tensor product. A probabilistic circuit C has, for each x an associated random variable $C(x)$ obtained by running the circuit with input x and uniformly random inputs on the RANDOM$(1/2)$ gates.

A mixed model that considers stochastic and classical gates might be considered with the addition of a measurement gate. This model turns out to be equivalent to the one we defined: any intermediate measurement can be simulated by a controlled gate. We encapsulate this fact in the following principle:

Principle 5 (Principle of deferred measurements)**:** *(Nielsen and Chuang (2010), sec. 4.4)*

Any circuit C with n inputs that has intermediate measurements can be transformed into an algorithm C' with an identical associated function by replacing intermediate measurements by controlled gates.

In what follows, we will describe probabilistic circuits as randomized algorithms (see Proposition 3.7 for a formalization of this fact) and we will make intermediate measurements if it simplifies the presentation.

Lastly, analogous to the classical case, we define the concept of probabilistic computability.

Definition 3.6: *A function $f : \mathbb{B}^* \to \mathbb{B}^*$ is probabilistic $T(n)$-computable if there exists a uniform family of probabilistic circuits $\mathcal{C} = \{C_n\}$ computable in $O(T(n))$ time such that for all $x \in L$,*

$$P[C(x) = f(x)] \geq \frac{2}{3}$$

The constant $\frac{2}{3}$ is arbitrary and can be replaced by any constant $c \in]\frac{1}{2}, 1[$ with at most polynomial overhead, as the following proposition shows.

Proposition 3.3 (Chernoff bound)**:** *(Nielsen and Chuang (2010), Box 3.4)*

Let $\varepsilon > 0, n \in \mathbb{N}, p = \frac{1}{2} + \varepsilon$ *and* $X_1, \ldots, X_n \sim \text{Bernouilli}(p)$ *independent identically distributed random variables. Then*

$$P\left[\sum_1^n X_i \leq \frac{n}{2}\right] \leq \exp(-2\varepsilon^2 n)$$

Proof.

Let $x_i \sim X_i$ such that among $x_1, \ldots, x_n$ there are at most $n/2$ ones. Since $p > \frac{1}{2}$, the probability mass function p of one such sequence is maximized when there are exactly $\lfloor n/2 \rfloor$ ones, that is, since they are independent, p is bounded above by

$$\begin{aligned} p(x_1 = X_1, \ldots, x_n = X_n) &= \prod_{i=1}^n p(x_i = X_i) = \prod_{i=1}^n p^{x_i}(1-p)^{1-x_i} \\ &\leq \left(\frac{1}{2} - \varepsilon\right)^{\frac{n}{2}} \left(\frac{1}{2} + \varepsilon\right)^{\frac{n}{2}} = \frac{(1-4\varepsilon)^{n/2}}{2^n}. \end{aligned}$$

There are at most 2^n sequences of that kind, therefore,

$$P\left[\sum_1^n X_i \leq \frac{n}{2}\right] \leq 2^n \cdot \frac{(1-4\varepsilon)^{n/2}}{2^n} = (1-4\varepsilon)^{n/2}.$$

Lastly, by the Taylor expansion of the exponential we have $1 - x \leq e^{-x}$, which proves the result,

$$P\left[\sum_1^n X_i \leq \frac{n}{2}\right] \leq \exp(-4\varepsilon^2 n/2) = \exp(-2\varepsilon^2 n).$$

□

3.2. Probabilistic polynomial complexity

The languages that are feasibly decided by probabilistic algorithms form the class BPP (*bounded probabilistic polynomial*):

Definition 3.7: $L \in \text{BPP}$ *if and only if* 1_L *is a probabilistic polynomial time computable function, that is, there exists a probabilistic polynomial time algorithm* M *such that*

1. *for every* $x \in L$, $P[M(x) = 1] > \frac{2}{3}$ *and*
2. *for every* $x \notin L$, $P[M(x) = 1] < \frac{1}{3}$.

As we saw in Proposition 3.3, we can make the bounds as close to 1 as we want and we will have the same class of languages. Another possibility is to consider unbounded

probabilistic algorithms, that are given by the class PP.

Definition 3.8: *$L \in \mathsf{PP}$ if and only if there exists a probabilistic polynomial time algorithm M such that*

1. *for every $x \in L$, $P[M(x) = 1] > \frac{1}{2}$ and*
2. *for every $x \notin L$, $P[M(x) = 1] \leq \frac{1}{2}$.*

In contrast to BPP, PP is considered unfeasible, as the following proposition hints at.

Proposition 3.4: *(Katz 2005)*

$$\mathsf{NP} \subseteq \mathsf{PP}$$

Proof.

Let $L \in \mathsf{NP}$ and $x \in \mathbb{B}^*$.

By Proposition 2.5, there exist a verifier V and a polynomial p such that $x \in L$ iff there exists $y \in \mathbb{B}^{p(|x|)}$ with $V(x, y) = 1$.

The following PP algorithm M decides L.

Accept x with probability $\frac{1}{2} - 2^{-p(|x|)-2}$. Otherwise, pick a random $y \in \mathbb{B}^{p(|x|)}$ and accept if $V(x, y) = 1$.

If $x \notin L$, then

$$P[M(x) = 1] = \frac{1}{2} - 2^{-p(|x|)-2} < \frac{1}{2}.$$

Otherwise, if $x \in L$ then $P[V(x, y) = 1] \geq 2^{-p(|x|)}$, thus

$$P[M(x) = 1] \geq \frac{1}{2} - 2^{-p(|x|)-2} + 2^{-p(|x|)} > \frac{1}{2}.$$

Therefore, M decides L. M runs in polynomial time since V runs in polynomial time on its first input.

□

As the proof of Proposition 3.4 shows, the difference between PP and BPP lies in the possibility of applying Proposition 3.3. The probability bounds of the algorithm stated in this proof can not be amplified efficiently by applying that procedure. The following relations hold between classical and probabilistic classes.

Proposition 3.5:

$$\mathsf{P} \subseteq \mathsf{BPP} \subseteq \mathsf{PP} \subseteq \mathsf{PSPACE}$$

Proof.

The first three inclusions are clear.

For the last one, that is, $\mathsf{PP} \subseteq \mathsf{PSPACE}$, let $L \in \mathsf{PP}$. There exists a uniform family of probabilistic circuits that computes PP.

Let $x \in \mathbb{B}^*$, $n = |x|$. Consider C_n and replace every RANDOM($\frac{1}{2}$) gate by an input (called random inputs). There is a polynomial amount of such random inputs, $p(n)$.

For every possible word $y \in \mathbb{B}^{p(n)}$ run the circuit with input x and random inputs y and count the number of accepting runs. If the number is over $2^{p(n)}/2$ accept x, otherwise reject.

Clearly, the algorithm accepts if and only if $P[\mathcal{C}(x) = 1] > \frac{1}{2} \iff x \in L$.

Lastly, the algorithm runs in polynomial space: we need to store the current random inputs y, the auxiliary space needed to simulate the circuit and $\lceil \log(2^{p(n)}) \rceil \in \text{poly}(n)$ space needed to count the number of accepting paths.

□

P and BPP are conjectured to be equal, that is, every probabilistic algorithm could be *derandomized* into a classical algorithm.

Lastly, we can show the following relation between BPP and non-uniform classes,

Proposition 3.6:

$$\mathsf{BPP} \subseteq \mathsf{P/poly}$$

Proof.

(Arora and Barak (2009), thm. 7.14)

Let $L \in \mathsf{BPP}$ and let $\{C_n\}$ be the polynomial sized uniform probabilistic circuit family that decides L.

By Proposition 3.3 we have that we can construct a circuit family such that for any $n \in \mathbb{N}$ and $x \in \mathbb{B}^n$,

$$P[C_n(x) \neq 1_L(x)] \leq 2^{-n-1}.$$

The circuit now has $p(n) \in \text{poly}(n)$ random values (given by the RANDOM($\frac{1}{2}$) gates). By the probability bound, the amount of possible random inputs $r \in \mathbb{B}^{p(n)}$ that misclassify x are at most $\frac{2^{p(n)}}{2^{n+1}}$.

The amount of words of length n is 2^n, therefore the amount of random inputs that misclassify at least one word is $2^n \cdot \frac{2^{p(n)}}{2^{n+1}} = 2^{p(n)-1}$.

Hence, there exists at least one random input r_n that correctly classifies every word or length n. Finally, consider the circuit family C'_n where the RANDOM$(\frac{1}{2})$ are replaced by ancillary gates that have the values given by r_n.

This circuit family is classical, polynomial sized and correctly decides L, therefore $L \in \mathsf{P}/\text{poly}$.

□

3.2.1. Probabilistic computation with Turing machines

The traditional presentation of probabilistic computation using Turing machines is given by the following proposition.

Proposition 3.7: *$L \in \mathsf{BPP}$ if and only if there exists a Turing machine M that takes polynomial time to execute with respect to the length of its first argument and a polynomial $p(n) \in \text{poly}(n)$ such that for every $x \in \mathbb{B}^*$, $|x| = n$,*

$$P[M(x,r) = 1_L(x)] \geq \frac{2}{3},$$

where $r \sim U(\mathbb{B}^{p(n)})$.

Proof.

Let $L \in \mathsf{BPP}$ and let $\{C_n\}$ be the polynomial sized uniform probabilistic circuit family that decides L. Let $p(n)$ be the amount of RANDOM$(1/2)$ gates on the circuit C_n.

Consider the following algorithm. On input x, r,

1. construct $C_{|x|}$,
2. replace every RANDOM$(1/2)$ gate with the corresponding bit on r,
3. simulate the modified circuit with input x and
4. output the result.

Clearly, the algorithm takes polynomial time. Furthermore, by the definition of BPP, the probability bound is achieved.

□

This model and the *probabilistic Turing Machine* model, that includes a tape with uniform random bits are equivalent in power to the probabilistic circuit model presented in the previous section (Arora and Barak (2009)). It justifies the informal description of probabilistic circuit families by randomized algorithms.

3.2.2. A BPP language

The general consensus among theoretical computer scientists is that P = BPP. Current attempts try to prove this equality by the use of *pseudorandom generators*, that transform a logarithmic number of random bits into a polynomial amount of almost-random bits. This suggest a simple way of transforming a randomized algorithm into a classical one: feed every possible logarithmic seed to the pseudorandom generator and run the randomized algorithm with the output. Unfortunately, this remains an open problem and there are problems that resist *derandomization* (Arora and Barak (2009), chap. 21).

In this section we show a simple decision problem known to be in BPP but not known to be in P, that is, no direct classical algorithm is known to solve this problem in polynomial time.

The problem is called *polynomial identity testing* and can be formally stated by making use of algebraic circuits, which essentially describe a polynomial expression. We follow the approach of (Saxena (2014)).

Definition 3.9: *An* algebraic circuit *is a circuit with one output with respect to the basis* $\{+, -, \times, \text{FANOUT}\}$.

The set of algebraic circuits is $\mathcal{A}$.

Here the state space is an arbitrary field $\mathbb{F}$ (or more generally a ring) and the product between state spaces is the Cartesian product. We assume that the field operations can be computed in constant time as it is the case for example, on a finite field $\mathbb{F}_p$[1].

Clearly, the function associated with an algebraic circuit A is a (multivariate) polynomial p_A. The output of the function for a given input can be computed in polynomial time, though obtaining the polynomial coefficients can in principle take exponential time (since its degree can be exponential on the number of gates).

Definition 3.10:

$$\text{ZEROP} = \{A \in \mathcal{A} \ : \ p_A \equiv 0\}$$

Solving the decision problem associated with ZEROP allows us to solve whether two polynomial expressions are equivalent, since we can reduce this problem to checking that their difference is the zero function.

The straightforward approach of obtaining the polynomial coefficients proves ZEROP $\in$ EXP but not ZEROP $\in$ P, since an algebraic circuit might have an exponential number of coefficients.

[1] Although this is a common assumption in the study of PIT it is not trivial. If for example we want to apply the algorithm to circuits over $\mathbb{Z}$ further considerations are needed to ensure the algorithm is efficient, since the binary representation of intermediate calculations might have exponential size. See (Arora and Barak (2009), sec 7.2.3) for a possible approach.

Surprisingly, a polynomial probabilistic algorithm can be given for this problem. The algorithm is based on the following lemma.

Lemma 3.8 (Schwartz-Zippel Lemma)**:** *(Saxena (2014), lemma 1.2)*

Let $p \in \mathbb{F}[X_1, \ldots, X_m]$ be a nonzero polynomial of (total) degree d and $S \subseteq \mathbb{F}$ a finite set. Then, if $a_1, \ldots, a_m$ are sampled independently and uniformly from S,

$$P[p(a_1, \ldots, a_m) \neq 0] \geq 1 - \frac{d}{|S|}$$

Proof.

The proof is by induction on the number of variables.

In the **base case** p is an univariate polynomial. p has a maximum of d roots and thus the probability of finding a root, i.e $P[p(a_1) = 0]$ is at most $\frac{d}{|S|}$.

For the **inductive case**, assume the lemma holds for any polynomial of total degree less than d. We may consider p as a polynomial on its first variable, $p \in \mathbb{F}[X_2, \ldots, X_m][X_1]$, where the coefficients are now polynomials p_i in the rest of variables.

$p \neq 0$, so let p_k be the polynomial coefficient accompanying the largest X_1 power.

Since the total degree of p is d, it follows that the total degree of p_k is at most $d - k$. By the inductive hypothesis, since $d - k < d$ we have that

1. if $a_2, \ldots, a_m$ are sampled independently and uniformly from S,

$$P[p_k(a_2, \ldots, a_m) \neq 0] \geq 1 - \frac{(d-k)}{|S|} \text{ and}$$

2. if $p_k(a_2, \ldots, a_m) \neq 0$, then $p(X_1, a_2, \ldots, a_m)$ is a univariate polynomial and by the base case,

$$P[p(a_1, \ldots, a_m) \neq 0 | p_k(a_2, \ldots, a_m) \neq 0] \geq 1 - \frac{k}{|S|}.$$

By joining the previous inequalities we have

$$P[p(a_1, \ldots, a_m) \neq 0] \geq \left(1 - \frac{k}{|S|}\right)\left(1 - \frac{d-k}{|S|}\right) \geq 1 - \frac{d}{|S|}$$

□

Using this lemma, we can prove the following proposition

Proposition 3.9:

$$\text{ZEROP} \in \mathsf{BPP}$$

Proof.

The polynomial associated with a circuit A has a total degree of at most $2^{|A|}$ and at most $|A|$ inputs. We call m the number of inputs of A.

The randomized algorithm is as follows:

1. Sample m elements $a_1, \ldots, a_m$ from a finite subset S such that $|S| > 3 \cdot 2^{|A|}$. In the case of finite fields, we might need to consider a field extension that has enough elements (for example by working over a cyclotomic extension).
2. Evaluate $p_A(a_1, \ldots, a_n)$. This can be done in a polynomial amount of field operations.
3. If the previous result is zero, accept, otherwise, reject.

Clearly, if $A \in$ ZEROP we always accept. If $A \notin$ ZEROP, by Lemma 3.8

$$P[p_A(a_1, \ldots, a_n) \neq 0] \geq 1 - \frac{2^{|A|}}{|S|} > \frac{2}{3}.$$

Thus ZEROP $\in$ BPP.

□

3.3. Probabilistic polynomial time verifiers

A natural generalization of NP is to consider the possibility of randomized verifiers. This gives rise to the class MA (*Merlin-Arthur*).

Definition 3.11: *$L \in$ MA if and only if there exists a polynomial $p(n)$ and a polynomial time probabilistic algorithm V such that*

2. *for every $x \in L$, $|x| = n$, there exists $y \in \mathbb{B}^*$ with $|y| \leq p(n)$ (the* proof*) such that*

$$P[V(x, y) = 1] \geq \frac{2}{3} \text{ and}$$

3. *for every $x \notin L$, $|x| = n$, and every $y \in \mathbb{B}^*$ with $|y| \leq p(n)$*

$$P[V(x, y) = 1] \leq \frac{1}{3}.$$

Clearly, NP, BPP $\subseteq$ MA. Furthermore, MA $\subseteq$ PP by a similar proof to the one given in Proposition 3.4.

Unfortunately, there are no known natural problems that lie in MA and are not in BPP $\cup$ NP. By a similar derandomization procedure to the one given for the P $=$ BPP conjecture, the class is believed to be equal to NP (Aaronson (2016a)). Nonetheless, this

class provides an intermediate step between classical and quantum proofs, that will be discussed in a later chapter.

The hierarchy of complexity classes discussed so far can be seen at fig. 3.1.

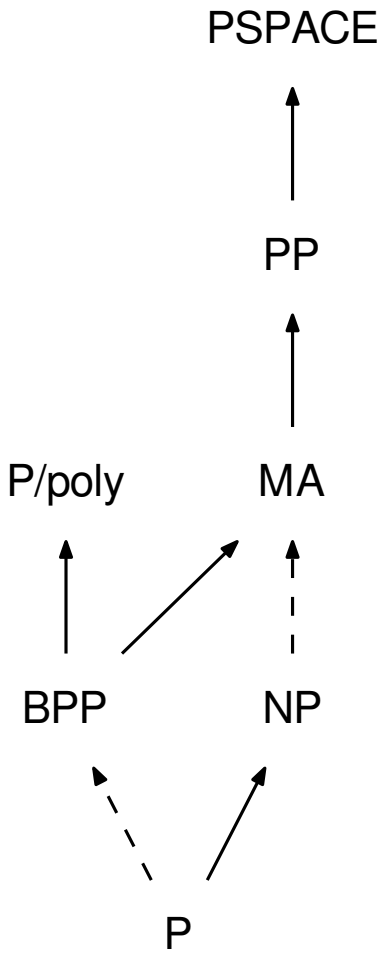


Figure 3.1.: The hierarchy of classical and probabilistic complexity classes. Inclusions are represented by an arrow from the subset to the superset. Dashed lines indicate the two classes are conjectured to be equal

4. Quantum computation models

4.1. Quantum circuits

Analogous to the concept of classical or probabilistic circuits, this notion can be further generalized into the quantum realm. Recall that in Definition 2.6 we defined circuits as dependent on a basis of gates.

Definition 4.1: *A* quantum gate *with n inputs and n outputs is a unitary operation*

$$U : Q^{\otimes n} \to Q^{\otimes n}.$$

A *quantum basis* will be a set of quantum gates together with the following non-unitary operations,

1. The ANCILLARY gate, with 0 inputs and 1 output, that outputs a $|0\rangle$ qubit and
2. the DISCARD gate with 1 input a 0 outputs, that discards a qubit.

A quantum equivalent to the FANOUT operation is missing due to Proposition 1.6.

Thus,

Definition 4.2: *A* quantum circuit *is a circuit with respect to a quantum basis.*

A unitary quantum circuit *is a quantum circuit with no ancillary or discard gates. It has the same number of inputs and outputs.*

The ancillary and discard gates are not operations allowed within the quantum model since they are not unitary; nonetheless every quantum circuit can be transformed into a unitary quantum circuit:

Definition 4.3: *(Watrous (2009), sec III.3)*

Let C be a quantum circuit. The unitary purification *of C is a unitary quantum circuit C′ constructed by the following process*

- *every ancillary gate is replaced by a new input and*
- *every discard gate is replace by an output.*

A unitary quantum circuit has an associated unitary function, whose construction is given in the following definition.

Definition 4.4: *Let C be a unitary quantum circuit with n inputs and outputs. The function associated with the quantum circuit, $C : Q^{\otimes n} \to Q^{\otimes n}$ is constructed as follows*

1. *Construct a topological sort of the gates of C, that is, a way of sorting the gates such that if there is a wire from g_1 to g_2, then $g_1 \leq g_2$.*
2. *Tensor each quantum gate with the identity matrix to construct an unitary operation on $Q^{\otimes n}$.*
3. *Compose the resulting unitary operations in reverse order.*

Step 1 is possible since the graph associated to a circuit is acyclic. The independence of the construction from the chosen topological sort is given by the identity

$$(f \circ g) \otimes (h \circ s) = (f \otimes h) \circ (g \otimes s).$$

Using the previous definitions, every quantum circuit C has an associated function that outputs a random variable; if C has n inputs then given $|\psi\rangle \in Q^{\otimes n}$ we apply the operation of its unitary purification to the tensor product of $|\psi\rangle$ by the original values of the ancillary qubits and measure with respect to the computational basis. Finally, we discard the outputs that originally were a DISCARD gate.

We denote this random variable also by $C(|\psi\rangle)$. Restricting the measurement to the original outputs would give the same random variable; this is known as the *Principle of unterminated wires* in the literature (Nielsen and Chuang (2010)).

The concept of circuit family is analogous to the classical and probabilistic cases:

Definition 4.5: *A* quantum circuit family *over a gate set $\mathcal{G}$ is a sequence $\mathcal{C} = \{C_n\}_{n \in \mathbb{N}}$ of quantum circuits such that for every $n \in \mathbb{N}$, C_n is a circuit over $\mathcal{G}$ with n inputs.*

Lastly, given a quantum state $|\psi\rangle$ of n qubits and a quantum circuit family $\mathcal{C} = \{C_n\}_{n \in \mathbb{N}}$, we can get a random variable $\mathcal{C}(|\psi\rangle) = C_n(|\psi\rangle)$ in an analogous way.

4.1.1. Universal quantum gate sets

The classical case

It is a well-known fact that, in the classical case, any function $f : \mathbb{B}^* \to \mathbb{B}^*$ can be computed exactly by a circuit (of at most exponential size) built using the basis of gates

$$\{\text{AND}, \text{OR}, \text{NOT}\}.$$

There are other possible basis universal in this sense, such as {NAND} or {NOR}. Of course, not all basis are universal; a full classification of the basis is given by Post's lattice (Lau (2006)).

Given two finite universal basis B_1 and B_2 one may construct every element of the latter with a circuit made of gates of the former. The size of a family of circuits transformed in this manner increases only up to a constant and thus, in the classical case this discussion concludes with the following trivial proposition:

Proposition 4.1: *Let B_1, B_2 two finite universal basis, that is, two basis of gates that allow us to construct a circuit that computes an arbitrary function $f : \mathbb{B}^N \to \mathbb{B}^M$.*

Given a family of circuits with respect to B_1 that has size $O(f(n))$, there exists a family of circuits with respect to B_2 that computes the same function and has size $O(f(n))$.

The main consequence from this proposition is that the chosen basis is irrelevant for proving there exists a family of a certain (asymptotic) size. A similar result can be stated for probabilistic circuits by using Proposition 3.2.

This section aims to prove a similar (yet weaker) result for quantum circuits. This result is of great theoretical importance, since we do not currently understand what quantum operations are physically realizable, and thus would like to build a theory independent of any specific quantum gate basis.

Quantum universal gate sets

The set of unitary operations over a fixed Hilbert space can be seen as a group by considering the composition operation and as a normed space by considering the usual *operator norm*.

Definition 4.6: *A set of quantum gates $\mathcal{G}$ is* universal *if for every $\varepsilon > 0$ and every unitary operation U there exists a quantum circuit C over $\mathcal{G}$ such that $\|U_C - U\| < \varepsilon$.*

Several universal gate sets are described in the literature. For example, a minimal universal gate set can be obtained with probability 1 by combining the CNOT with a single qubit gate (Aaronson (2016b), sec. 2.1).

We now present an example of a universal gate set, which is not minimal but will serve our purposes.

Theorem 4.2: *(Nielsen and Chuang (2010), sec. 4.5.3)*

$\{\text{CNOT}, H, R_{\pi/4}\}$ is a universal gate set.

Proof sketch.

We present a brief proof sketch adapted from (Nielsen and Chuang (2010), sec. 4.5.3).

A *two-level unitary operation* is a unitary operation that acts different from the identity in at most two computational basis states. The first step towards proving the theorem is to prove that these gates form an (infinite) universal gate set.

Claim. (Nielsen and Chuang (2010), sec. 4.5.1) The set of two-level unitary operations is a universal gate set.

This can be easily shown to be the case by constructing matrices whose multiplication zero-out the components of a unitary matrix. Using this claim we can prove:

Claim. (Nielsen and Chuang (2010), sec. 4.5.2) The set of single qubit gates together with the CNOT gate forms a universal gate set.

To prove this claim we use the previous claim and construct an arbitrary two-level unitary gate by appropiately composing single qubit gates and the CNOT gate so that they are applied to the basis states that we need.

Lastly, to prove the theorem, it is shown that $\{H, R_{\pi/4}\}$ can approximate an arbitrary rotation of the qubit Hilbert space from any axis. This, together with a decomposition of an arbitrary single-qubit gate into these rotations, proves that $\{H, R_{\pi/4}\}$ can approximate any single qubit gate, which using the last lemma proves the result.

□

Solovay-Kitaev theorem

The previous theorem does not mention how "fast" the gates can be approximated; this is, what the asymptotic change of size of a circuit family would be if we were to replace the gates on it by approximations. This is crucial information; without it we would be unable to construct a theory independent of the universal gate set, and thus it would be unclear whether a certain algorithm would be efficient in practice.

The *Solovay-Kitaev Theorem* shows that the approximation can be done in quasi-logarithmic time for *any* universal gate set. This celebrated result is essential for the usefulness of certain algorithms, such as Grover's algorithm.

Theorem 4.3 (Solovay-Kitaev theorem)**:** *(Dawson and Nielsen 2005)*

Let $\mathcal{G}$ be a universal gate set closed under inverses and U a unitary operation.

There exists a constant c such that U can be approximated within ϵ accuracy by a sequence of $O(\log^c(1/\epsilon))$ gates from $\mathcal{G}$.

Proof sketch.

What follows is a proof sketch adapted from (Dawson and Nielsen 2005).

Let $n \in \mathbb{N}$. A recursive algorithm can be defined that returns the sequence of gates that approximates U. This algorithm gives more accurate and longer sequences depending on n.

For $n = 0$, we consider a basic approximation up to a certain accuracy ε_0. This approximation can be done for any gate by considering all possible sequences of gates up to a certain length.

If $n > 1$, we consider the approximation given by $n-1$, U_{n-1}. Then, let $\Delta = U_{n-1}U_{n-1}^\dagger$, where $U_{n-1}^\dagger$ is the adjoint of U_{n-1} (see Definition 1.2). We can express Δ as the conmutator of two unitary operators V, W, that is

$$\Delta = [V, W] = VWV^\dagger W^\dagger,$$

such that V and W are close enough to the identity.

We then find approximations to V and W of depth $n-1$, V_{n-1}, W_{n-1} and return as the approximation of depth n

$$U_n = [V_{n-1}, W_{n-1}]U_{n-1}.$$

This turns out to be an approximation up to an accuracy of order $\varepsilon_n = c_{\text{approx}}\varepsilon_{n-1}^{3/2}$.

This defines a sequence such that, to obtain an accuracy ε, we need to get an approximation of depth logarithmic on the length of ε which in turns gives a sequence of length

$$O(\ln^{\ln(5)/\ln(3/2)}(1/\varepsilon)),$$

which fulfills the conditions of the theorem.

□

Suppose a certain family of quantum circuits has $O(f(n))$ size. Theorem 4.3 states then that there is an ε-approximation by any universal quantum gate set of the family by another family that has $O(f(n)\log(f(n)/\varepsilon))$ size. If $f(n) \in \text{poly}(n)$, this gives us a *poly-logarithmic time* family.

4.1.2. Simulating classical operations

Classical and probabilistic operations can in principle be non-reversible, as is the case of the (linear extensions of) AND and OR gates, the NAND gate, the FANOUT gate or the RANDOM(1/2) probabilistic gate. For the quantum model to be a generalization of the probabilistic one we must be able to compute the same functions in a reversible manner.

This can be done thanks to the use of ancillary qubits. In the following discussion, let $|x\rangle$, $|y\rangle$ and $|z\rangle$ be classical states, that is states from the computational basis.

Recall from Proposition 1.5 the Toffoli gate, that maps $|x, y, z\rangle \mapsto |x, y, z \oplus xy\rangle$. By using an ancillary qubit on z and discarding qubits x, y we can simulate a NAND gate, that is

$$\text{TOFFOLI}\,|x, y, 1\rangle = |x, y\rangle\,\text{NAND}\,|x\rangle\,|y\rangle\,.$$

This therefore allows us to simulate any classical logical gate.

Similarly, the FANOUT gate can also be simulated by using a TOFFOLI gate,

$$\text{TOFFOLI}\,|x\rangle\,|1\rangle\,|0\rangle = |1\rangle\,|x\rangle\,|x\rangle\,.$$

This is not a contradiction with Proposition 1.6, since we are only cloning classical states.

Lastly, the RANDOM$(1/2)$ gate can be simulated with the aid of a Hadamard gate, since

$$H\,|0\rangle = \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle),$$

and therefore when measuring with respect to the computational basis we have a random bit.

These equivalences can be seen on fig. 4.1.

Clearly, we can then simulate any probabilistic circuit on a quantum computer by replacing each gate by its simulation.

In general, the unitary purification of the quantum circuit obtained in this way gives us for a function $f : \mathbb{B}^N \to \mathbb{B}^M$ a reversible function that maps

$$|x\rangle\,|c\rangle \mapsto |x\rangle\,|c \oplus f(x)\rangle\,,$$

which uses an extra polynomial amount of wires and gates.

4.2. Quantum computability

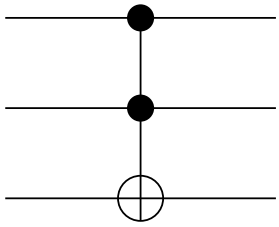
In this section we define what it means for a function to be efficiently computed by a quantum computer.

Definition 4.7: *A **quantum algorithm** is a uniform family of quantum circuits $\mathcal{C} = \{C_n\}$ over a finite universal gate set.*

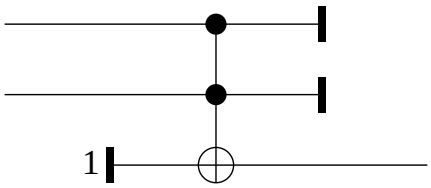
We say the algorithm is polynomial time *if the function $1^n \mapsto C_n$ can be computed in polynomial time.*

Computability in the classical and quantum notions coincides; the behavior of a uniform family of quantum circuits can be simulated with at most exponential slowdown. Thus any quantum computable function can be computed classically. The quantum simulation of classical operations proves the other implication.

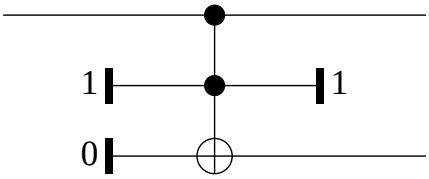
The interest lies therefore in studying complexity results, since, at least *a priori*, some functions might be able to be computed faster in the quantum model. The feasibly computable functions are, as in the classical and probabilistic case, those calculated in



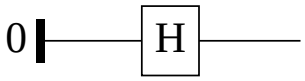
(a) The TOFFOLI gate symbol.



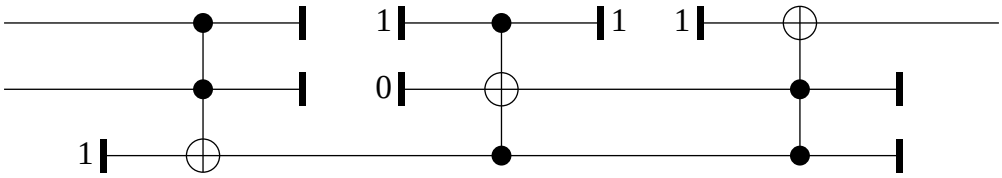
(b) NAND simulation.



(c) FANOUT simulation.



(d) RANDOM(1/2) simulation.



(e) AND gate simulated composing NAND gates.

Figure 4.1.: Reversible simulation of the probabilistic gate basis. Made with Quipper.

polynomial time. Analogous to the probabilistic case, we define quantum computability with bounded error.

Definition 4.8: *A function $f : \mathbb{B}^* \to \mathbb{B}^*$ is (polynomial time) **quantum computable** if there exists a (polynomial time) quantum algorithm $\{C_n\}$ such that*

$$\text{for every } x \in \mathbb{B}^*,\ P[C(|x\rangle) = f(x)] \geq \tfrac{2}{3}.$$

As it happened in the probabilistic case, by Proposition 3.3 we can see that the constant $\frac{2}{3}$ in Definition 4.8 is irrelevant when talking about polynomial time; by repeatedly executing the quantum algorithm a polynomial number of times we can ensure correctness with any probability $c \in]1/2, 1[$.

Furthermore, Theorem 4.3 shows that Definition 4.8 is independent from the chosen universal gate set: we may approximate it with an ε sufficiently small so as to remain above the $\frac{1}{2}$ threshold and we may do so in polynomial time if we are interested in efficiently computable functions.

4.3. Quantum time complexity

In the realm of complexity classes, the set of languages that can be efficiently decided by a quantum computer is formalized in the class BQP.

Definition 4.9: *$L \in \mathsf{BQP}$ if and only if the characteristic function of L, $1_L : \mathbb{B}^* \to \mathbb{B}$ is a quantum polynomial time computable function.*

Any language efficiently decidable in the probabilistic case is also efficiently decidable in the quantum case.

Proposition 4.4: $\mathsf{BPP} \subseteq \mathsf{BQP}$

Proof.

Let $L \in \mathsf{BPP}$. There exists a $O(p(n))$ sized uniform family of probabilistic circuits $\{C_n\}$ that decides L.

Let $n \in \mathbb{N}$. Consider the circuit C'_n formed by replacing every gate by its quantum simulation, as described in section Simulating classical operations.

$|C'n|$ is bounded above by $4|C_n|$, since the simulation of any probabilistic gate on the standard gate set can be done with at most 4 quantum gates. Therefore the family $\{C'_n\}$ is also $O(p(n))$ sized.

Their associated random variables for each input are identical, and thus the probability bound is achieved.

□

Furthermore, a classical upper bound on BQP can be given. Consider the following class:

Definition 4.10: *$L \in \mathsf{PQP}$ if and only if there exists a quantum polynomial time algorithm $\mathcal{C} = \{C_n\}$ such that*

1. *for every $x \in L$, $P[\mathcal{C}(x) = 1] > \frac{1}{2}$ and*
2. *for every $x \notin L$, $P[\mathcal{C}(x) = 1] \leq \frac{1}{2}$.*

This class is the quantum equivalent of PP, that is, a class like BQP where the probability bounds have been relaxed. As such, it is clear that $\mathsf{BQP} \subseteq \mathsf{PQP}$.

The following theorem shows that when these bounds are relaxed quantum computers provide no more than polynomial advantage.

Theorem 4.5: $\mathsf{PQP} = \mathsf{PP}$

Proof sketch.

The following proof sketch is adapted from (Watrous (2009), sec. IV.5).

A function $f : \mathbb{B}^* \to \mathbb{Z}$ is GapP if there exists a non-deterministic Turing Machine M such that for any x, $f(x)$ is the difference of the number of accepting paths and the number of rejecting paths of M with input x.

A simple characterization of PP is that there exists a function $f \in$ GapP such that $f(x) > 0$ if and only if $x \in L$.

(Fortnow 1997) proves the following claim:

Claim. Let $p, q \in \mathrm{poly}(n)$.

For each n, let $A_n, 1, \ldots, A_n, p(n) \in \mathcal{M}_{2^{q(n)} \times 2^{q(n)}}(\mathbb{C})$ be a sequence of matrices.

If there exist $f, g \in$ GapP functions such that

$$f(1^n, 1^k, i, j) = \mathrm{Re}(A_{n,k}(i, j)) \text{ and}$$

$$g(1^n, 1^k, i, j) = \mathrm{Im}(A_{n,k}(i, j)),$$

where i, j are written in binary, then there exists $F, G \in$ GapP functions such that

$$F(1^n, i, j) = \mathrm{Re}((A_{n,p(n)} \ldots A_{n,k})(i, j)) \text{ and}$$

$$G(1^n, i, j) = \mathrm{Im}((A_{n,p(n)} \ldots A_{n,k})(i, j)).$$

If $L \in \mathsf{PQP}$ then, there exists a uniform polynomial family of quantum circuits $\mathcal{C}$ that decides the language. Furthermore, a universal gate set may be chosen such that every gate has rational coefficients.

The procedure stated at Definition 4.4 gives us then a sequence of matrices that verifies the conditions of the claim:

- It is polynomial and polynomial-sized because of the size of the circuits and
- since it is uniform and the gates have rational coefficients, the functions f and g can be constructed for each matrix.

Applying the claim we then have a GapP function such that

$$P[\mathcal{C}(x) = 1] = \frac{F(x)}{2^{p(|x|)}},$$

and therefore by taking $h(x) = F(x) - 2^{p(|x|)-1}$ we have that $L \in \mathsf{PP}$.

□

Thus, the probabilistic class PP and therefore the classical class PSPACE are non-quantum upper bounds of the BQP class. This shows that the power of quantum computers is limited to what classical computers can do in polynomial space.

4.4. Quantum verifiers

Following the chapters on classical and probabilistic models of computation, it is now natural to consider quantum verifiers. In this kind of verifiers both the proof and the verifier are quantum.

Definition 4.11: *(Vidick and Watrous (2016), dfn. 3.1)*

$L \in \mathsf{QMA}$ *if and only if there exists a polynomial* $p(n)$ *and a polynomial time quantum algorithm* $\mathcal{V}$ *such that*

1. *for every* $x \in L$, $|x| = n$, *there exists a quantum state* $|\psi\rangle$ *of at most* $p(n)$ *qubits (the* proof*) such that*
$$P[\mathcal{V}(|x\rangle\,|\psi\rangle) = 1] \geq \frac{2}{3} \text{ and}$$
2. *for every* $x \notin L$, $|x| = n$, *and every quantum state* $|\psi\rangle$ *of at most* $p(n)$ *qubits*
$$P[\mathcal{V}(|x\rangle\,|\psi\rangle) = 1] \leq \frac{1}{3}.$$

If we look at Proposition 2.5, we can see QMA is a straightforward generalization of NP (and MA) to the quantum realm; thus we can easily guess the following proposition.

Proposition 4.6: $\mathsf{BQP}, \mathsf{MA} \subseteq \mathsf{QMA}$

Proof.

1. BQP $\subseteq$ QMA. Let $L \in$ BQP. Then there exists a polynomial time quantum algorithm $\mathcal{C}$ that computes 1_L. When $x \in L$, any proof will suffice, while when $x \notin L$, no proof will be correct with probability more than $\frac{1}{3}$.
2. MA $\subseteq$ QMA. Let $L \in$ MA. By definition, there exists a probabilistic verifier M, that can be made quantum by replacing each gate with its simulation. When $x \in L$, the proof will be the quantum state associated with the classical proof y of M. The probability bounds hold since they are identical in each class.

□

There is an obvious similarity between the P vs. NP problem and the BQP vs. QMA problem (the latter are also believed to be different). Nonetheless, there is no known formal result that relates the two problems, that is, separating one pair of classes would give us, in principle, no information about the other pair (Aaronson (2010)).

The constants $\frac{2}{3}$ and $\frac{1}{3}$ can be substituted by any $c \in]\frac{1}{2}, 1[$, yet the proof in this case is not as straightforward as in the case of the previous classes. The problem lies in the quantum *no-cloning theorem*, that prevents us from copying the quantum proof and running the algorithm several times.

Two approaches are possible for this error reduction; either multiplying the length of the proof by a constant (known as *parallel error reduction*) or relying on the information of the *garbage* registers to reconstruct the proof and run the proof sequentally (*witness-preserving error reduction*) (Vidick and Watrous (2016), sec. 3.2). The latter process has the advantage of preserving the proof size yet its proof is somewhat more contrived.

These give rise to the following proposition.

Proposition 4.7 (QMA error reduction)**:** *(Vidick and Watrous (2016), sec. 3.2)*

Let $c \in]\frac{1}{2}, 1[$.

Then $L \in$ QMA *if and only if there exists a polynomial* $p(n)$ *and a polynomial time quantum algorithm* $\mathcal{V}$ *such that*

1. *for every* $x \in L$, $|x| = n$, *there exists a quantum state* $|\psi\rangle$ *of at most* $p(n)$ *qubits (the* proof*) such that*
$$P[\mathcal{V}(|x\rangle\,|\psi\rangle) = 1] \geq c \text{ and}$$
2. *for every* $x \notin L$, $|x| = n$, *and every quantum state* $|\psi\rangle$ *of at most* $p(n)$ *qubits*
$$P[\mathcal{V}(|x\rangle\,|\psi\rangle) = 1] \leq 1 - c.$$

Proof sketch.

Although we omit the proof, a brief sketch of the proof in the parallel case can be given, adapted from (Vidick and Watrous (2016), sec. 3.2).

Let $L \in \mathsf{QMA}$, and let $\mathcal{V}$ be a verifier for L. For a given $x \in L$, we call a valid proof $|\psi_x\rangle \in Q^{\otimes p(|x|)}$.

We create a verifier $\mathcal{V}'$ that takes as a proof of length at most $Tp(|x|)$ qubits for a certain $T \in \mathbb{N}$. $\mathcal{V}'$ takes groups of $p(|x|)$ qubits and runs verifier $\mathcal{V}$ on them, outputting the majority vote.

Clearly, $|\psi_x\rangle^{\otimes T}$ would be a valid proof for $\mathcal{V}'$, and by applying Proposition 3.3 we can choose an appropiate T such that condition 1 is fulfilled.

What remains is showing that there is no possibility of taking advantage of the new verifier by using some superposition. This analysis can be done by considering the verifier as a measurement operator, which is outside the scope of this text.

□

Proposition 4.7 shows that we can prove a similar result to Proposition 3.4, namely:

Proposition 4.8: *(Watrous (2009), sec V.4)*

$\mathsf{QMA} \subseteq \mathsf{PP}$

Proof.

By Theorem 4.5, we know that $\mathsf{PQP} = \mathsf{PP}$, thus, it suffices to prove $\mathsf{QMA} \subseteq \mathsf{PQP}$.

The proof of this result is then identical to Proposition 3.4, using quantum circuits and adjusting for the probability bound of QMA as was suggested for MA.

□

This result gives us information about the tightness of the upper bound of BQP by PP. The chain $\mathsf{BQP} \subseteq \mathsf{QMA} \subseteq \mathsf{PP}$ hints that the bound is not very tight, since quantum proofs are presumed to have higher computational power than quantum algorithms.

The complete diagram of relationships between quantum, probabilistic and classical complexity classes is shown in fig. 4.2.

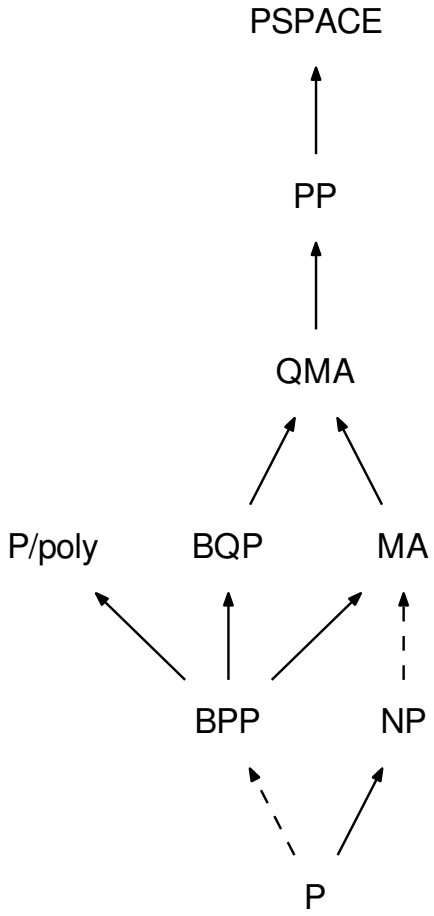


Figure 4.2.: The hierarchy of classical, probabilistic and quantum complexity classes. Inclusions are represented by an arrow from the subset to the superset. Dashed lines indicate the two classes are conjectured to be equal

Part II.

Concrete quantum algorithms

5. Quipper

In this chapter we present Quipper, a quantum programming language that we use to implement concrete quantum algorithms as uniform families of quantum circuits, and we describe the setup followed for the implementation.

5.1. Quantum programming languages

Quantum programming languages are programming languages that would be practically useful in the design and implementation of quantum algorithms. Since the early 2000s there has been a surge in proposals for quantum programming languages, which can roughly be classified into *imperative quantum programming languages*, which follow Knill's QRAM model (Knill (1996)) and *functional quantum programming languages*, which extends lambda calculus in some fashion, frequently using sophisticated type systems to ensure the correctness of programs (Gay (2006)).

Even without practically useful quantum computers in the present, the presence of quantum programming languages is essential for the design and verification of quantum algorithms, since they can be more easily tested against different examples. Furthermore, these programming languages can be given precise formal semantics that allow for their verification (Ying (2016), chap. 1).

In this bachelor's thesis I have chosen to use *Quipper*, a purely functional scalable programming language, since it is both practically usable and both its actual programs and its underlying semantics most closely resemble the uniform families of quantum circuits model for quantum programs. Quipper was designed as part of IARPA's QCS project and it has been used to implement several non-trivial quantum algorithms present in the literature (Green et al. (2013a)). Parts of Quipper have been given formal semantics, and the language has been generalized to describe families of morphisms in a symmetric monoidal category (Ross (2015), Rios and Selinger (2018)).

The following sections introduce practical aspects of programming in Quipper, and the next chapters each present the implementation of one or several quantum algorithms that exemplify these notions.

5.2. Quipper: a functional quantum programming language

Quipper is an embedded language based on *Haskell*, a functional programming language widely used within the functional programming community. In practical terms, Quipper consists of

1. `Quipper`, a Haskell library that defines several constructs and functions that allow us to define and manipulate both classical and quantum circuits,
2. `QuipperLib`, a second library that allows us to optimize, represent and simulate those circuits and
3. a preprocessing script that compiles the Quipper-specific syntax into Haskell.

The following sections describe these parts in more detail, as well as the host language Haskell.

5.2.1. Haskell

The host language used by Quipper is Haskell. Haskell is a purely functional programming language created in 1990, that has strong static typing and lazy evaluation. It has a formal specification given by the Haskell Report (Marlow (2010)).

In this document we make use of the version of Haskell implement by the Glasgow Haskell's compiler version 8.2.2 (("The Glasgow Haskell Compiler" 2019)). We make use of several language extensions that are not part of the Haskell standard since these are used by Quipper for its internal implementation.

We will not make an extensive description of Haskell here, yet here we briefly highlight a number of features that are relevant for the understanding of Quipper. An introduction to Haskell can be found at (Hutton (2016)).

Some relevant features of Haskell for the implementation of quantum algorithms in Quipper are:

Higher-kinded types Haskell's type system (together with GHC's non-standard extensions) allow for the use of *higher kinded types*. Haskell does not have dependent types, but rather its types have a *kind*. Base types such as `Char`, `Int` or `Double` have the kind `Type`, while type constructors such as lists can have a kind `Type -> Type`, that is, they take a base type and return a different base type (for example, `[Int]`, the type of lists of integers).

Typeclasses Polymorphic functions in Haskell are defined through the use of *typeclasses*. In its most general terms, a typeclass `Class a1 a2 .. an` is defined by a public interface that defines a number of polymorphic functions that can make use of the (possibly higher kinded) types `a1`, `a2`, `... an` in their type signatures. Then, the programmer can manually specify instances of the typeclass for a fixed colection of types `T1`, ..., `Tn`, by giving the definition of these functions for the fixed types. For

example, the typeclass `Ord a` defines (among others) the "less than" function, with type signature: `haskell (<) :: Ord a => a -> a -> Bool` It has instances for most types, such as `Int`, `Char` or `Double`.

Monads A particular typeclass that is extensively used in Quipper is the `Monad` typeclass. It is defined for types of kind `Type -> Type`, and it is inspired by the concept of *monad* in category theory, a monoid object in the category of endofunctors over a particular category. They are widely used in functional programming as computational contexts, to handle side effects and failure. Its public interface can be given[1] by two functions,

- `pure :: a -> Monad a`, that introduces an object into a minimal computational context and
- `(<=<) :: Monad m => (b -> m c) -> (a -> m b) -> a -> m c`, a composition operator for monadic functions.

Any instance of the `Monad` class should verify that the defined `<=<` is an associative operation with `pure` as its neutral element.

Haskell introduces a special syntax for monadic operations, the `do` notation. This syntax resembles imperative code, and will be used for defining the families of circuits.

5.2.2. Quipper features

Quipper implements a mixed circuit model of quantum computing, that includes both classical and quantum circuits, and allows for the mixing of both. As discussed in previous chapters, this model is equivalent to using pure quantum circuits, yet it allows for further flexibility in the implementation. We will also mix circuit code with Haskell code when this is possible for simplicity. While a complete formalization of the semantics of Haskell that would allow for a formal justification of this fact is not available, as most programming languages Haskell is considered to be Turing-equivalent, and thus code written this way would in principle still be able to be transformed to the circuit model.

Datatypes

Within its circuit model we can distinguish three phases of execution (Green et al. (2013a)): compile time (when Quipper code is transformed into Haskell code and compiled), *circuit generation* time (when, during program execution, the circuit is generated) and *circuit execution* time (when the circuit execution is simulated).

Using this distinction, we can distinguish two kinds of types

[1] This is not the actual definition in the Haskell Report, but it is equivalent. We are describing a monad in terms of its Kleisli category.

1. *parameters*, known at circuit generation time. These are represented by the datatype `Bool`.
2. *inputs*, known at circuit execution time. The classical inputs are represented by the type `Bit` and the quantum inputs are represented by `Qubit`.

Parameters can be turned into inputs but not vice versa. Furthermore, classical inputs can be turned into quantum inputs, but transforming quantum inputs into classical ones requires performing a measurement.

To define a circuit in Quipper we use the `Circ` monad. For example, a function $f : Q^{\otimes 2} \to Q^{\otimes 3}$ would have type

```
f :: (Qubit, Qubit) -> Circ (Qubit, Qubit, Qubit)
```

Example 5.1 (A simple circuit)**:** *The file `src/apps/Classical.hs` in the associated code*[2] *defines several simple classical and quantum gates, that have been used to produce the diagrams in this document.*

For example, the FANOUT *gate can be seen as having type*

```
fanout :: Qubit -> Circ (Qubit, Qubit)
```

Its definition is

```
fanout x = do
  y  <- qinit True
  z  <- qinit False
  (x, y, z) <- toffoli (x, y, z)
  qterm True y
  pure (x, z)
```

For its definition we use the `qinit` function, that transforms parameters into quantum inputs (serving as the ANCILLARY *gate), the `toffoli` gate and the `qterm` function for the* DISCARD *gate.*

To allow for the definition of families of quantum circuits, Quipper introduces the typeclass `QShape ba qa ca`. It is defined for triples of types with kind `Type`, and its purpose is to allow for generic definitions of circuits.

For example, an instance of `QShape` can be given by the types `Bool`, `Qubit` and `Bit`, but also for `[Bool]`, `[Qubit]` and `[Bit]`. This allows for generic definitions of circuits that can have an arbitrary shape of input, thus easily defining families of circuits.

Example 5.2 (Oracles)**:** *As we saw in section simulating classical operations, a function*

$$f : \mathbb{B}^n \to \mathbb{B}$$

[2] See the Appendix for instructions on how to obtain the code if it has not been provided with this file.

can be transformed into a reversible function

$$U_f : Q^{\otimes n+1} \to Q^{\otimes n+1}$$

that maps

$$|x\rangle\,|y\rangle \mapsto |x\rangle\,|y \oplus f(x)\rangle\,.$$

In the attached code, these reversible functions are given by the datatype `Oracle`*, that has a* `shape` *and* `circuit`*:*

```
data Oracle qa = Oracle {
  shape   :: qa, -- ^ The shape of the input
  circuit :: (qa,Qubit) -> Circ (qa,Qubit) -- ^ The circuit oracle
  }
```

We also define a function `buildOracle` *that can build* U_f *given* f*. For this we make use of the* `QShape` *datatype, with which we can express the type signature as:*

```
buildOracle
  :: QShape ba qa ca
  => qa
  -> (qa -> Circ Qubit)
  -> Oracle qa
```

Circuit generation

Quipper includes a powerful circuit generation system, that uses Haskell's compile-time metaprogramming capabilities to produce circuits from classical code. This can be used for transforming `Bool` based functions into circuits.

For this, Quipper introduces the special syntax `build_circuit`, that is not legal Haskell syntax. If placed before a boolean function it produces a circuit based on the code of that function. For example, to define the XNOR gate:

```
build_circuit
booleanXnor (x, y) = (not x || y) && (x || not y)

xnor :: (Qubit, Qubit) -> Circ Qubit
xnor = unpack template_booleanXnor
```

This is used to define an automatic circuit generator from a truth table given in a CSV file (or, if the filename is missing, from the standard input). We will use this to pass concrete functions to the defined algorithms.

5.2.3. stack and `quipperlib` setup

`stack` is Haskell's most popular package manager ("The Haskell Tool Stack" 2019). It provides a framework for reproducible builds and an easy way of installing a package dependencies. It is compatible with Windows, macOS and Linux based OSs.

In order to ease the programming in Quipper, for this work I have bundled the official Quipper code into a `stack`-compatible Haskell package, which is available on Github. Quipper's preprocessor has been manually included in `stack`'s building process so as to be able to use the full Quipper language.

A Makefile is provided to aid in the compilation of the binaries.

The provided binaries are

1. `quantum`, that provides a command-line interface for the simulation and graphical representation of several quantum algorithms.
2. `diagrams`, that produces the circuit diagrams that are used in this document making use of Quipper's libraries.

The following chapters will describe the quantum algorithms in detail, as well as their implementations.

6. Query complexity and the Deutsch-Jozsa algorithm

In this chapter we develop our first example of a quantum algorithm. The first section presents the setting in which the complexity of this and later algorithms will be analyzed, and the following section presents the solution.

6.1. The query complexity model

The focus of the previous sections on computation models was on time complexity. These are the most meaningful practical measures of complexity; however, they are notoriously difficult to analyze, as the wide range of existing open problems in the field shows (Aaronson (2016a)).

In the analysis of the algorithms that will be presented we will sometimes focus on an alternative complexity measure: *query complexity*, also known in the classical case as *decision tree complexity* (Ambainis (2017), sec. 2).

In the quantum case, we are given an *oracle* (or, more generally, a family of oracles) that gives us information about a certain binary string $x \in \mathbb{B}^N$ (Kaye, Laflamme, and Mosca (2007), sec 9.2). Similar to the approach for simulating classical operations, we can have a unitary operation that maps

$$|j\rangle\,|y\rangle \mapsto |j\rangle\,|y \oplus x_j\rangle\,.$$

We would then like to compute some property about the oracle; the question then becomes: how many *queries* have to be made to the oracle in order to compute such property with bounded error?

That is we say that a quantum algorithm $\{C_N\}$ (with respect to any set of gates that includes the quantum oracle) that computes a certain function $f : \mathbb{B}^* \to \mathbb{B}^*$ (with bounded error) has query complexity $O(T(N))$ if the function that for each N counts the number of oracle gates in C_N is $O(T(N))$.

An alternative way of presenting query complexity is to talk about a function $f : \mathbb{B}^n \to \mathbb{B}$ that outputs $f(j) = x_j$ (where j is passed as a binary string). We note that $N = 2^n$, and thus in this approach the query complexity would be $O(T(2^n))$.

Query complexity then gives a lower bound estimate of the actual time complexity; since we would have to add in the non-query gates as well as the amount of gates needed to

simulate the oracle.

6.2. Deutsch's problem

The Deutsch-Jozsa algorithm is one of the first quantum algorithms that provide some quantum speedup in the query complexity model. It was first proposed in 1992 (Cleve et al. 1998). It will serve as a first approximation to a concrete algorithm, although it is of little practical value.

The problem it solves is as follows

Problem 6.1 (Deutsch's Problem)**:** *Find out whether a function is balanced or constant.*

- ***Input:*** *A function* $f : \mathbb{B}^n \to \mathbb{B}$
- ***Promise:*** *The function is either constant or it is* balanced*, i.e.,* $|f^{-1}(0)| = |f^{-1}(1)|$.
- ***Output:*** *A bit* b *where* $b = 0$ *if the function is constant or* $b = 1$ *if it is balanced.*

As we work in the query complexity model, we assume f is given as a reversible oracle that maps

$$|x\rangle |y\rangle \mapsto |x\rangle |y \oplus f(x)\rangle .$$

In the classical case, at worst we have to check $2^{n-1} + 1$ inputs to distinguish between the two classes. In the quantum case in contrast, it can be checked with exactly one query.

First, consider the following simple lemma,

Lemma 6.1: *(Nielsen and Chuang (2010), eq. 1.50)*

Let $n \in \mathbb{N}, N = 2^n$ *and* $x \in \mathbb{B}^n$*. Then*

$$H^{\otimes n} |x\rangle = \frac{1}{\sqrt{2^n}} \sum_{y \in \mathbb{B}^n} (-1)^{x \odot y} |y\rangle ,$$

where, if $x = x_1 \dots x_n$ *and* $y = y_1 \dots y_n$*, then*

$$x \odot y = \sum_{j=1}^{n} x_j y_j \mod 2,$$

is its "bit-wise inner product" in $\mathbb{Z}_2$*.*

Proof.

Let $n = 1$. We have

$$H |x\rangle = \frac{1}{\sqrt{2}}(|0\rangle + (-1)^x |1\rangle) = \sum_{y=0}^{1} (-1)^{xy} |y\rangle = \sum_{y \in \mathbb{B}} (-1)^{x \odot y} |y\rangle .$$

Let $n > 1$. Then

$$\begin{aligned} H^{\otimes n} |x\rangle &= \bigotimes_{k=1}^{n} \sum_{y_k=0}^{1} (-1)^{x_k y_k} |y_k\rangle \\ &= \sum_{y \in \mathbb{B}^n} \bigotimes_{k=1}^{n} (-1)^{x_k y_k} |y_k\rangle \\ &= \sum_{y \in \mathbb{B}^n} (-1)^{\sum x_k y_k} |y\rangle \\ &= \sum_{y \in \mathbb{B}^n} (-1)^{x \odot y} |y\rangle . \end{aligned}$$

□

In particular, we can obtain an uniform superposition of all possible n bit strings by using Lemma 6.1,

$$H^{\otimes n} |0\rangle^{\otimes n} = \frac{1}{\sqrt{2^n}} \sum_{y \in \mathbb{B}^n} (-1)^{0 \odot y} |y\rangle = \frac{1}{\sqrt{2^n}} \sum_{y \in \mathbb{B}^n} |y\rangle .$$

Another simple lemma that will be useful later is

Lemma 6.2: *Let $f : \mathbb{B}^n \to \mathbb{B}$ be a function and $U_f : Q^{\otimes n+1} \to Q^{\otimes n+1}$ be the reversible function associated with f, that is,*

$$U_f |x\rangle |y\rangle = |x\rangle |y \oplus f(x)\rangle .$$

Let

$$|\downarrow\rangle = \frac{1}{\sqrt{2}} (|0\rangle - |1\rangle) .$$

Then

$$U_f |x\rangle |\downarrow\rangle = (-1)^{f(x)} |x\rangle |\downarrow\rangle .$$

Proof.

Clearly,

$$U_f |x\rangle |\downarrow\rangle = \frac{1}{\sqrt{2}} (|x\rangle |f(x)\rangle - |x\rangle |1 \oplus f(x)\rangle).$$

If $f(x) = 0$ then

$$|x\rangle |f(x)\rangle - |x\rangle |1 \oplus f(x)\rangle = |x\rangle |0\rangle - |x\rangle |1\rangle = |x\rangle \otimes (|0\rangle - |1\rangle).$$

Similarly, if $f(x) = 1$,

$$|x\rangle |f(x)\rangle - |x\rangle |1 \oplus f(x)\rangle = |x\rangle |1\rangle - |x\rangle |0\rangle = - |x\rangle \otimes (|0\rangle - |1\rangle).$$

□

Next, consider the following algorithm:

Algorithm 1 (Deutsch-Jozsa algorithm)**:** *(Nielsen and Chuang (2010), sec. 1.4.4)*

Solves: *Problem 6.1.*

1. *Initialize n qubits to $|0\rangle$ and an extra qubit to $|1\rangle$.*
2. *Apply Hadamard gates to each qubit.*
3. *Apply the oracle for f to the whole state.*
4. *Apply Hadamard transform to the first n qubits.*
5. *Discard the last qubit and measure the first n qubits.*
6. *Output the logical OR of the measured bits.*

An example of the algorithm for $n = 2$ can be seen in fig. 6.1.

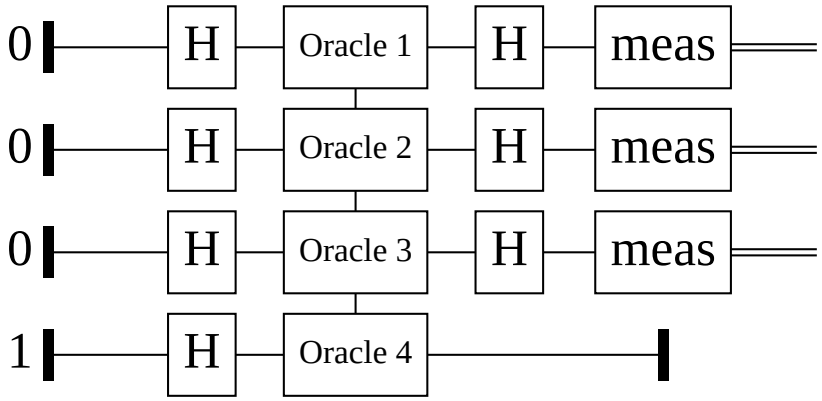


Figure 6.1.: Deutsch-Jozsa algorithm for a function $f : \mathbb{B}^3 \to \mathbb{B}$

Proposition 6.3 (Correctness of Deutsch-Jozsa algorithm)**:** *(Nielsen and Chuang (2010), sec. 1.4.4)*

Let $f : \mathbb{B}^n \to \mathbb{B}$ be a constant or balanced function. Then Algorithm 1 output is non-zero if and only if f is balanced.

Proof.

Let $|\phi_j\rangle$ be the state at step j.

By construction,

$$|\phi_1\rangle = |0\rangle^{\otimes n} |1\rangle .$$

Applying Hadamard gates to each qubit gives us

$$|\phi_2\rangle = H^{\otimes n} |0\rangle^{\otimes n} H |1\rangle = \left(\frac{1}{\sqrt{2^n}} \sum_{y \in \mathbb{B}^n} |y\rangle \right) \otimes \frac{1}{\sqrt{2}} (|0\rangle - |1\rangle) .$$

We now apply f's oracle and by Lemma 6.2 we have

$$|\phi_3\rangle = \frac{1}{2^n} \sum_{y \in \mathbb{B}^n} (-1)^{f(y)} |y\rangle \otimes \frac{1}{\sqrt{2}} (|0\rangle - |1\rangle) .$$

By Lemma 6.1 we see that

$$|\phi_4\rangle = \frac{1}{2^n} \left(\sum_{z \in \mathbb{B}^n} \sum_{y \in \mathbb{B}^n} (-1)^{y \odot z + f(y)} |z\rangle \right) \otimes \frac{1}{\sqrt{2}} (|0\rangle - |1\rangle) .$$

Ignore the last qubit and consider the probability of measuring $|0\rangle^{\otimes n}$ at step 5. Its probability is given by

$$\left| \frac{1}{2^n} \sum_{y \in \mathbb{B}^n} (-1)^{f(y)} \right|^2 .$$

If f is constant, then this probability is exactly 1, and the logical OR will give us 0. If f is balanced, then

$$\sum_{y \in \mathbb{B}^n} (-1)^{f(y)} = \sum_{y \in f^{-1}(0)} (-1)^0 + \sum_{y \in f^{-1}(1)} (-1)^1 = 0.$$

Hence, at least one of the measured bits will be 1 and the output will be 1.

□

In contrast with the deterministic case, Algorithm 1 solves Problem 6.1 in one query, thus we can obtain a provable speedup from $O(N)$ to $O(1)$.

Although this is a non-negligible improvement, the speedup vanishes when randomness is allowed. Consider an algorithm that uniformly samples two different strings x_1, x_2 from $\mathbb{B}^n$ and outputs constant if $f(x_1) = f(x_2)$ or balanced otherwise.

If f is constant, then the algorithm always outputs the correct answer. If f is balanced, assume, without loss of generality, that $f(x_1) = 0$. The probability of success is then

$$P(\text{Success}) = P(f(x_2) = 1) = \frac{N/2}{N-1} = \frac{1}{2} + \frac{1}{2(N-1)},$$

and therefore by Proposition 3.3 the error can be bounded by repeating the algorithm and taking the majority vote.

Algorithm 1 may seem superior to this probabilistic alternative in that it outputs the correct answer with certainty and it does so in exactly one query, when, in comparison the probabilistic algorithm takes at least two queries. Nonetheless, this property is dependent on the chosen gate basis, since the approximation given by the Solovay-Kitaev theorem might introduce some error, so the seeming advantage might vanish in practice.

Hence, the two alternatives are asymptotically equivalent and there is no speedup gained by this algorithm in comparison with the classical case.

6.3. Quipper implementation

This section outlines the implementation of Algorithm 1 in Quipper. The complete code is available at `src/lib/Algorithms/Deutsch.hs`. We will need some functions and datatypes mentioned in Quipper features.

The algorithm will have type

```
deutschJozsa :: (QShape ba qa ca) => Oracle qa -> Circ ca
```

since it takes an oracle with input shape `qa` and returns classical bits `ca` with the same shape. We name the input oracle `oracle`.

First, we need to initialize as much qubits as the oracle takes to $|0\rangle$ and an extra one to *ket*1. We use `qinit :: QShape qa ba ca => ba -> Circ qa`, that initializes an arbitrary data structure with qubits that have some fixed value. It can be used in conjunction with `qc_false` to initialize the qubits:

```
(x, y) <- qinit (qc_false (shape oracle), True)
```

Next, we apply `map_hadamard :: QShape qa ba ca => qa -> Circ qa` that applies the Hadamard gate to each qubit:

```
(x, y) <- map_hadamard (x, y)
```

For the next step we apply the oracle. To have a simpler graphical representation, we use the function `box`, which encapsulates a circuit into a box, so that on its depiction it appears on a different page. This has no effect on the simulation.

```
(x, y) <- boxedOracle (x, y) -- boxedOracle = box "Oracle" $ circuit oracle
```

We apply the Hadamard gate again, this time to the top qubits (called `x`), measure the top qubits and discard the bottom qubit `y`. Finally, we return the result using `pure`.

The complete algorithm can be seen in the following code snippet.

```
deutschJozsa :: (QShape ba qa ca) => Oracle qa -> Circ ca
deutschJozsa oracle = do
  -- Initialize x (to False) and y (to True)
  (x, y) <- qinit (qc_false (shape oracle), True)

  -- Apply Hadamard gates to each qubit
  (x, y) <- map_hadamard (x, y)

  -- Apply oracle
  (x, y) <- boxedOracle (x, y)

  -- Apply Hadamard gate again to x
  x      <- map_hadamard x

  -- Measure and discard
  z      <- measure x
  qdiscard y
  pure z
```

In order to solve Problem 6.1, we use the function `run_generic_io` to get the result:

```
result <- run_generic_io (0 :: Double) deutschCircuit
```

and the output will be given by whether there is or not a `True` boolean:

```
if or result then Balanced else Constant
```

This algorithm can be run by using the `quantum` binary with the `deutsch` subcommand.

7. Shor's algorithm

7.1. Introduction

In this chapter we describe one of the most well-known quantum algorithms: Shor's algorithm, which shows that factoring is feasible quantumly. To develop this algorithm we will also need to describe two essential quantum subroutines that will find later use: the Quantum Fourier Transform and the quantum phase estimation algorithm.

As it is usual in complexity theory, we encode natural numbers by their binary representation, so that the length of the input $N \in \mathbb{N}$ is $\lceil \log N \rceil \in O(\log N)$.

The decision problem we will solve is the one associated with the language:

Definition 7.1:

$$\text{FACTORING} = \{(n,k) \ : \ n,k \in \mathbb{N},\ n \text{ has a non-trivial divisor smaller than } k\}.$$

In order to solve it we will solve the associated function problem: we will give a quantum algorithm that gets a non-trivial factor of a number.

Problem 7.1 (Factoring)**:** *Find a factor of an integer N.*

- ***Input:*** *An integer N.*
- ***Promise:*** *N is composite*
- ***Output:*** *A non trivial factor of N.*

Solving this function problem and solving the decision problem of FACTORING are equivalent up to a classical polynomial time reduction. Furthermore, the promise of N not being prime can be checked in classical polynomial time by first using a primality testing algorithm (Agrawal, Kayal, and Saxena (2004)).

Thus, we will prove:

Theorem 7.1: *(Nielsen and Chuang (2010), sec. 5.3.2)*

FACTORING $\in$ BQP. *In particular, Problem 7.1 is solvable in $O(\log^3 N)$ quantum time with bounded error.*

How is Problem 7.1 classified classically? FACTORING $\in$ NP is easy to show via Proposition 2.5: for $(n,k) \in$ FACTORING, the proof would be the non-trivial factor m,

that would have at most as many digits as k, and the verifier would check that $m|n$, which can easily be done in polynomial time by the usual division algorithm.

On the other hand, it is believed that FACTORING $\notin$ P. Although this is not proven (since a proof would imply solving the P vs. NP problem), the best known classical algorithm for solving this problem runs in time $\exp(O(\log^{1/3} N \sqrt{\log \log N})$ (Lenstra et al. (1990)).

Factoring is an important problem because of the reliance of some widely used cryptographic systems on the hardness of this problem (Arora and Barak (2009), sec. 9.2.1). Shor's algorithm is widely appraised for being one of the most important results of quantum computation, since it gives definite proof of an exponential quantum speedup between the best known classical algorithm and the best known quantum algorithm.

Nonetheless, its applicability is somewhat limited. FACTORING is believed not to be NP-complete, that is, an algorithm that decides this language does not seem to allow us to solve an arbitrary NP problem. Hence, Shor's algorithm speedup is not as useful in principle as the one that will be provided by Grover's algorithm (although this one is only polynomial).

7.2. The Quantum Fourier Transform

In this section we will describe the Quantum Fourier Transform, an essential part of Shor's algorithm and one of the main algorithms that allows us to achieve superpolynomial speedups. It depends on the notion of discrete normalized Fourier transform.

Recall from Definition 1.4 that the *amplitude vector* of a quantum state is the coordinate vector of that state with respect to the computational basis.

Definition 7.2 (Unitary DFT)**:** *(Rao, Kim, and Hwang (2010), sec 2.1.3)*

The discrete normalized Fourier Transform *(UDT) is the map* $\mathrm{UDT} : \mathbb{C}^N \to \mathbb{C}^N$ *given by*

$$\mathrm{UDT}(x) = y, \text{ where } y_k := \frac{1}{\sqrt{N}} \sum_{j=0}^{N-1} x_j \exp\left(\frac{2\pi i j k}{N}\right),$$

for all $k \in \{0, \ldots, N-1\}, x = (x_0, \ldots, x_{N-1}) \in \mathbb{C}^N$.

The unitary DFT has a quantum equivalent:

Definition 7.3 (Quantum Fourier Transform)**:** *(Nielsen and Chuang (2010), sec. 5.1)*

The quantum Fourier transform *(QFT) is the unique linear operator* $\mathrm{QFT} : Q^{\otimes n} \to Q^{\otimes n}$ *that*

maps

$$|x\rangle \mapsto \frac{1}{\sqrt{N}} \sum_{k=0}^{N-1} \exp\left(\frac{-2\pi i x k}{N}\right) |k\rangle$$

for every for $x \in \{0, \ldots, N-1\}$.

Alternatively, it is the operator that maps a quantum state with amplitude vector x *in the quantum state with amplitude vector* $\mathrm{UDT}(x)$.

Its most straightforward application is the sampling according to the distribution given by the Fourier transform coefficients, but it is not directly applicable for the calculation of the UDT, as the amplitude vector can not be recovered directly.

Despite this important caveat, the QFT is an essential part of Shor's algorithm and the quantum counting algorithm among others, since it allows us to efficiently encode information on the amplitude vector. To show how to compute it efficiently, we first arrive at an alternative representation of the QFT that simplifies its calculation.

Lemma 7.2 (Product representation)**:** *(Nielsen and Chuang (2010), sec. 5.1)*

Let $x \in \{0, \ldots, N-1\}$ *and* $x_j \in \{0, 1\}$ *such that*

$$x/2^n = \sum_{j=1}^{n} x_j 2^{-j} = 0.x_1 \ldots x_n.$$

Then

$$\begin{aligned} \mathrm{QFT}\,|x\rangle &= \mathrm{QFT}\,|x_1 \ldots x_n\rangle \\ &= \frac{1}{\sqrt{N}}(|0\rangle + \exp(2\pi i 0.x_n)\,|1\rangle) \otimes \cdots \otimes (|0\rangle + \exp(2\pi i 0.x_1 \ldots x_n)\,|1\rangle) \end{aligned}$$

Proof.

Let $k/2^n = \sum_{j=1}^{n} k_j 2^{-j} = 0.k_1 \ldots k_n$.

We then have

$$\begin{aligned}
\sum_{n=0}^{N-1} x_n \exp\left(2\pi i x \frac{k}{2^n}\right) &= \sum_{n=0}^{N-1} x_n \exp\left(2\pi i x \sum_{j=1}^{n} k_j 2^{-j}\right) \\
&= \sum_{k\in\mathbb{B}^n} \bigotimes_{j=1}^{n} \exp(2\pi i x k_j) \left|k_j\right\rangle \\
&= \bigotimes_{j=1}^{n} \left(\sum_{k_j=0}^{1} \exp(2\pi i x k_j 2^{-j}) \left|k_j\right\rangle\right) \\
&= \bigotimes_{j=1}^{n} (\left|0\right\rangle + \exp(2\pi i x 2^{-j}) \left|1\right\rangle) \\
&= \bigotimes_{j=1}^{n} (\left|0\right\rangle + \exp(2\pi i 0.x_{n-j+1}\dots x_n) \left|1\right\rangle),
\end{aligned}$$

where in the last equality we have used $x2^{-j} = x_1 \dots x_{n-j}.x_{n-j+1}\dots x_n$, and therefore

$$\begin{aligned}
\exp(2\pi i x 2^{-j}) &= \exp(2\pi i x_1 \dots x_{n-j}) \exp(2\pi i 0.x_{n-j+1}\dots x_n) \\
&= \exp(2\pi i 0.x_{n-j+1}\dots x_n),
\end{aligned}$$

since $\exp(2\pi i a) = 1$ for any $a \in \mathbb{Z}$.

□

This expression allows us to easily construct a polynomial-size uniform family of quantum circuits that computes the QFT. For comparison, notice that the best-known classical algorithm for computing the DFT (the FFT algorithms) have an asymptotic efficiency of $O(N \log N)$ (Rao, Kim, and Hwang (2010)).

Theorem 7.3 (QFT Algorithm)**:** *There exists a polynomial-size uniform family of quantum circuits that computes the quantum Fourier transform.*

Specifically, for a quantum input of $N = 2^n$ qubits the circuit has a size $O(n^2) = O(\log^2 N)$.

Proof.

Let $n \in \mathbb{N}$ be the number of qubits of the input, of shape $\left|x\right\rangle = \left|x_1 \dots x_n\right\rangle$.

For $k = 2, ;n$, let $R_k := R_{2\pi/2^k}$ be a phase rotation gate.

Clearly,

$$H\left|x_i\right\rangle = \frac{1}{\sqrt{2}}(\left|0\right\rangle + (-1)^{x_i}\left|1\right\rangle) = \frac{1}{\sqrt{2}}(\left|0\right\rangle + e^{2\pi i 0.x_1}\left|1\right\rangle).$$

Let $k \in \{1, \ldots, n\}$. Assume a qubit is in state

$$|\psi\rangle = \frac{1}{\sqrt{2}}(|0\rangle + e^{2\pi i 0.x_1 x_2 \ldots x_{k-1}}),$$

then, if we apply an R_k gate controlled by $|x_k\rangle$ we then have

$$C - R_k |x_k\rangle |\psi\rangle = \frac{1}{\sqrt{2}}(|0\rangle + e^{2\pi i 0.x_1 x_2 \ldots x_k}).$$

Hence, if for each qubit $|x_k\rangle$ we apply the Hadamard gate and the controlled R_j gate for $j = 1, \ldots, n-k+1$ controlled by $|x_j\rangle$, we will have, up to reordering of the qubits, the expression given by Lemma 7.2.

The circuit will have size

$$\frac{n(n+1)}{2} \in O(n^2),$$

since each qubit needs $n - k + 2$ gates.

□

As any result having to do with concrete sizes of families of quantum circuits, the $O(n^2)$ Theorem 7.3 is given with respect to a basis that has the appropriate gates, and for a fixed basis there might be a poly-logarithmic difference given by Theorem 4.3.

An example circuit for the Quantum Fourier Transform of 3 qubits can be seen in fig. 7.1.

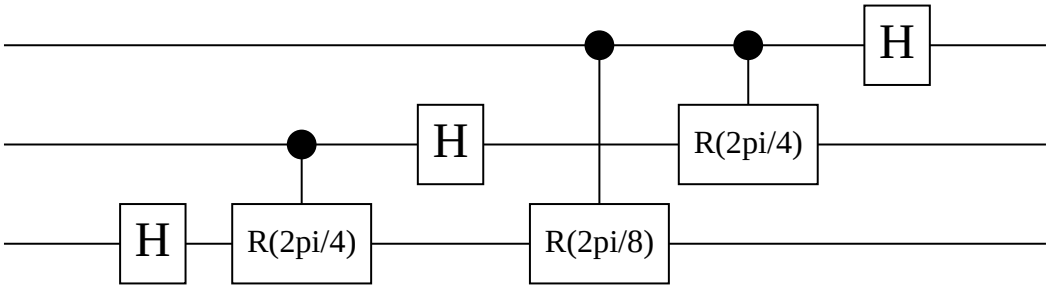


Figure 7.1.: Circuit for the 3 qubit Quantum Fourier Transform.

7.2.1. Quipper implementation

The Quipper implementation for the Quantum Fourier Transform is a simplified version of the one implemented at `QuipperLib.QFT`. It can be found at

`src/lib/Algorithms/QFT.hs`.

It is a simple recursive algorithm. The base case uses the empty list and does nothing,

```
qft [] = pure []
```

The recursive case applies the base case to all but the first qubit, conditionally rotates the first qubit with respect to the rest and then applies a Hadamard gate.

```
qft (x:xs) = do
  xs'  <- qft xs
  xs'' <- rotations x xs' (length xs')
  x'   <- hadamard x
  pure (x' : xs'')
```

The `rotations` function implements a simple loop recursively by making use of applicative functors.

```
rotations :: Qubit -> [Qubit] -> Int -> Circ [Qubit]
rotations _ []     _ = pure []
rotations c (q:qs) n =
  (:) <$> rGate ((n + 1) - length qs) q `controlled` c <*> rotations c qs n
```

`n` is the total length of the qubits that we want to condition.

7.3. Quantum phase estimation

We now present an algorithm that computes an n-bit approximation of the eigenvalue of an operator. This will be directly useful to solve the factoring problem, and it will also be applicable in the case of Grover's algorithm.

Problem 7.2 (Phase estimation)**:** *Approximate an eigenvalue of the eigenvector $|u\rangle$ a unitary operator U.*

- ***Input**: For each j, an oracle that computes a controlled U^{2^j} operation and a vector $|u\rangle$*
- ***Promise**: $|u\rangle$ is an eigenvector of U.*
- ***Output**: An n-bit approximation of φ_u such that*

$$\exp(2\pi i \varphi_u)$$

is the eigenvalue of U associated with U.

Since U is unitary, its eigenvalues will lie on the unit circle, and thus they can all be expressed as $\exp(2\pi i\varphi)$ for some $\varphi \in [0, 1[$. Hence, if we approximate $\varphi \approx 0.x_1 \dots x_n$, we can output $|x_1 \cdots x_n\rangle$ as an answer. We call φ_u a *phase*.

Next, we present a quantum algorithm that solves this problem in a polynomial number of queries in the number of bits of the approximation.

Algorithm 2 (Quantum phase estimation)**:** *(Nielsen and Chuang (2010), sec 5.2)*

Solves: *the phase estimation problem, Problem 7.2.*

Let $t = n + 2$.

1. *Initialize* $|0\rangle^{\otimes t} |u\rangle$. *Call the first* t *qubits the* phase qubits *and the rest the* eigenvector qubits.
2. *Obtain an uniform superposition on the phase qubits by applying Hadamard gates.*
3. *For each* $j = 1 \dots t$, *apply the* U^{2^j} *operation on the eigenvector qubits controlled on the* j^{th} *phase qubit.*
4. *Apply the inverse Quantum Fourier Transform on the phase qubits.*
5. *Discard the bottom qubits and measure the phase qubits.*

It is easy to show by applying Lemma 7.2 that the algorithm is correct on the case where φ has exactly n qubits, since the output of the first part of the algorithm gives us in that case the QFT of the desired output.

An example circuit implementing Algorithm 2 is shown in fig. 7.2.

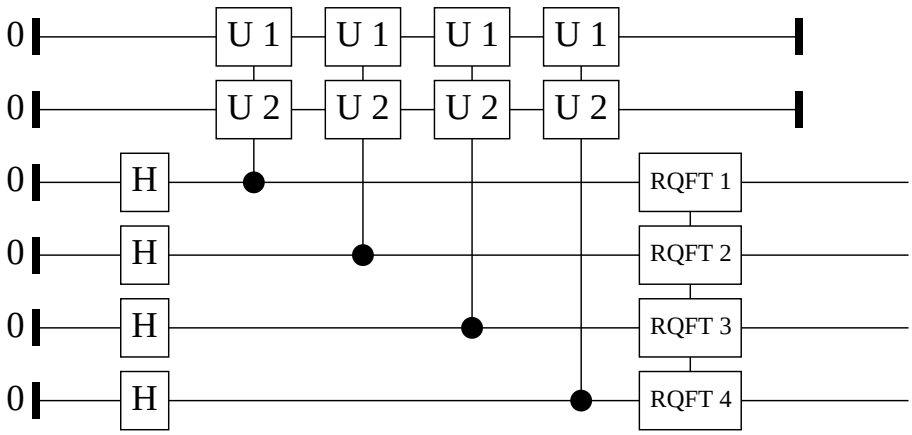


Figure 7.2.: The Quantum Phase Estimation subroutine for an unitary operator $U : Q^{\otimes 2} \to Q^{\otimes 2}$ with eigenvector $|00\rangle$ and an approximation using $t = 4$ qubits. "RQFT" does the reverse QFT.

We now present the general proof of correctness.

Theorem 7.4 (Correctness of QPE)**:** *(Nielsen and Chuang (2010), sec. 5.2.1)*

Let U *be a unitary operator with an eigenvalue* $\exp(2\pi i\varphi)$. *An* n*-bit approximation of* φ *is polynomial-time computable in the quantum black box model* ($O(n^2)$)

Proof.

The proof is also adapted from (Nielsen and Chuang (2010), sec. 5.2.1).

The number of quantum gates used by Algorithm 2 is $O(n^2)$, since we only add a linear amount of gates apart from the ones used in the QFT algorithm.

For the correctness, we focus on the phase qubits only, since the eigenvector qubits remain unaltered throughout the whole algorithm. Recall that $t = n + 2$ and let $T = 2^t$.

Let $\widetilde{\varphi}$ be the best n bit approximation to φ, that is, $b = \widetilde{\varphi}T \in \{0, T-1\}$ and $0 \leq \delta < 2^{-t}$, where $\delta = \varphi - \widetilde{\varphi}$.

First, notice that since $|u\rangle$ is an eigenvector of U, the state of the phase qubits after step 3 is

$$\begin{aligned} |\phi_3\rangle &= \frac{1}{\sqrt{2^t}}(|0\rangle + \exp(2\pi i 0.\varphi_t)\,|1\rangle) \cdots (|0\rangle + \exp(2\pi i 0.\varphi_1 \ldots \varphi_t)\,|1\rangle) \\ &= \frac{1}{\sqrt{2^t}} \sum_{k=0}^{T-1} \exp(2\pi i \varphi k)\,|k\rangle \end{aligned}$$

If we apply the inverse QFT we have

$$|\phi_4\rangle = \frac{1}{2^t} \sum_{k=0}^{T-1} \sum_{l=0}^{T-1} \exp\left(\frac{-2\pi i k l}{2^t}\right) \exp(2\pi i \varphi k)\,|l\rangle$$

If we reorganize terms we have that the amplitude of $|(b+l) \mod T\rangle$ is

$$\alpha_l = \frac{1}{2^t} \sum_{k=0}^{T-1} \left(\exp\left(2\pi i(\varphi - (b+l)/2^t)\right)\right)^k.$$

The sum is a geometric series, so we can give the following closed expression

$$\begin{aligned} \alpha_l &= \frac{1}{2^t} \frac{1 - \exp(2\pi i(\varphi T - (b+l)))}{1 - \exp(2\pi i(\varphi - (b+l)/T))} \\ &= \frac{1}{2^t} \frac{1 - \exp(2\pi i(\delta T - l))}{1 - \exp(2\pi i(\delta - (b+l)/T))} \end{aligned}$$

We now check the probability of getting an n-bit approximation. For that to occur, the maximum error between the measured result m and the desired result b, must be lower than $e = 2^{t-n} - 1 = 2^{n+2-n} - 1 = 3$.

We bound the probability of having an error higher than this for a measured result m:

$$P[|m-b|>e]) = \sum_{l=-2^{t-1}+1}^{-(e+1)} |\alpha_l|^2 + \sum_{l=e+1}^{2^{t-1}} |\alpha_l|^2. \tag{7.1}$$

By the triangle inequality, if $\theta \in \mathbb{R}$, $e^{i\theta} \in \mathbb{T}$, so $|1-e^{i\theta}| \leq 2$, so

$$|\alpha_l| \leq \frac{2}{2^t |1-\exp(2\pi i(\delta - l/T))|}.$$

If $|\theta| \in [-\pi,\pi]$, then $|1-e^{i\theta}| > 2|\theta|/\pi$. Let $\theta = 2\pi(\delta - l/T)$. $\theta \in [-\pi,\pi]$, since $|l| \leq 2^{t-1}$. Therefore

$$|\alpha_l| \leq \frac{1}{2^{t+1}(\delta - l/T)} \tag{7.2}$$

If we combine eq. 7.1 and eq. 7.2, we have

$$P[|m-b|>e] \leq \frac{1}{4}\left(\sum_{l=-2^{t-1}+1}^{-(e+1)} \frac{1}{(l-T\delta)^2} + \sum_{l=e+1}^{2^{t-1}} \frac{1}{(l-T\delta)^2}\right).$$

Since $T\delta \in [0,1]$:

$$P[|m-b|>e] \leq \frac{1}{4}\left(\sum_{l=-2^{t-1}+1}^{-(e+1)} \frac{1}{(l-1)^2} + \sum_{l=e+1}^{2^{t-1}} \frac{1}{(l-1)^2}\right),$$

changing the indices and joining both sums,

$$P[|m-b|>e] \leq \frac{1}{2}\sum_{l=e}^{T-1} \frac{1}{l^2}.$$

Lastly, we bound this sum from above by the corresponding integral,

$$P[|m-b|>e] \leq \frac{1}{2}\int_e^{T-1} \frac{\mathrm{d}l}{l^2} = \frac{1}{2(e-1)} = \frac{1}{4}.$$

Hence the algorithm will be correct with probability at least $3/4 > 2/3$.

□

7.3.1. Quipper implementation

This algorithm is also implemented on the file `src/lib/Algorithms/QFT.hs`. For the reverse Quantum Fourier Transform we use Quipper's `reverse_generic_endo`, which

reverses a function with shape `f :: QShape qa ba ca => qa -> Circ qa`.

Having this, we need three elements for the quantum phase estimation algorithm:

1. A function `operator :: Int -> (qa -> Circ qa)` that gives us the powers of the unitary operator of which we want to calculate the eigenvalues,
2. The eigenvector, `eigv :: qa` (or superposition of eigenvectors if necessary),
3. the desired precision, `n :: Int`.

Given these arguments the function initializes the phase qubits to a uniform superposition, and applies the unitary operator raised to binary powers controlling on each qubit. It then applies the reversed Quantum Fourier transform. The complete code is available in the following snippet.

```
estimatePhase
  :: (QData qa) => (Int -> (qa -> Circ qa)) -> qa -> Int -> Circ [Qubit]
estimatePhase operator eigv n = do
  phase <- qinit (replicate t False)
  phase <- map_hadamard phase
  eigv  <- forEach (numbered phase)
                   (\(i, q) qx -> operator (2 ^ i) qx `controlled` q)
                   eigv
  phase <- box "RQFT" reverseQft phase
  qdiscard eigv
  pure phase
  where t = n + 2
```

Here we have used the `forEach` combinator, that combines a number of monadic maps into a single one. Its type signature is:

```
forEach :: (Monad m) => [b] -> (b -> a -> m a) -> (a -> m a)
```

7.4. Order finding

As a direct application of Theorem 7.4 and as an intermediate step towards proving Theorem 7.1, we prove that the order of an element in $U(\mathbb{Z}_N)$ can be calculated in polynomial quantum time, that is, we can solve the problem:

Problem 7.3 (Order finding)**:** *Calculate the order of an element x in the group of units $U(\mathbb{Z}_N)$.*

- ***Input:*** *Two integers x and N.*
- ***Promise:*** *$x \in U(\mathbb{Z}_N)$, that is, $0 < x < N$ and $\gcd(x, N) = 1$.*
- ***Output:*** *The order of x as an element of $U(\mathbb{Z}_N)$, that is, the least integer r such that $x^r = 1$ in $U(\mathbb{Z}_N)$.*

We show how to solve it by using Algorithm 2. Notice that if Algorithm 2 is applied with

a superposition of eigenvectors $\sum \alpha_i |u_i\rangle$ as inputs, the output will be a superposition of estimations of its eigenvalues, $\sum \alpha_i \left|\widetilde{\varphi_i}\right\rangle$. This fact will be later used again for solving the quantum counting problem.

For stating the theorem, we need to make use of the *continued fractions algorithm*.

Definition 7.4: *Given a decimal number $\alpha \in \mathbb{R}$ with n bits, its* continued fraction *is a sequence $a_0, a_1, \ldots$ such that*

$$\alpha = a_0 + \frac{1}{a_1 + \frac{1}{a_2 + \ldots}}.$$

The k^{th} convergent *of α is the rational number given by considering the first k terms of a_n.*

If α is a whole number, then the sequence has length 1 and $a_0 = \alpha$. Otherwise, we can find it recursively: $a_0 = \lfloor \alpha \rfloor$ and if b_n is the continued fractions sequence for $1/(\alpha - a_0)$, then $a_n = b_{n-1}$. This gives rise to a simple polynomial algorithm.

The following lemma will be needed for solving Problem 7.3 and it is stated without proof.

Lemma 7.5: *(Nielsen and Chuang (2010), thm. 5.1)*

Let $p/q \in \mathbb{Q}$ and $\varphi \in \mathbb{R}$ such that

$$|p/q - \varphi| \leq \frac{1}{2q^2}.$$

Then p/q is a convergent of the continued fraction for φ.

Given this lemma we can prove:

Theorem 7.6: *Problem 7.3 is solvable in quantum polynomial time.*

Proof.

Let us consider the unique unitary map U that maps, for $j, k \in 0, \ldots, N-1$ (expressible in n qubits each),

$$|k\rangle \mapsto \begin{cases} \left|x^j k \mod N\right\rangle & \text{if } k \leq N \\ |k\rangle & \text{otherwise.} \end{cases}$$

This map is efficiently computable for powers 2^j via binary exponentiation. It is unitary since x and N are coprime.

Let $r = \mathrm{ord}_{\mathbb{Z}_N}(x)$ and $s \in \{0, \ldots, r-1\}$. Let

$$|u_s\rangle = \frac{1}{\sqrt{r}} \sum_{k=0}^{r-1} \exp\left(-2\pi i k \frac{s}{r}\right) \left|x^k \mod N\right\rangle.$$

For all $s \in \{0, \ldots, r-1\}$, $|u_s\rangle$ is an eigenvector of U since

$$\begin{aligned} U|u_s\rangle &= \frac{1}{\sqrt{r}} \sum_{k=0}^{r-1} \exp\left(-2\pi i k \frac{s}{r}\right) \left|x^{k+1} \mod N\right\rangle \\ &= \exp\left(2\pi i \frac{s}{r}\right) |u_s\rangle . \end{aligned}$$

The phase is thus s/r.

Furthermore, consider that

$$\begin{aligned} \frac{1}{\sqrt{r}} \sum_{s=0}^{r-1} |u_s\rangle &= \frac{1}{r} \sum_{s=0}^{r-1} \sum_{k=0}^{r-1} \exp\left(-2\pi i k \frac{s}{r}\right) \left|x^k \mod N\right\rangle \\ &= \frac{1}{r} \sum_{k=0}^{r-1} \left(\sum_{s=0}^{r-1} \exp\left(-2\pi i k \frac{s}{r}\right) \right) \left|x^k \mod N\right\rangle = \frac{1}{r} \sum_{k=0}^{r-1} r\delta_{k0} \left|x^k \mod N\right\rangle \\ &= \left|x^0 \mod N\right\rangle = |0 \ldots 01\rangle \end{aligned}$$

Consider applying Algorithm 2 with unitary map U and eigenvector $|0 \ldots 01\rangle$. Clearly, the output will be an approximation of the phase of one of the eigenvalues of some $|u_s\rangle$.

Hence, the output will be an approximation of $\frac{s}{r}$. If the approximation is accurate enough, the period r can be recovered from the decimal expression of this fraction by using Lemma 7.5.

□

7.4.1. Quipper implementation

Modular exponentiation and oracle

Binary exponentiation is needed for both calculating factors and for creating the oracle circuit used by the quantum phase estimation algorithm. The Haskell code for binary exponentiation is

```
binaryExp :: Integer -> Integer -> Integer -> Integer
binaryExp x 0 m = 1
binaryExp x a m = binaryExp (x ^ 2 `mod` m) q m `mod` m * (x `mod` m) ^ r
  where (q, r) = a `divMod` 2
```

where `binaryExp x a m` computes $x^a \mod m$. The function `divMod` returns the quotient and remainder of a given number.

For the quantum oracle we use the `QuipperLib.Arith` module, that defines a series of integer types usable in the quantum setting. These are `IntM` for parameters and `QDInt` for quantum integers. We define the function `quantumOp x n j` that returns a circuit

that maps

$$|y\rangle \mapsto \left|x^j y \mod n\right\rangle .$$

The code is as follows:

```
quantumOp :: Integer -> Integer -> Integer -> QDInt -> Circ QDInt
quantumOp x n j y = do
  (y,z) <- q_mult_param a y -- x^n mod n * y
  q_n <- qinit (fromIntegral n) -- q_n
  (z, q_n, res) <- q_mod_unsigned z q_n
  pure res -- x^n*y mod n
  where
    a :: IntM
    a = fromIntegral $ binaryExp x j n
```

We first calculate $x^j \mod n$ using the `binaryExp` function. We then multiply `y` by this using `q_mult_param` (quantum multiplication with classical parameter).

Then we apply the modulus `n` again using `q_mod_unsigned` (for which we need to transform `n` into a quantum integer `q_n`).

Order finding algorithm

First, we estimate the phase by simply applying the phase estimation algorithm.

```
getOrder :: Integer -> Integer -> Circ [Qubit]
getOrder x n = do
  eigv <- qinit 1
  estimatePhase (quantumOp x n) eigv (2 + ceiling (logBase 2 (fromIntegral n)))
```

We then would have to calculate the convergents of the phase until we find one that has as its denominator the phase that we are looking for. For this we need to implement the continued fractions algorithm, which is straightforward, and then calculate each convergent. This is available at the `src/lib/Floating.hs` module. What remains is classically recovering the divisor from its order if possible.

This algorithm can't be feasibly executed since it takes too much time. Therefore, only the circuit is generated.

7.5. Classical part

The classical part of Shor's algorithm is a randomized reduction of the decision problem associated with FACTORING to Problem 7.3. To prove that such reduction works, we need to present two auxiliary lemmas from (Nielsen and Chuang (2010)).

Lemma 7.7: *Let N be a composite integer and $x \in \mathbb{Z}_N$, $x \notin \{-1, 1\}$ an element such that $x^2 = 1$.*

Then $\gcd(x-1, N)$ is a non-trivial divisor of N and it is computable in classical time $O(\log^3 N)$.

Proof.

Since $x^2 = 1$ in $\mathbb{Z}_N$, we have that $N | x^2 - 1 = (x-1)(x+1)$. Furthermore, $x-1, x+1 < N$, and hence $N \nmid x-1$ and $N \nmid x+1$.

Let k be such that $x^2 - 1 = Nk$. Assume $x-1$ and N are coprime. Then, by applying Bézout's identity, we have that there exists a, b such that

$$aN + b(x-1) = 1.$$

Multiplying by $x+1$ we have

$$aN(x+1) + b(x^2-1) = x+1 \implies N(a(x+1) + bk) = x+1.$$

Therefore, $N | x+1$, but this is a contradiction.

Therefore, $1 < \gcd(x-1, N) < N$, and thus it is a non-trivial divisor of N. We may then compute it using the well-known Euclidean algorithm, which has the desired complexity.

□

Lemma 7.7 hints at the reduction used by Shor's algorithm: if we can find a non-trivial solution to

$$x^2 \equiv_N 1, \tag{7.3}$$

then we can find a non-trivial divisor of N.

In particular, suppose we have $a \in \mathbb{Z}_N$, we compute its order using Theorem 7.6 and it turns out to be even. Then, by choosing $x = a^{r/2}$, we have a solution to eq. 7.3. If it is not trivial we can apply Lemma 7.7 to get a non-trivial factor of N.

The following lemma shows that this happens with enough probability if we sample $a \in U(\mathbb{Z}_N)$ uniformly.

Lemma 7.8: *(Nielsen and Chuang (2010), thm. A4.13)*

Let N be an odd positive composite integer with more than one prime factor. Then, if x is sampled uniformly from $U(\mathbb{Z}_N)$ and $r = \mathrm{ord}_{U(\mathbb{Z}_N)}(x)$ we have

$$P[r \text{ is odd or } x^{r/2} \equiv_N -1] \leq \frac{1}{3}.$$

Proof.

We show that

$$P[r \text{ is odd or } x^{r/2} \equiv_N -1] \leq \frac{1}{4}.$$

Let $N = \prod_{i=1}^{m} p_i^{n_i}$ By the Chinese remainder theorem, we have that

$$U(\mathbb{Z}_N) \cong \prod_{i=1}^{m} U(\mathbb{Z}_{p_i^{n_i}}),$$

hence sampling uniformly $x \sim U(\mathbb{Z}_N)$ is equivalent to sampling uniformly and independently $x_i \sim U(\mathbb{Z}_{p_i^{n_i}})$, and x would be the unique element such that $x \equiv_{p_i^{n_i}} x_i$ for each i.

Let $r_i = \text{ord}_{U(\mathbb{Z}_{p_i^{n_i}})}(x_i)$ and let

1. 2^d be the largest power of 2 such that $2^d | r$ and
2. for each i, 2^{d_i} the largest power of 2 such that $2^{d_i} | r_i$.

Assume r is odd. For all i, $r_i | r$, and hence r_i is odd, so $d_i = 0$.

Assume r is even and $x^{r/2} \equiv_N -1$. Then for all i, $x^{r/2} \equiv_{p_i^{n_i}} -1$ and hence $r_i \nmid r/2$, but $r_i | r$. Therefore, $d_i = d$.

In any of these cases we have that d_i is equal to the same value for all i (0 or d). Consider the following claim:

Claim. (Nielsen and Chuang (2010), lemma A4.12)

Let $p \neq 2$ be prime, $n \geq 1$ be an integer and 2^d be the largest power of 2 such that $2^d | \varphi(p^n)$, where φ is Euler's phi function. Then

$$P\left[2^d | \text{ord}_{U(\mathbb{Z}_{p^n})}(x)\right] = \frac{1}{2},$$

where x is sampled uniformly from $U(\mathbb{Z}_{p^n})$.

Proof.

$\varphi(p^n) = p^{n-1}(p-1)$ is even and therefore $d \geq 1$.

Let $x \in U(\mathbb{Z}_{p^n})$ and $r = \text{ord}_{U(\mathbb{Z}_{p^n})}(x)$. Since p is prime, $U(\mathbb{Z}_{p^n})$ is a cyclic group. Let $U(\mathbb{Z}_{p^n}) = \langle a \rangle$, so that there exists some $k \in \{1, \ldots, \varphi(p^n)\}$ such that $a^k = x$.

If k is odd, then $a^{kr} = 1$ and hence $\varphi(p^n) | kr$, therefore $2^d | r$.

If k is even then

$$a^{k/2\varphi(p^n)} = (a^{\varphi(p^n)})^{k/2} = a^{k/2} = 1,$$

therefore $r | \varphi(p^n)/2$ and hence $2^d \nmid r$.

□

This claim shows then that, by applying the bound for each i and using independence,

$$P[r \text{ is odd or } x^{r/2} \equiv_N -1] \leq \frac{1}{2^m} \leq \frac{1}{4},$$

where the last inequality follows from N having more than one prime factor.

□

Hence, we can apply the previously sketched-out algorithm to odd integers with more than one factor. What remains is only checking the other cases.

We can check if N is even by looking at its last digit on its binary representation, which takes constant time on a classical family of circuits. For the remaining case, that is, $N = a^b$, with $a \geq 1, b \geq 2$, we may approximate $\sqrt[b]{N}$ for every $b \in \{2, \dots \lceil \log N \rceil\}$ and check whether it is an integer.

Hence, Shor's algorithm is complete and is presented at Algorithm 3.

Algorithm 3 (Shor's algorithm)**:** *(Nielsen and Chuang (2010), sec. 5.3)*

Solves Problem 7.1.

1. *Check if N is even, if so* ***return*** *2.*
2. *Check if $N = a^b$, for integers $a \geq 1, b \geq 2$, if so* ***return*** *a.*
3. *Sample x uniformly from $\mathbb{Z}_N \backslash \{0\}$.*
4. *Check if $\gcd(x, N) > 1$, if so* ***return*** *$\gcd(x, N)$.*
5. *Find $r = \text{ord}(x, N)$ by applying Algorithm 2.*
6. *If r is odd or $x^{r/2} \not\equiv_N -1$,* ***fail****.*
7. ***Return*** *$\gcd(x^{r/2} - 1, N)$.*

By the previous discussion this proves Theorem 7.1. For finding *all* factors of a given number we may repeat Algorithm 3 and successively divide the integer until we reach a base case. This would also take a polynomial amount of time, $O(\log^4 N)$.

7.5.1. Implementation

The complete algorithm for factorization is available in the attached code at the file `src/lib/Algorithms/Shor.hs`. First, we define a datatype that deals with the possible modes of failure of the algorithm, indicating whether the integer is one (`One`), a prime number (`Prime`), a bad order was found (`BadOrder`) or any other case of failure of the intermediate algorithms (`Other`).

```
data Failure = One
             | Prime
```

```
  | BadOrder Integer Integer
  | Other
```

Power of a number

First, we look at the case where $N = a^b$. For this we define the function `getPower`, given by the following (simplified[1]) code snippet

```
getPower :: Integer -> Either Failure Integer
getPower n = base <$> exponent
 where
  a :: Double
  -- We check every possible root up to `a`
  a = ceiling $ logBase 2 n'

  base :: Double -> Double
  -- The base for a given exponent
  base x = round (n' ** (1 / x))

  exponents :: [Double]
  -- List of possible exponents, from biggest to smallest
  exponents = filter (\x -> n == round ((base x) ** x)) [a, a - 1 .. 2]

  exponent :: Either Failure Double
  exponent = if null exponents then Left Other else Right (head exponents)
```

We first calculate a list of possible `exponents`. They can range from 2 to $a = \lceil \log_2 N \rceil$. We filter those that give that are true roots up to the rounding error given by the square root algorithm, and finally get the first of them in `exponent`.

We then calculate that square root (the `base` for the chosen `exponent`) and return it.

Main factorization function

The main factorization function, `factor`, takes an integer and returns a valid factor. It distinguishes several cases. It is an IO function since we need to make use of randomness.

```
factor :: Integer -> IO (Either Failure Integer)
factor n
  | n < 0 = pure $ Right (-1)
  | n == 1 = pure $ Left One
```

[1]Explicit management of numeric types conversion has been omitted from this and the following snippets. The complete code can be checked in the attached files

```
  | isPrime n = pure $ Left Prime
  | n `mod` 2 == 0 = pure $ Right 2
  | isRight root = pure root
  | otherwise = do
    x <- randomRIO (1, n - 1)
    if gcd x n > 1 then pure (Right (gcd x n)) else factorFromOrder x n
  where root = getPower n
```

The different cases are

1. The integer is negative, then we return -1.
2. The integer is 1 or it is prime. We return the corresponding failure code. The prime factorization is done using a Haskell library that implements the Baille-PSW primality test (Pomerance, Selfridge, and Wagstaff (1980)).
3. The integer is even, we then return 2.
4. The integer is of the form $N = a^b$, checked by the implemented algorithm in the previous section.

Otherwise, we follow the algorithm and apply the quantum case that has previously been described. For the full factorization, we would apply a recursive algorithm that factors each number until we get to a base case: either the number is one or it is prime.

This algorithm is not feasibly executable in practice because of the number of qubits needed in the quantum estimation phase, thus an assesment is made as to what the procedure would be and what amount of resources would be needed for calculating the factors of a number in this way. This assesment can be seen by running the `quantum` executable in mode `shor`.

8. Grover's algorithm

In this section we present *Grover's algorithm*, a quantum algorithm that can be used to solve search and minimization problems, providing polynomial speedup in the query complexity setting with respect to the best possible classical algorithm.

The chapter is organized as follows: first we present the general problem Grover's algorithm attempts to solve and why it can be potentially useful.

Then, we describe the classical algorithm and the quantum Grover's algorithm with a known number of solutions. Finally, we show how to obtain the number of solutions and prove that it is asymptotically optimal in the query complexity setting.

8.1. Search problems

The general problem Grover's algorithm attempts to solve is a *search problem*. It is a very general problem with many potential applications.

Problem 8.1 (Search)**:** *Search a string that has a certain property.*

- ***Input:*** *A function* $f : \mathbb{B}^n \to \mathbb{B}$.
- ***Promise:*** *There is at least one solution to the equation* $f(x) = 1$.
- ***Output:*** *A string* x *such that* $f(x) = 1$.

As a possible application, consider a language $L \in \mathsf{NP}$, and fix a word $x \in \mathbb{B}^n$ (if we allow randomness, we can consider $L \in \mathsf{MA}$ and proceed similarly).

Recalling Proposition 2.5 consider as $f : \mathbb{B}^{p(n)} \to \mathbb{B}$ the function associated with the verifier V. Then, if we have an algorithm to solve the search problem we can decide whether $x \in L$ or not. Given the wide variety of problems of practical usefulness in NP (Arora and Barak (2009), chap. 2), this is an important and useful application of such problem.

As a more practical application, one can consider a search on a database. In this case the function would indicate whether there has been a match or not for each item in the database. Notice that we allow for potentially more than one matching string.

8.2. The classical case

In the query complexity setting we are not interested in the time it takes to compute f but rather in the number of queries necessary to provide an answer.

In the classical case, there is a trivial algorithm that provides an answer: simply check for every possible input until a match is found. The query complexity of this algorithm is $O(N) = O(2^n)$.

Even if f takes unit time to compute, this algorithm is not efficient for solving the search problem associated with an NP language since it would take at least exponential time on the size of word being decided.

Furthermore, this algorithm is clearly optimal in the classical setting: in the worst case the last inputs checked are the matching ones and thus $\Omega(N)$ queries are needed.

Randomness does not add an advantage. With a limited number k of queries, the best possible random algorithm would be to randomly sample k words without replacement from the set of possible inputs, in which we would have a probability of success of k/N. If bounded error is required, then we must do a linear amount of queries (e.g. $k = 2N/3$) to answer, and thus asymptotically the query complexity is equal to the classical case (Kaye, Laflamme, and Mosca (2007), sec 8.1).

As we shall see in the next section, Grover's algorithm builds on this idea and uses the features of quantum computing to *amplify* the probability of outputting a correct answer in order to provide

8.3. Grover's algorithm

Firstly, we need to define Grover's operator, an operator associated with a given oracle.

Consider the following operation.

Definition 8.1 (Diffusion)**:** *Let $n \in \mathbb{N}$. The* diffusion *operator is the operator*

$$D_n : Q^{\otimes n} \to Q^{\otimes n}$$

given by the following procedure

1. *Apply the Hadamard transform to each qubit,*
2. *Apply a phase shift such that*

$$|x\rangle \mapsto -|x\rangle \text{ if } x \neq 0, \quad |0\rangle \mapsto |0\rangle,$$

3. *apply the Hadamard transform again to each qubit.*

It can be done with a linear number of gates; a circuit performing the diffusion operator for $n = 3$ qubits can be seen in fig. 8.1.

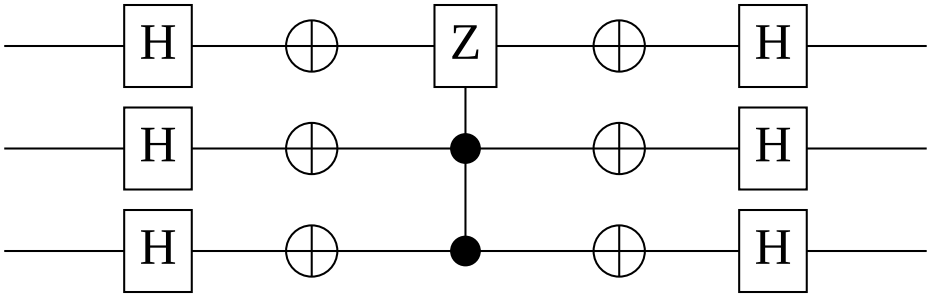


Figure 8.1.: Diffusion operator for $n = 3$ qubits. The "Z" gate is the phase change gate for an angle of $\theta = \pi$.

Using the diffusion procedure, we now define *Grover's operator*, which is associated with a given oracle. We omit dealing with the auxiliary qubits, since these are only altered by the oracle. Recall that for a given function $f : \mathbb{B}^n \to \mathbb{B}$, its oracle $U_f : Q^{\otimes n+1} \to Q^{\otimes n+1}$ is the unique linear map that maps

$$|x\rangle |y\rangle \mapsto |x\rangle |y \oplus f(x)\rangle .$$

Definition 8.2 (Grover Operator)**:** *Given an oracle U_f, its* Grover's operator,

$$G_f : Q^{\otimes n+1} \to Q^{\otimes n+1}$$

is given by the following procedure

1. *Apply the oracle U_f to all qubits and*
2. *apply D_n to all but the last qubit.*

This operator is crucial in the description of Grover's algorithm but also in the quantum counting and quantum existence algorithms. When used by algorithms, we will always assume that the last qubit input, that is, qubit $|y\rangle$, is $H|1\rangle$, so that by Lemma 6.2, we can see the oracle as performing in all but the last qubit the map

$$|x\rangle \mapsto (-1)^{f(x)} |x\rangle .$$

Grover's operator can be given a geometrical interpretation that we can see in the following lemma.

Lemma 8.1: *Let $f : \mathbb{B}^n \to \mathbb{B}$ and $M = |f^{-1}(1)|$.*

Let $|\psi\rangle$ be a uniform superposition, that is,

$$|\psi\rangle = H^{\otimes n} |0\rangle^{\otimes n} = \frac{1}{\sqrt{N}} \sum_{y \in \mathbb{B}^n} |y\rangle$$

and $|\beta\rangle$ be the uniform superposition of solutions to the equation $f(x) = 1$, that is

$$|\beta\rangle = \frac{1}{\sqrt{M}} \sum_{x \in f^{-1}(1)} |x\rangle .$$

The restriction of Grover's operator to the subspace spanned by $|\psi\rangle$ and $|\beta\rangle$ is a rotation, whose angle θ verifies

$$\cos \frac{\theta}{2} = \sqrt{\frac{N-M}{N}}.$$

Proof.

The subspace spanned by $|\psi\rangle$ and $|\beta\rangle$ can be given by the orthonormal basis $\{|\alpha\rangle, |\beta\rangle\}$, where $|\alpha\rangle$ is the uniform superposition of those vectors that are not a solution to $f(x) = 1$,

$$|\alpha\rangle = \frac{1}{\sqrt{N-M}} \sum_{x \in f^{-1}(0)} |x\rangle .$$

In particular, expressed in terms of the basis we have

$$|\psi\rangle = \sqrt{\frac{N-M}{N}} |\alpha\rangle + \sqrt{\frac{M}{N}} |\beta\rangle .$$

The restriction of G_f to $\mathrm{Lin}(|\alpha\rangle, |\beta\rangle)$ is composed by two reflections

1. U_f is a reflection about $|\alpha\rangle$, since

$$U_f(a |\alpha\rangle + b |\beta\rangle) = a |\alpha\rangle - b |\beta\rangle .$$

2. The diffusion operator is an operator of the form $D_n = H^{\otimes n} O H^{\otimes n}$, where the phase shift O is a reflection about the vector $|0\rangle^{\otimes n}$. Thus, we can easily check that since $H^{\otimes n}$ is its own inverse, D_n is a reflection about the vector $|\psi\rangle = H^{\otimes n} |0\rangle^{\otimes n}$.

The composition of two reflections is a rotation, and the angle of rotation θ will be twice the one between the vectors that give rise to the reflections, $|\psi\rangle$ and $|\alpha\rangle$. Hence, we have

$$\cos \frac{\theta}{2} = \langle \psi | \alpha \rangle = \sqrt{\frac{N-M}{N}}.$$

□

This gives us a rough idea of the algorithm that we can follow to get a possible answer: rotate some vector in the plane so as to minimize its distance from $|\beta\rangle$ and measure the result. This idea is formalized in Algorithm 4. We assume the number of solutions is known in advance.

Algorithm 4 (Grover's algorithm)**:** *(Kaye, Laflamme, and Mosca (2007), sec 8.1)*

Solves: *The search problem, Problem 8.1, given the number of solutions* $M = |f^{-1}(1)|$.

1. *Pick a random solution and check whether it is a match. If so, output and stop.*
2. *Initialize an n qubit register to* $|0 \dots 0\rangle$ *and a last qubit to* $|1\rangle$.
3. *Apply the H gate to each qubit to obtain an uniform state in the first n qubits,* $|\psi\rangle$ *and the state* $|\downarrow\rangle = (|0\rangle - |1\rangle)/\sqrt{2}$ *in the last one.*
4. *Apply* $T = \lfloor \frac{\pi}{4}\sqrt{\frac{N}{M}} \rfloor$ *times Grover's operator* G_f.
5. *Measure the n qubit register and output the result.*

A schematic for Grover's algorithm can be seen in fig. 8.2.

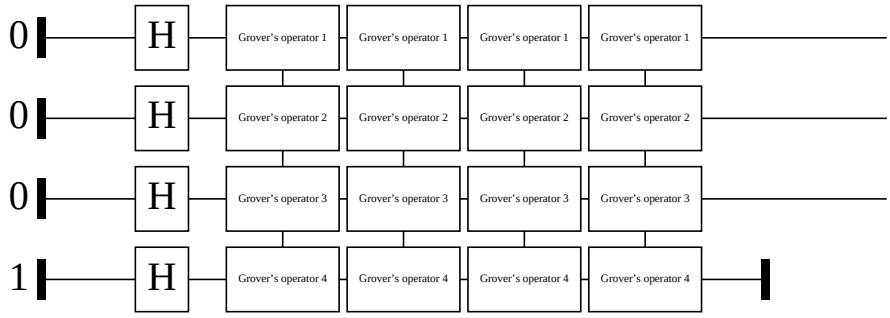


Figure 8.2.: A circuit for Grover's algorithm with $n = 5$ input qubits, $T = 4$ repetitions (the optimal number is 1)

Lastly, we prove that Grover's algorithm outputs the correct answer with bounded error.

Theorem 8.2 (Correctness of Grover's algorithm)**:** *(Kaye, Laflamme, and Mosca (2007), sec. 8.1)*

Let $f : \mathbb{B}^n \to \mathbb{B}$ *be a function such that* f *is not constantly zero. Then Algorithm 4 succeeds with bounded error.*

Proof.

The proof is adapted from (Kaye, Laflamme, and Mosca (2007), sec. 8.1).

Let $M = |f^{-1}(1)|$. If $M \geq N/2$ then the probability of getting a match on the first step is over one half. Assume for the remainder of the proof that $M < N/2$.

Since the last qubit remains unaltered after step 1, we focus on the n qubit register only. Using the expression of $|\psi\rangle$ in the proof of Lemma 8.1, we can see that

$$|\psi\rangle = \cos\frac{\theta}{2}|\alpha\rangle + \sin\frac{\theta}{2}|\beta\rangle,$$

hence by Lemma 8.1 we have that

$$G^k|\psi\rangle = \cos\left(\frac{2k+1}{2}\theta\right)|\alpha\rangle + \sin\left(\frac{2k+1}{2}\theta\right)|\beta\rangle.$$

To get a probability of success close to one, we would like $\sin\left(\frac{2k+1}{2}\theta\right) \approx 1$, thus we should have $\frac{2k+1}{2}\theta \approx \pi/2$.

For a real k' picking

$$k' = \frac{\pi}{2\theta} - \frac{1}{2}$$

would get us the exact result.

By Lemma 8.1, we now that

$$\theta = 2\arcsin\left(\sqrt{\frac{M}{N}}\right) \approx 2\sqrt{\frac{N}{M}},$$

where the error of the approximation is under one half since $M < N/2$.

Thus, if we pick

$$k = \left\lfloor \frac{\pi}{4}\sqrt{\frac{M}{N}} \right\rfloor$$

we have $k - k' \leq 1/2$ and therefore

$$\frac{2k+1}{2}\theta = \frac{\pi}{2} + \varepsilon, \text{ where } \varepsilon \in O\left(\frac{1}{\sqrt{N}}\right)$$

thus $\sin(\frac{\pi}{2} + \varepsilon) = \cos(\varepsilon) \geq 1 - \frac{\varepsilon^2}{2} = 1 - O\left(\frac{1}{N}\right)$. where we have used the second order approximation of the cosine by its Taylor series.

□

8.3.1. Quipper implementation

Grover's algorithm is implemented on `src/lib/Algorithms/Grover.hs`. It can be run by using the `grover` subcommand on the `quantum` binary in the attached code.

First, we need to define the diffusion operator. For this we first define the phase shift, that negates all qubits, applies a π conditional rotation to one of them, and negates all qubits again.

```
phaseShift :: [Qubit] -> Circ [Qubit]
phaseShift (x:xs) = do
  x:xs <- mapUnary qnot (x : xs)
  x    <- gate_Z x `controlled` xs
  mapUnary qnot (x : xs)
```

The `controlled` operator can make an operation controlled on a list of qubits.

The diffusion operator is then defined by composing this monadic function with the monadic functions `map_hadamard`, that apply the Hadamard gate to each qubit.

```
diffusion :: [Qubit] -> Circ [Qubit]
diffusion = map_hadamard >=> phaseShift >=> map_hadamard
```

This allows us to define Grover's operator for a given oracle. We define it by composing the oracle's circuit with the diffusion operator, taking care to leave the lower qubit (the one that has state $|\downarrow\rangle$) unchanged.

```
groverOperator :: Oracle [Qubit] -> ([Qubit], Qubit) -> Circ ([Qubit], Qubit)
groverOperator oracle (xs, y) = do
  (xs, y) <- circuit oracle (xs, y)
  xs      <- diffusion xs
  pure (xs, y)
```

Finally, this allows us to define Grover's algorithm for a given number of solutions. For this we need the function `timesM`, defined in module `src/lib/Utils.hs`, that allows us to raise a monadic function to a certain power (effectively working as a loop with a fixed number of iterations).

```
grover :: Int -> Oracle [Qubit] -> Circ [Bit]
grover m oracle = do
  (x, y) <- qinit (qc_false (shape oracle), True)
  (x, y) <- map_hadamard (x, y)
  (x, y) <- n `timesM` box "Grover's operator" (groverOperator oracle) $ (x, y)
  qdiscard y
  measure x
  where n :: Int
        n = round $ pi / 4 * sqrt $ inputSize oracle / fromIntegral m
```

As we see, we first initialize the x register to a uniform superposition and the y register to state $|\downarrow\rangle$. We then apply Grover's operator a number of times given by the previously calculated formula, and, discard the auxiliary qubit y and finally we measure the bits to get an answer. To boost the probability of correctness, we run the algorithm a fixed number of times.

8.4. Quantum counting

Algorithm 4 requires the number of solutions to be known in advance. Furthermore, we need to check the promise that the number of solutions is non-zero. Therefore, we need to solve the following problem.

Problem 8.2 (Counting)**:** *Count the number of solutions to an equation.*

- ***Input****: A function* $f : \mathbb{B}^n \to \mathbb{B}$
- ***Output****:* $M = |f^{-1}(1)|$

The answer to solving it lies in using of the quantum phase estimation algorithm, Algorithm 2.

The geometrical characterization given by Lemma 8.1 tells us that Grover's operator is a rotation of an angle $\theta/2$ such that

$$\sin^2\left(\frac{\theta}{2}\right) = \frac{M}{N}. \tag{8.1}$$

Thus, if we can estimate the angle, then, since N is known, we may estimate M.

It is easy to check that the eigenvalues of the matrix are $\exp(\pm i\theta/2)$ corresponding to eigenvectors that add up to the uniform superposition, $|\psi\rangle$ (Kaye, Laflamme, and Mosca (2007), sec. 8.3).

Thus, by applying Algorithm 2 with enough accuracy, we can estimate either $\theta/2$ or $2\pi - \theta/2$, which is enough to estimate θ and thus M.

It turns out that $O(\sqrt{N})$ queries are enough to obtain an exact estimate. A lower accuracy might be used if we just want to check if $M > 0$, though it is asymptotically equivalent (Nielsen and Chuang (2010), sec. 6.3). We present the algorithm from (Kaye, Laflamme, and Mosca (2007)), but omit the details of the accuracy estimates.

Algorithm 5 (Quantum counting)**:** *(Kaye, Laflamme, and Mosca (2007), Exact Counting)*

- ***Solves:*** *Problem 8.2*

1. *Run Algorithm 2 twice with* $|\psi\rangle$ *as the eigenvector and* $t = \lceil n/2 \rceil + 6$ *qubits of precision. Call the resulting estimates* M_1, M_2.

2. *Run Algorithm 2 again with* $t = \log_2(M)$*, where*

$$M = \min\left(\left\lceil 30\sqrt{(NM_1+1)(N-NM_1+1)}\right\rceil, \left\lceil 30\sqrt{(NM_2+1)(N-NM_2+1)}\right\rceil\right)$$

and return the estimate.

Combining Algorithm 4 and Algorithm 5 gives us and algorithm that solves Problem 8.1 using $O(\sqrt{N})$ queries, which is better than any classical or randomized algorithm.

8.4.1. Quipper implementation

The quantum counting and quantum existence algorithms have been implemented on Quipper. The latter can be executed by running the subcommand `existence` of the `quantum` binary. Both are available at `src/lib/Algorithms/Count.hs`.

The quantum counting algorithm needs to much precision for it to be feasibly simulated on a laptop, therefore the code is shown for illustrative purposes and to create diagrams. Its code is a simple application of the quantum phase estimation algorithm with Grover's operator.

```
quantumCount :: Oracle [Qubit] -> Int -> Circ [Qubit]
quantumCount oracle t = do
  eigv <- qinit (qc_false (shape oracle), True)
  eigv <- map_hadamard eigv
  estimatePhase (`timesM` groverOperator oracle) eigv t
```

Given this phase, we estimate the number of solutions by transforming this phase using eq. 8.1. The following simplified snippet presents the algorithm:

```
phaseToNum :: Oracle [Qubit] -> Int -> IO Int
phaseToNum oracle t = do
  bits <- run_generic_io (0 :: Double) (quantumCount oracle t)
  let phi = bitsToFloating bits
  let theta = if phi <= 0.5 then 2 * pi * phi else 2 * pi * (1 - phi)
  pure $ ceiling (n * sin theta ^^ 2)
 where n = inputSize oracle
```

Checking whether the estimated phase φ is above or below $\frac{1}{2}$ tells us whether the positive or negative eigenvalue was estimated, and thus allows for the recovery of the number of solutions. The quantum existence algorithm then uses `phaseToNum` to see whether it is positive or zero.

8.5. Optimality

In the case of solving an NP problem, Grover's algorithm gives us at most a quadratic speedup replacing a query complexity of $O(2^n)$ to one of $O(\sqrt{2^n}) = O(2^{n/2})$. This is an insufficient speedup from a practical point of view, since it still leaves us with an (at least) exponential run-time. Is there any possible improvement of this technique?

The following result proves that in the query complexity model Grover's algorithm is asymptotically optimal, meaning that any other algorithm that solves the search problem will have to make $\Omega(\sqrt{N})$ queries. This does not prove of course that $\mathsf{NP} \not\subseteq \mathsf{BQP}$ or any other similar result. It does however show that the general task of solving NP problems or other search problems with an exponential search space must be attacked by using the inner structure of the problem, since any algorithm that does not do so would have a running time in $\Omega(\sqrt{N})$.

Theorem 8.3 (Grover's optimality)**:** *Algorithm 4 is asymptotically optimal in the query complexity setting; any quantum algorithm that solves Problem 8.1 needs $\Omega(\sqrt{N})$ queries to do so with bounded error.*

Proof.

We present the proof for the case where there is exactly one matching string from (Nielsen and Chuang (2010), sec. 6.6). The result can be generalized to the case where there are several matches.

Let U_x be the oracle for the function

$$f(y) = \begin{cases} 1 & \text{if } y = x \\ 0 & \text{otherwise.} \end{cases}$$

Any circuit C_n within a quantum algorithm $\{C_n\}$ can be represented as a sequence of unitary operators

$$U_x, A_1, \ldots, A_{k-1}, U_x, A_k,$$

together with a starting state $|\psi_0\rangle$. We are assuming that the quantum algorithm applies the oracle exactly k times.

Let

$$|\psi_k^x\rangle = A_k U_x \ldots A_1 U_1 |\psi_0\rangle \text{ and}$$
$$|\phi_k\rangle = A_k \ldots A_1 |\psi_0\rangle .$$

Define the *deviation after k steps* as

$$D_k = \sum_{x \in \mathbb{B}^n} \| |\psi_k^x\rangle - |\phi_k\rangle \|^2 .$$

This number measures the difference that applying the oracle k times produces.

We now prove two claims that will lead us to the result. First, the deviation is $O(k^2)$.

Claim. For all $k \in \mathbb{N}$, $D_k \leq 4k^2$.

Proof.

The proof is by induction.

Clearly, for $k = 0$, $D_k = 0$.

Let $k \in \mathbb{N}$. Then

$$\|U_x |\psi_k^x\rangle - |\phi_k\rangle\|^2 = \|U_x(\psi_k^x - \phi_k) + (U_x - I_N)\phi_k\|^2 = \|U_x(\psi_k^x - \phi_k) - 2\langle x|\phi_x\rangle |x\rangle\|^2 .$$

If we apply $\|u+v\|^2 \leq \|u\|^2 + 2\|u\|\,\|v\| + \|v\|^2$, and add up for all x, we have

$$D_{k+1} \leq \sum_{x\in\mathbb{B}^n} \|\psi_k^x - \phi_k\|^2 + 4\|\psi_k^x - \phi_k\|\,|\langle x|\psi_k\rangle| + 4|\langle x|\psi_k\rangle|^2.$$

We now apply the Cauchy-Schwarz inequality, obtaining

$$D_{k+1} \leq D_k + 4\sqrt{\sum_{x\in\mathbb{B}^n} \left\|\psi_k^x - \phi_k\right\|^2}\sqrt{\sum_{x\in\mathbb{B}^n} |\langle x|\phi_k\rangle|^2} + 4.$$

Since $|\phi_k\rangle$ is unitary, $\sum_{x\in\mathbb{B}^n} |\langle x|\phi_k\rangle|^2 = 1$, and thus we can rewrite

$$D_{k+1} \leq D_k + 4\sqrt{D_k} + 4.$$

By induction,

$$D_{k+1} \leq 4k^2 + 8k + 4 = 4(k+1)^2.$$

□

And secondly, if the algorithm outputs the correct answer with probability at least $1/2$, then the deviation must be $\Omega(N)$.

Claim. Suppose $|\langle x|\psi_k^x\rangle|^2 \geq \frac{1}{2}$ for all $x \in \mathbb{B}^n$.

Then $D_k \geq cN$ for any $c < (\sqrt{2} - \sqrt{2-\sqrt{2}})^2$.

Proof.

Let

$$E_k = \sum_{x\in\mathbb{B}^n} \|\psi_k^x - x\|^2 \text{ and } F_k = \sum_{x\in\mathbb{B}^n} \|\phi_k - x\|^2 .$$

We may multiply $|x\rangle$ by a certain constant so that $\langle x|\psi_k^x\rangle = |\langle x|\psi_k^x\rangle|$. If we do so we have

$$\||\psi_k^x\rangle - |x\rangle\|^2 = \langle\psi_k^x|\psi_k^x\rangle + \langle x|x\rangle - 2\langle\psi_k^x|x\rangle = 2 - 2|\langle\psi_k^x|x\rangle| \leq 2 - \sqrt{2},$$

hence $E_k \leq (2-\sqrt{2})N$. Furthermore, by applying the Cauchy-Schwarz inequality we have $F_k \geq 2N - 2\sqrt{N}$.

We then have, applying $\|u+v\|^2 \geq \|u\|^2 + \|v\|^2 - 2\|u\|\|v\|$ and the Cauchy-Schwarz inequality,

$$\begin{aligned}
D_k &= \sum_{x\in\mathbb{B}^n} \|(\psi_k^x - x) + (x - \phi_k)\| \\
&\geq \sum_{x\in\mathbb{B}^n} \|\psi_k^x - x\|^2 + \sum_{x\in\mathbb{B}^n} \|\phi_k - x\|^2 - 2\sum_{x\in\mathbb{B}^n} \|\psi_k^x - x\|\,\|\phi_k - x\| \\
&\geq E_k + F_k - 2\sqrt{E_k F_k} \\
&= (\sqrt{F_k} - \sqrt{E_k})^2 \\
&\geq \left(\sqrt{2} - \sqrt{2-\sqrt{2}}\right) N
\end{aligned}$$

□

Combining the previous claims we have

$$k \geq \sqrt{\frac{\left(\sqrt{2} - \sqrt{2-\sqrt{2}}\right) N}{4}},$$

and hence any algorithm must take $\Omega(\sqrt{N})$ steps.

□

Conclusions and further work

In this document I have developed an overview of quantum computation and quantum computing models, giving both the complexity theoretical perspective as well as a more applied one, by describing and implementing several key quantum algorithms.

The study of quantum computers and their properties is relatively young as a mathematical and computer science subject, yet it has seen several important successes. Quantum computers are a promising technology that has the potential to be a key tool in the resolution of several practical engineering problems. Algorithms such as Shor's algorithm are a resounding success, since they provide a practical polynomial-time algorithm for a problem that lacks such an algorithm classically, even heuristically.

Its theoretical description can also be celebrated, as it provides a succint description of this model that easily generalizes the probabilistic model.

Nonetheless, the study of this technology remains mostly theoretical, due to the difficulty of its physical implementation and the limits of quantum simulation in classical computers.

Several relevant questions were not addressed in this document. The optimality of Grover's algorithm in the query complexity model raises the need for the development of concrete algorithms for each problem. Projects such as (Jordan (2011)) describe some of these algorithms, some of which could be implemented on Quipper and be further developed.

In the theoretical realm several questions remain. Firstly, the circuit model can be feasibly generalized to other monoidal categories. For example, if the unitary matrices are restricted to have real entries, this model can be proved to be equivalent to the general quantum model (Faddeev (2009)). What about other fields or division rings? Can a simpler description of quantum computation be found?

Secondly, the study of relativized models of quantum and classical computation was not addressed in this document, yet it provides for a wide array of theoretical results that hint at the separation and incomparability of quantum classes. A recent celebrated result is the relativized proof that $\mathsf{BQP} \not\subset \mathsf{PH}$, that might lead to the resolution of problems that are not even in the polynomial hierarchy (Raz and Tal (2018)).

On the practical side, functional quantum programming languages could have stronger verification capacities with the use of dependent types for reflecting the shape of inputs and outputs or with linear types to model resource consumption and avoid the cloning

of qubits. The development of Linear Haskell, a linear types implementation on Haskell, could put some of these ideas into action on Quipper (Bernardy et al. (2017)).

Appendices

Code

The source files that generate this document are available at `github.com/mx-psi/tfg`.

The official page for the Quipper language is `mathstat.dal.ca/~selinger/quipper`. It is bundled as a `stack`-compatible package at `github.com/mx-psi/quipper`. This repository also includes a `stack` template with a ready to use project for implementing quantum algorithms in Quipper.

The code associated with this document is available at `github.com/mx-psi/quantum-algorithms`. Its documentation can be generated using `make docs`.

Code description

What follows is a formatted copy of the file `README.md`, available with the attached code.

Dependencies

All dependencies are managed by `stack`. For installing `stack` on Unix OSs `make install-deps` can be run. On Windows, a binary is provided on https://docs.haskellstack.org/en/stable/README/.

The first time there is an attempt to build the binary all dependencies will be installed by `stack`, including GHC Haskell compiler, so this may take some time.

Building

Important note: The building process has only been tested on Linux distributions. In order to run the program on Windows, the `stack.yaml` file MUST be edited so that the preprocessor works. In particular, `quipper/convert_template.sh` should be replaced by `quipper/convert_template.bat`.

`make build` will build the associated binaries and copy them to the `bin` folder.

The first time there is an attempt to build the binary all dependencies will be installed by `stack`, including GHC Haskell compiler, so this may take some time.

Binaries created on my computer (running Linux Mint 18.1) are attached.

Running the binaries

The previous process creates two binaries, Both binaries include help instructions by running them with the option `--help`.

The first one, `quantum`, implements a command-line interface for running several quantum algorithms. Each algorithm is associated with a subcommand.

Some algorithms take an oracle as input, defined by its truth table. This can be passed as an argument or, if called without this argument, the oracle will be read from standard input.

The syntax for oracles is a subset of csv files. Each line is expected to have an input value, represented by a string of `0` and `1`, and separated by a comma, an output value, represented by a single bit.

For example, the NOT logical operation can be given as

```
0,1
1,0
```

Some examples of oracles are included in the `oracles` folder.

The second binary, `diagrams`, generates the diagrams used in the document. The option `--gen-all` generates a selection of the diagrams used, while providing a circuit in the form of a truth table with the option `--circuit` generates a circuit diagram for that specific circuit.

Documentation

The documentation, generated with Haddock, is available in the `docs` folder. It can be rebuilt by using `make docs`.

Note that the process for building the documentation requires the manual preprocessing of the code files and thus can not directly be done using `stack`.

Testing

The code includes a test suite with both unit tests and property-based testing. It can be run using `make tests`.

Note that due to the nature of the implemented algorithms, it is possible (yet unlikely) that some of these tests fail (this is indicated in the output log).

Code organization

The code is organized as follows

1. the `oracles` folder includes examples of oracles written in the syntax accepted by several algorithms.
2. the `quipper` folder includes preprocessing scripts that were taken from the Quipper project to allow for the compilation.
3. the `src` folder contains the code. It has several subfolders
    - the `apps` subfolder has a folder per binary and includes the main program,
    - the `lib` subfolder contains files common to both binaries, defining the algorithms (in the `Algorithms` subfolder), as well as several auxiliary files.
    - the `test` subfolder includes the testing suite code.
4. the `docs` folder includes the generated documentation.
5. the `package.yaml` and `stack.yaml` files define the dependencies and compilation steps.

Acknowledgements

Professor Serafín Moral Callejón has helped me greatly with his corrections and advice in writing this document, allowing me to clarify my thoughts when writing this document, and teaching me how to write clearly in a mathematical style. I am also thankful to him and more generally to the Department of Computer Science and Artificial Intelligence at the University of Granada for allowing me to pursue this project with a collaboration grant.

Professor Peter Selinger has assisted me in the bundling of Quipper as a `stack` project by updating his maintained packages and answering my doubts via e-mail. I am also grateful for his work and others developing Quipper, which has allowed me to better understand quantum computers.

I would also like to thank the LibreIM organizers and attendants, for attending to my seminar on the topic and providing useful feedback. Thank you to Antonio, Mario, David, Guillermo, Sofía, José Alberto, Dani, José Manuel, Blanca and Violeta.

Lastly, a special thank you to Mario Román for his help in understanding the categorical perspective on quantum circuits and to José María Martín for his advice on the design and style of this document.

Bibliography

The bibliographical sources used for each section and chapter are quoted within the text. Several sources were used, depending on the chapter:

- Watrous (2009), Nielsen and Chuang (2010) and Arora and Barak (2009) were the main sources for the chapters relating to the basic quantum physics framework definition and the definitions and relations among the different complexity classes.
- Nielsen and Chuang (2010) and Kaye, Laflamme, and Mosca (2007) were the main sources for the chapters related to the concrete implementation and discussion of quantum algorithms.
- Lastly, Green et al. (2013a), Rios and Selinger (2018) and Green et al. (2013b) were the main sources used for the understanding and development of Quipper.

Aaronson, Scott. 2010. "BQP Vs. QMA." *Theoretical Computer Science Stack Exchange*. https://cstheory.stackexchange.com/questions/3304/bqp-vs-qma.

———. 2016a. "P $\stackrel{?}{=}$ NP." In *Open Problems in Mathematics*, 1–122. Springer, Cham. https://doi.org/10.1007/978-3-319-32162-2_1.

———. 2016b. "Lecture Notes for the 28th McGill Invitational Workshop on Computational Complexity."

Agrawal, Manindra, Neeraj Kayal, and Nitin Saxena. 2004. "PRIMES Is in P." *Annals of Mathematics* 160 (2):781–93. https://www.jstor.org/stable/3597229.

Ambainis, Andris. 2017. "Understanding Quantum Algorithms via Query Complexity." *arXiv:1712.06349 [Quant-Ph]*, December. http://arxiv.org/abs/1712.06349.

Arora, Sanjeev, and Boaz Barak. 2009. *Computational Complexity A Modern Approach*. Cambridge University Press.

Bernardy, Jean-Philippe, Mathieu Boespflug, Ryan R. Newton, Simon Peyton Jones, and Arnaud Spiwack. 2017. "Linear Haskell: Practical Linearity in a Higher-Order Polymorphic Language." *Proc. ACM Program. Lang.* 2 (POPL):5:1–5:29. https://doi.org/10.1145/3158093.

Cleve, Richard, Artur Ekert, Chiara Macchiavello, and Michele Mosca. 1998. "Quantum Algorithms Revisited." *Proceedings of the Royal Society of London. Series A: Mathematical, Physical and Engineering Sciences* 454 (1969):339–54. https://doi.org/10.1098/rspa.1998.0164.

Cobham, Alan. 1965. "The Intrinsic Computational Difficulty of Functions." In *Logic, Methodology and Philosophy of Science: Proceedings of the 1964 International Congress (Studies in Logic and the Foundations of Mathematics)*, edited by Yehoshua Bar-Hillel, 24–30. North-Holland Publishing.

Dawson, Christopher M, and Michael A Nielsen. 2005. "The Solovay-Kitaev Algorithm." *arXiv Preprint Quant-Ph/0505030*.

Faddeev, L. D. 2009. *Lectures on Quantum Mechanics for Mathematics Students*. English ed. Student Mathematical Library, v. 47. Providence, R.I: American Mathematical Society.

Feynman, Richard P. 1982. "Simulating Physics with Computers." *International Journal of Theoretical Physics* 21 (6):467–88.

Fortnow, Lance. 1997. "Complexity Theory Retrospective II." In, edited by Lane A. Hemaspaandra and Alan L. Selman, 81–107. Berlin, Heidelberg: Springer-Verlag. http://dl.acm.org/citation.cfm?id=284842.284846.

Gay, Simon J. 2006. "Quantum Programming Languages: Survey and Bibliography." *Mathematical Structures in Computer Science* 16 (4):581–600.

Green, Alexander S., Peter LeFanu Lumsdaine, Neil J. Ross, Peter Selinger, and Benoît Valiron. 2013a. "An Introduction to Quantum Programming in Quipper." *arXiv:1304.5485 [Quant-Ph]* 7948:110–24. https://doi.org/10.1007/978-3-642-38986-3_10.

———. 2013b. "Quipper: A Scalable Quantum Programming Language." *ACM SIGPLAN Notices* 48 (6):333. https://doi.org/10.1145/2499370.2462177.

Grover, Lov K. 1996. "A Fast Quantum Mechanical Algorithm for Database Search." *arXiv:Quant-Ph/9605043*, May. http://arxiv.org/abs/quant-ph/9605043.

Hutton, Graham. 2016. *Programming in Haskell*. Cambridge University Press.

Jordan, Stephen. 2011. "Quantum Algorithm Zoo." https://math.nist.gov/quantum/zoo//#BQP.

Katz, Jonathan. 2005. "Notes on Complexity Theory." http://www.cs.umd.edu/~jkatz/complexity/f05/lecture7.pdf.

Kaye, Phillip, Raymond Laflamme, and Michele Mosca. 2007. *An Introduction to Quantum Computing*. Oxford: Oxford Univ. Press.

Knill, E. 1996. "Conventions for Quantum Pseudocode." LA-UR-96-2724. Los Alamos National Lab., NM (United States). https://doi.org/10.2172/366453.

Lau, Dietlinde. 2006. *Function Algebras on Finite Sets: Basic Course on Many-Valued Logic and Clone Theory*. Springer Monographs in Mathematics. Berlin Heidelberg: Springer-Verlag. https://www.springer.com/la/book/9783540360223.

Lenstra, Arjen K., Hendrik W. Lenstra Jr, Mark S. Manasse, and John M. Pollard. 1990. "The Number Field Sieve." In *Proceedings of the Twenty-Second Annual ACM Symposium on Theory of Computing*, 564–72. ACM.

Lipton, Richard J., and Kenneth W. Regan. 2014. *Quantum Algorithms via Linear Algebra: A Primer*. Cambridge, Massachusetts: The MIT Press.

Marlow, Simon. 2010. "Haskell 2010 Language Report," 329. https://www.haskell.org/definition/haskell2010.pdf.

Nielsen, Michael A., and Isaac L. Chuang. 2010. *Quantum Computation and Quantum Information*. 10th Anniversary Edition. Cambridge University Press.

Pomerance, Carl, John L. Selfridge, and Samuel S. Wagstaff. 1980. "The Pseudoprimes to $25 \cdot 10^9$." *Mathematics of Computation* 35 (151):1003–26.

Rao, K. Ramamohan, D. N. Kim, and J. J. Hwang. 2010. *Fast Fourier Transform: Algorithms and Applications*. Signals and Communication Technology. Dordrecht ; New York: Springer.

Raz, Ran, and Avishay Tal. 2018. "Oracle Separation of BQP and PH." 107. https://eccc.weizmann.ac.il/report/2018/107/.

Rios, Francisco, and Peter Selinger. 2018. "A Categorical Model for a Quantum Circuit Description Language (Extended Abstract)." *Electronic Proceedings in Theoretical Computer Science* 266 (February):164–78. https://doi.org/10.4204/EPTCS.266.11.

Ross, Neil J. 2015. "Algebraic and Logical Methods in Quantum Computation." *arXiv:1510.02198 [Quant-Ph]*, October. http://arxiv.org/abs/1510.02198.

Saxena, Nitin. 2014. "Progress on Polynomial Identity Testing," no. 99:31.

Selinger, Peter. 2010. "A Survey of Graphical Languages for Monoidal Categories." *arXiv:0908.3347 [Math]* 813:289–355. https://doi.org/10.1007/978-3-642-12821-9_4.

Shor, P. W. 1994. "Algorithms for Quantum Computation: Discrete Logarithms and Factoring." In *Proceedings 35th Annual Symposium on Foundations of Computer Science*, 124–34. https://doi.org/10.1109/SFCS.1994.365700.

"The Glasgow Haskell Compiler." 2019. https://www.haskell.org/ghc/.

"The Haskell Tool Stack." 2019. https://docs.haskellstack.org/en/stable/README/.

van Leeuwen. 1990. *Handbook of Theoretical Computer Science*. Vol. A. The MIT Press.

Vidick, Thomas, and John Watrous. 2016. "Quantum Proofs." *Foundations and Trends in Theoretical Computer Science* 11 (1-2):1–215. https://doi.org/10.1561/0400000068.

Vollmer, Heribert. 1999. *Introduction to Circuit Complexity: A Uniform Approach*. Texts in Theoretical Computer Science. Berlin ; New York: Springer.

Watrous, John. 2009. "Quantum Computational Complexity." *Encyclopedia of Complexity and Systems Science*, 7174–7201. https://doi.org/10.1007/978-0-387-30440-3_428.

Wootters, W. K., and W. H. Zurek. 1982. "A Single Quantum Cannot Be Cloned." *Nature* 299 (5886):802. https://doi.org/10.1038/299802a0.

Ying, Mingsheng. 2016. *Foundations of Quantum Programming*. Amsterdam: Morgan Kaufmann is an imprint of Elsevier.