OPERADORES DE MUTACIÓN CUÁNTICA PARA QISKIT

Trabajo de Fin de Grado, 2022

GRADO EN MATEMÁTICAS, FACULTAD DE CIENCIAS MATEMÁTICAS, UNIVERSIDAD COMPLUTENSE DE MADRID

Jaime De la Vega Fornié

Dirigido por:
Luis Fernando Llana Díaz

Agradecimientos

A Tuty, por todo.
Adrian Barcelo Polo, por tus guías fundamentales durante estos cuatro años.
Martin Avendaño, por su comprensión y disposición para ayudar.
Luis Llana, por aguantar mis ritmos de trabajo tan aleatorios y saber reconducir mis ganas de hacer un TFG de mil páginas y abarcar mucho más de lo que puedo.
Natalia López, por guiarme y ayudarme en momentos de oscuridad.
Irene Sánchez, sin ti y las Ayudas Concepción Arenal no presento este trabajo.

Resumen

La computación cuántica es una de las disciplinas que están a la orden del día y en la vanguardia del progreso. Con su inicio en el Siglo XX, la aparición de las primeras computadoras cuánticas, aporta un rayo de esperanza al desarrollo de la computación cuántica. Hoy en día, el software y los algoritmos cuánticos están muy por delante del desarrollo de hardware, sin embargo, eso nunca ha sido un problema para que los teóricos puedan desarrollar su trabajo. En programación cuántica, el desarrollo de programas es complejo ya que se realiza con circuitos y, además, muchas veces es muy poco intuitivo. Es por esto que se requieren cada vez más herramientas de testing que nos faciliten encontrar errores. Una de las herramientas de testing más útiles es el testing con mutantes, introduciendo un mutante en el código y sometiéndolo a métodos para ver si somos capaces de detectar el mutante. En esta línea trabajan, por ejemplo, el metamorphic testing y el mutant testing. En este trabajo el objetivo será iniciar al lector en la computación cuántica, partiendo de ciertas nociones matemáticas y los postulados de la mecánica cuántica de una manera muy sencilla, para más tarde aplicarlos en sistemas cuánticos de qubits.
Después, se profundizará en la computación cuántica con los circuitos y algoritmos dando un entendimiento de cuál es su funcionamiento. Una vez se de una introducción al testing y al testing con mutantes, se culminará con la creación de clases y funciones en Python usando el lenguage de programación cuántico Qiskit para poder generar mutantes cuánticos para cualquier circuito. Este código creado dará lugar a una manera sistemática de crear mutantes para su uso en testing cuántico con Qiskit.

Palabras clave

Computación Cuántica, Qiskit, Testing Cuántico, Metamorphic Testing, Mutant Testing, Mutantes cuánticos, Algoritmos cuánticos

Abstract

Quantum computing is cutting edge technology and at the forefront of progress. With its inception in the 20th century, the appearance of the first quantum computers provides a glimmer of hope for the development of quantum computing. Today, quantum software and algorithms are far ahead of hardware development, yet this has never been a problem for theorists to develop their work. In quantum programming, the development of programs is complex because it is done with circuits and, in addition, it is often very unintuitive. This is why more and more testing tools are required to make it easier to find errors. One of the most useful testing tools is testing with mutants, introducing a mutant in the code and subjecting it to methods to see if we are able to detect the mutant. In this line work, for example, metamorphic testing and mutant testing. In this work the objective will be to introduce the reader to quantum computation, starting from certain mathematical notions and the postulates of quantum mechanics in a very simple way, to later apply them in quantum qubit systems. Afterwards, quantum computation will be deepened with circuits and algorithms, giving an understanding of how they work. Once an introduction to testing and mutant testing is given, it will culminate with the creation of classes and functions in Python for the quantum programming language Qiskit to be able to generate quantum mutants for any circuit. This created code will lead to a systematic way to create mutants for use in quantum testing with Qiskit.

Keywords

Quantum Computing, Qiskit, Quantum Testing, Metamorphic Testing, Mutant testing, Quantum Mutants, Quantum Algorithms

Contents

Chapter 1

Introducción

1.1 Contexto histórico

En el inicio del Siglo XX la física clásica se revolucionó con la aparición de la física cuántica. Con las teorías de la física clásica dando lugar a resultados absurdos como los electrones decayendo en espiral al núcleo, estaba claro que era necesario el desarrollo de una nueva teoría. La formulación de la mecánica cuántica surgió de la mano de Schrödinger y Heisenberg usando mecánica de ondas y de matrices, siendo probadas equivalentes más tarde. La fisica cuántica tenía una particularidad que sigue manteniendo hoy en día, es la física de lo pequeño, y sus mediciones y modelos basados en la probabilidad siguen siendo lo vigente hoy en día para cualquier sistema microscópico.
Asimismo, en el Siglo XX también se inició el desarrollo de la computación, con la llegada de las revolucionarias ideas de Alan Turing sobre un ordenador programable, o máquina de Turing, demostró que cualquier cálculo del mundo real puede traducirse en un cálculo equivalente que implique una máquina de Turing, lo que se conoce como la tesis de Church-Turing.

Muy pronto se fabricaron los primeros ordenadores y el hardware informático creció a un ritmo tan rápido que el crecimiento, a través de la experiencia en la producción, se codificó en una relación empírica llamada ley de Moore. Esta "ley" es una tendencia proyectiva que establece que el número de transistores en un circuito integrado se duplica cada dos años. A medida que los transistores empezaron a ser cada vez más pequeños para poder empaquetar más potencia por superficie, los efectos cuánticos empezaron a aparecer en la electrónica dando lugar a interferencias involuntarias. Esto condujo a la llegada de la informática cuántica, que utilizaba la mecánica cuántica para diseñar algoritmos.
En ese momento, los ordenadores cuánticos prometían ser mucho más rápidos que los clásicos para ciertos problemas específicos. Uno de estos problemas fue desarrollado por David Deutsch y Richard Jozsa, conocido como el algoritmo Deutsch-Jozsa. Sin embargo, este problema tenía poca o ninguna aplicación práctica. Este notable primer paso dado por Deutsch fue mejorado en la década siguiente culminando con la demostración de Peter Shor en 1994 de que dos problemas enormemente importantes: el problema de encontrar los factores primos de un número entero, y el de Perspectivas globales 7 llamado problema del "logaritmo discreto" - podían resolverse eficazmente en un ordenador cuántico.
Esto suscitó un gran interés porque estos dos problemas se creían, y se siguen creyendo, que no tienen una solución eficiente en un ordenador cuántico. se creía que no tenían una solución eficiente en un ordenador clásico. Los resultados de Shor son un poderoso indicio de que los ordenadores cuánticos son más potentes que las máquinas de Turing, incluso máquinas de Turing probabilísticas. Otra prueba de la potencia de los ordenadores cuánticos en 1995, cuando Lov Grover demostró que otro problema importante, el de de realizar una búsqueda en un espacio no estructurado, también podía acelerarse en en un ordenador cuántico. Aunque el algoritmo de Grover no proporcionó una aceleración tan espectacular como los algoritmos de Shor, la amplia aplicabilidad de las metodologías basadas en la búsqueda ha ha despertado un gran interés en el algoritmo de Grover. Más o menos al mismo tiempo que se descubrieron los algoritmos de Shor y Grover, mucha gente estaba desarrollando una idea que Richard Feynman había sugerido en 1982.

Feynman había señalado que parecía haber dificultades esenciales para simular sistemas de mecánica cuántica en ordenadores clásicos, y sugirió que la construcción de ordenadores basados en los principios de la mecánica cuántica nos permitiría evitar esas dificultades. En los década de 1990, varios equipos de investigadores empezaron a desarrollar esta idea, demostrando que, efectivamente, es que es posible utilizar los ordenadores cuánticos para simular eficientemente sistemas que no tienen simulación eficiente en un ordenador clásico. Es probable que una de las principales aplicaciones de los ordenadores cuánticos en el futuro será realizar simulaciones de sistemas mecánicos cuánticos demasiado difíciles de simular en un ordenador clásico, un problema con profundas implicaciones científicas y tecnológicas. ¿Qué otros problemas pueden resolver los ordenadores cuánticos más rápidamente que los clásicos? La respuesta corta es que no lo sabemos. Parece que es difícil encontrar buenos algoritmos cuánticos. Un pesimista podría pensar que eso se debe a que no hay nada que los ordenadores cuánticos que no sean las aplicaciones ya descubiertas. Nuestra opinión es diferente. El diseño de algoritmos para ordenadores cuánticos es difícil porque los diseñadores se enfrentan a dos problemas difíciles a los que no se enfrentan en la construcción de algoritmos para ordenadores clásicos. En primer lugar, nuestra intuición humana está arraigada en el mundo clásico. Si utilizamos esa intuición como Si utilizamos esa intuición como ayuda para la construcción de algoritmos, las ideas algorítmicas que se nos ocurran serán serán ideas clásicas. Para diseñar buenos algoritmos cuánticos hay que "apagar" la intuición clásica intuición clásica durante al menos una parte del proceso de diseño, utilizando efectos verdaderamente cuánticos para lograr el fin algorítmico deseado. En segundo lugar, para ser realmente interesante no basta con diseñar un algoritmo que sea meramente mecánico cuántico. El algoritmo debe ser mejor que cualquier algoritmo clásico existente. Así, es posible que se encuentre un algoritmo que que haga uso de los aspectos verdaderamente cuánticos de la mecánica cuántica, que sin embargo no sea de interés generalizado porque existen algoritmos clásicos con características de rendimiento comparables. La combinación de estos dos problemas hace que la construcción de nuevos algoritmos cuánticos sea un problema desafiante para el futuro. De forma aún más amplia, podemos preguntarnos si hay alguna generalización que podamos hacer sobre la potencia de los ordenadores cuánticos frente a los clásicos. ¿Qué es lo que hace que los ordenadores cuánticos que los ordenadores clásicos -suponiendo que esto sea así-? caso? ¿Qué clase de problemas pueden resolverse eficazmente en un ordenador cuántico y cómo comparación con la clase de problemas que pueden resolverse eficazmente en un ordenador clásico? clásico? Una de las cosas más interesantes de la computación y la información cuánticas es lo poco que se sabe sobre el es lo poco que se sabe sobre las respuestas a estas preguntas. Es un gran reto para el futuro comprender mejor estas cuestiones.
Los ordenadores cuánticos hoy en día son una realidad, pero aún lejana de lo que a uno le gustaría que fueran. Consisten la equivalencia al ordenador clásico ENIAC en 1945, una de las primeras máquinas de propósito general de la historia, de dimensiones exorbitadas y utilidad limitada. Por supuesto, a mediados del siglo XX todavía quedaba mucho por hacer para desarrollar los ordenadores que tenemos hoy, pero se trataba más de refinar una tecnología con la que ya nos sentíamos cómodos coqueteando que de resolver grandes desafíos.

La computación cuántica, sin embargo, aún plantea retos enormes. Titánicos, en realidad. De hecho, son tan desafiantes que algunos científicos, como el matemático israelí, y profesor en la Universidad de Yale, Gil Kalai, defienden que nunca tendremos ordenadores cuánticos completamente funcionales y con capacidad de enmendar sus propios errores. Y es que este es, precisamente, uno de los mayores desafíos en los que están trabajando los investigadores en computación cuántica: la corrección de errores.

1.2 Motivación

Aunque el estado de los ordenadores cuánticos hoy en día no es avanzado, una de las principales habilidades del matemático es resolver problemas, y más interesante aún si ni siquiera existen. El uso de ciertos programas nos permite simular la ejecución de programas cuánticos sin preocuparnos de los errores que nos provocaría un ordenador real. Por esto mismo, hoy en día podemos desarrollar software cuántico que está por delante del hardware, pero el desarrollo de software lleva ciertos problemas asociados. De aquí surgen herramientas como el testing, verificación de programas, etc. En la intesección entre los métodos formales y la computación cuántica aparece la necesidad de un marco de testing para comprobar si los programas cuánticos son correctos a través de un testing correcto, y esta será la motivación principal del

trabajo a desarrollar.

1.3 Objetivos

El primero y más importante será la introducción a la computación cuántica, la familiarización con los conceptos básicos, estando apoyado el lector por una base matemática robusta no es difícil pero si extenso. Esto se iría compaginando con el aprendizaje de Qiskit, el cual es un módulo de Python que permite escribir circuitos, que es como se llaman los programas cuánticos. También permite su ejecución en computadoras cuánticas reales, lo cual está aún muy limitada como ya se ha mencionado en la introducción. Uno de los objetivos del trabajo es desarrollar el aprendizaje de qiskit en paralelo al aprendizaje de la computación cuántica, de una manera casi simbiótica. Por esto mismo, una de las herramientas fundamentales de este trabajo es el cuaderno de Jupyter Notebook con el que se han realizado todas las figuras incluidas en el trabajo, así como los circuitos. El lector lo puede encontrar en `https://github.com/JaimeDelaVegaFornie/TFG-2022.git` y también en el anexo del trabajo, de hecho se recomienda ir comprobando en paralelo y probando los distintos programas y cirucuitos del trabajo para tener una introducción. Pues como decía aristóteles, *«Lo que tenemos que aprender a hacer, lo aprendemos haciendo»*..
Una vez aprendido eso, se llega al objetivo específico del trabajo. Para poder ejecutar los programas, que se representan en una notación interna de quiskit, hay que investigar esa notación interna para luego poder hacer modificaciones en ella y poder crear mutantes con el objetivo de testear los programas.

Chapter 2

Repaso de matemáticas y física

Es exageradamente pretencioso tratar de hacer una introducción completa a la mecánica cuántica en una sección de un trabajo de fin de grado, por lo tanto, lo que se pretende aquí es tratar conceptos básicos que puedan aportar cierto entendimiento de las bases utilizadas en la mecánica cuántica para ganar intuición y entendimiento. El lector puede saltar esta sección y avanzar directamente l capítulo 3 y volver cuando necesite saber algo del fundamento matemático o físico de las secciones posteriores. A continuación haremos un repaso rápido de álgebra lineal y espacios de Hilbert y, un vez que tenemos esto, se intentará realizar una breve y humilde introducción a la mecánica cuántica.

2.1 Álgebra lineal

Se asume que el lector está familiarizado con algebra lineal básica propia de la enseñanza matemática, a partir de ahí, se intentará desarrollar aquello que sea clave en el entendimiento de la mecánica cuántica.

2.1.1 Notación de Dirac

La notación usual utilizada en la mecánica cuántica es la notación Bra-ket, que se utiliza para denotar estados cuánticos, utiliza "angle brakets" *deberiahaberunket* y una barra vertical | para construir "bras" y "kets"
Un ket es de la forma $|v\rangle$. Matemáticamente denota un vector v en un espacio vectorial abstracto, que en nuestro caso será sobre el cuerpo de los complejos. Fisicamente representa el estado de un sistema cuántico.
Un bra es de la forma $\langle f|$. Matematicamente denota una forma lineal $f : V \to \mathbb{C}$. Básicamente una aplicación lineal que manda cada vector de V a un número en el plano complejo $\mathbb{C}$. Más notación de física que se utilizará:

Notación	Descripción
z^*	Conjugación compleja de z, $(1+i)^* = 1-i$
$\langle\psi\| \, \|\phi\rangle$	Producto escalar de ψ y ϕ
$\|\psi\rangle \, \|\phi\rangle$	Producto tensorial de ψ y ϕ ($\|\psi\rangle \otimes \|\phi\rangle$)
$\langle\psi\| \, A \, \|\phi\rangle$	Producto interior de ψ y Aϕ

Como se ha mencionado anteriormente, se asumen conceptos básicos y propios de un curso introductorio de álgebra lineal tales como bases, autovectores o matrices adjuntas y se prodece a extender aquellos que pudieran resultar ligeramente más avanzados.

2.1.2 Definiciones y resultados básicas

Como recordatorio, veamos lo que es un operador lineal.

Definition 2.1.1 (Aplicación lineal) *Sean $\mathbb{V}$ y $\mathbb{V}'$ espacios vectoriales complejos. Una aplicación lineal es una función $f : \mathbb{V} \to \mathbb{V}'$ que verifica que $\forall\, \Phi, \Psi \in \mathbb{V}$ y $c \in \mathbb{C}$:*
(i) $f(\Phi + \Psi) = f(\Phi) + f(\Psi)$
(ii)$f(c\Phi) = cf(\Phi)$

Practicamente todas las funciones que vamos a considerar serán aplicaciones lineales. Las matrices, cuando actúan es un espacio vectorial complejo son aplicaciones lineales.

Definition 2.1.2 (Operador) *Un* operador *es una aplicación lineal $f : \mathbb{C} \to \mathbb{C}$ de un espacio complejo en si mismo, es decir, un endomorfismo.*

Definition 2.1.3 (Espacio de Hilbert) *Un espacio de Hilbert H es un espacio vectorial real o complejo equipado con un producto interior que también es un espacio métrico completo con la función de distancia definida a partir del producto interior.*

En todo momento vamos a trabajar en espacios vectoriales complejos de dimensión finita donde un espacio de Hilbert es exactamente lo mismo que un espacio vectorial equipado con producto interior. Por lo que a partir de ahora utilizaremos el término espacio de Hilbert para referirnos a ello. En dimensiones infinitas no es equivalente, pero no vamos a trabajar en espacios de dimensión infinita.

Definition 2.1.4 (Conjugado Hermitico) *Sea A un operador lineal en un espacio de Hilbert V. existe un único operador lineal $A^\dagger \in V$ tal que, para cualesquiera vectores $|v\rangle, |w\rangle \in V$ tenemos que*

$$\langle v| A |w\rangle = \langle w| A^\dagger |v\rangle \tag{2.1}$$

Este operador lineal $A^\dagger$ se conoce como el conjugado hermitico de A.

Definition 2.1.5 (Operador normal) *Un operador A se dice normal si $AA^\dagger = A^\dagger A$*

Theorem 2.1.1 (Descomposión espectral) *Todo operador normal M en un espacio vectorial V es diagonal con respecto cierta base ortonormal para V. De igual manera, todo operador diagonalizable es normal.*

2.1.3 Producto tensorial

En esta sección seguiré añadiendo cosas según las vaya necesitando en la sección de mecánica cuántica (PENDIENTE)

2.2 Mecánica cuántica

A continuación describiremos los postulados básicos que dieron lugar a la mecánica cuántica atendiendo a los desarrollados por [1] y [2] pero simplificándolos para ajustarnos a un entendimiento básico y no dispersarnos del objetivo prinpipal. Estos postulados dan lugar a la conexión entre el formalismo matemático de la mecánica cuántica y el mundo físico.
Los postulados y su motivación pueden parecer verdaderamente sorprendentes, tanto para el lector no familiarizado como para el experto en la materia, por lo que no hay que alarmarse en el caso de encontrar intrincadas las siguientes páginas. Lo fundamental será llegar a construir un concepto de lo que constituyen estos postulados y cómo aplicarlos.
Podría llegar a ser de utilidad saltar esta sección en una primera lectura y volver para entender mejor los sistemas cuánticos una vez que uno esté motivado por su aplicación en la computación cuántica.

Postulado 1 (Espacio de estados) *A cualquier sistema físico aislado se le asocia un espacio vectorial complejo con producto interior (es decir, un espacio de Hilbert) conocido como el espacio de estado del sistema. El sistema está completamente descrito por su vector de estado, que es un vector unitario en el espacio de estado del sistema.*

El sistema cuántico más simple que existe es un qubit, para el cuál ocuparemos nuestro tiempo en los siguientes capítlos y es la pieza básica de la computación cuántica.

Postulado 2 (Evolución) *La evolución de un sistema cuántico cerrado viene descrta por una transformación unitaria. Es decir,el estado $|\psi\rangle$ de un sistema en tiempo t_1 está relacionado con el estado $|\psi'\rangle$ del sistema en tiempo t_2 mediante un operador unitario U que depende exclusivamente de t_1 y t_2,*

$$|\psi'\rangle = U|\psi\rangle \tag{2.2}$$

Estos operadores serán los que más adelante llamaremos puertas cuánticas, que provocarán la evolución de los estados de nuestros sistemas cuánticos.
Notese que el postulado sólo nos habla de como evoluciona respecto de dos tiempos unicamente, en la versión continua de este postulado aparece la famosa ecuación de Schrödinger, lo que da lugar a una reformulación del postulado para considerar el tiempo continuo:

Postulado 3 (Evolución continua) *La evolución en el tiempo de un estado en un sistema cuántico cerrado viene dada por la ecuación de Schrödinger,*

$$i\hbar\frac{d|\psi\rangle}{dt} = H|\psi\rangle \tag{2.3}$$

Donde $\hbar$ es la constante de Plank, cuyo valor se determina experimentalmente.
Y H es un operador hermitico fijo conocido como el Hamiltoniano del sistema cerrado.

Postulado 4 (Medición cuántica) *Ls mediciones cuánticas vienen descritas por una colección $\{M_m\}$ de operadores de medición. Estos son operadores que actuan en el espacio de estados del sistema a medir, donde el índice m se refiere a los distintos resultados que puedan surgir de las mediciones en el experimento. Si el estado de un sistema es $|\psi\rangle$ inmediatamente antes de la medición entonces la pronanilidad de que ocurra el resultado m viene dad por:*

$$p(m) = bra\psi M_m^\dagger M_m \langle\psi| \tag{2.4}$$

y el estado del sistema después de la medición es

$$\frac{M_m|\psi\rangle}{\sqrt{\psi|M_m^\dagger M_m|\psi}} \tag{2.5}$$

Y los operadores de medición verifican la ecuación de completitud

$$\sum_m M_m^\dagger M_m = I \tag{2.6}$$

La ecuación de completitud expresa el hecho de que las probabilidades suman 1:

$$1 = \sum_m p(m) = \sum_m braket\psi|M_m^\dagger M_m|\psi \tag{2.7}$$

Postulado 5 (Sistemas compuestos) *El espacio de estados de la composición de sistemas físicos es el producto tensorial de los espacios de estados que lo componen. Es decir, si tenemos una colección de estados $\{\psi_i\}_{i=1}^n$, el estado conjunto del sistema es $|\phi_1\rangle \otimes |\phi_2\rangle \otimes ... |\phi_n\rangle$*

¿Por qué el producto tensorial es la estructura matemática utilizada para describir el espacio de estados de un sistema físico compuesto? En un nivel, podemos simplemente aceptarlo como un postulado básico, no reducible a algo más elemental, y seguir adelante. Después de todo, ciertamente esperamos que haya alguna forma canónica de describir los sistemas compuestos en la mecánica cuántica. ¿Hay alguna otra forma de llegar a este postulado? He aquí una heurística que se utiliza a veces. A los físicos a veces les gusta hablar del principio de superposición de la mecánica cuántica, que dice que si $|x\rangle$ y $|y\rangle$ son dos estados de un sistema cuántico, entonces cualquier superposición $\alpha|x\rangle + \beta|y\rangle$ y debe ser también un estado permitido de un sistema cuántico, donde $|\alpha|^2 + |\beta|^2 = 1$. Para los sistemas compuestos, parece natural que si $|A\rangle$ es un estado del sistema A, y $|B\rangle$ es un estado del sistema B, entonces debería haber algún estado correspondiente que podríamos denominar $|A\rangle|B\rangle$, del sistema conjunto AB. Aplicando el principio de

superposición a los estados del producto de esta forma, llegamos al postulado del producto tensorial dado arriba [3]. Esto no es una derivación, ya que no estamos tomando el principio de superposición como una parte fundamental de nuestra descripción de la mecánica cuántica, pero da una idea de la posible aparición de este postulado.

Chapter 3

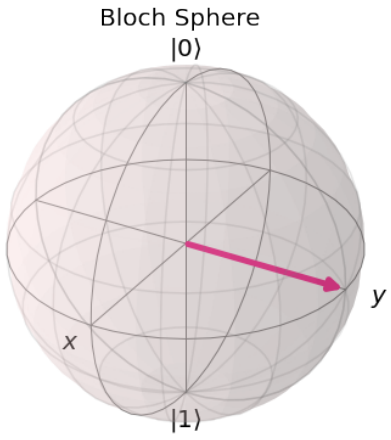
Computación cuántica

3.1 Del bit clásico al qubit cuántico

La base de la computación actual son los bits, es la unidad básica de información. El nombre es un acrónimo de binary digit y representa un estado lógico, normalmente "0" o "1". La computación se basa en cadenas de bits que codifican todo el almacenamiento, la corriente y también de manera fundamental, los circuitos del hardware de los ordenadores que se usan a diario.
La computación cuántica y la información cuántica se basa en un concepto análogo, una generalización del bit que es la base de la potencia de este paradigma de computación.
Matemáticamente podemos considerar un bit como un estado representado por digito binario, de esta manera, se puede trabajar matemáticamente con él. Un bit procesado se implementa mediante uno de los dos niveles de baja tensión continua, y mientras se pasa de uno de estos dos niveles al otro, debe superarse lo más rápidamente posible la llamada "zona prohibida" entre dos niveles lógicos, ya que la tensión eléctrica no puede cambiar de un nivel a otro instantáneamente. Por lo tanto, consideramos que un bit tiene sólo dos estados posibles, que ahora representaremos como $|0\rangle$ y $|1\rangle$
Un qubit es un sistema cuántico que tiene un espacio bidimensional de estados. Si suponemos que $|0\rangle$ y $|1\rangle$ forman una base ortonormal para ese espacio de estados, la medición de un qubit siempre nos devolverá uno de los dos estados, sin embargo, el qubit puede estar en un estado distinto a $|0\rangle$ y $|1\rangle$, puede ser una combinación lineal de estados, lo que se suele llamar *superposición*:

$$|\psi\rangle = \alpha\,|0\rangle + \beta\,|1\rangle \tag{3.1}$$

Donde α y β son números complejos, aunque para muchos propósitos no se pierde mucho en ellos como números reales. Dicho de otro modo, el estado de un qubit es un vector en un espacio vectorial complejo de dos dimensiones. Los estados especiales $|0\rangle$ y $|1\rangle$ se conocen como estados computacionales básicos y forman una base ortonormal para este espacio vectorial. Podemos examinar un bit para determinar si está en el estado 0 o 1, los ordenadores hacen esto todo el tiempo, por ejemplo cuando recuperan el contenido de su memoria.No obstante, no podemos examinar un qubit para determinar su estado cuántico, es decir, los valores de α y β. En su lugar, la mecánica cuántica nos dice que sólo podemos adquirir información mucho más restringida sobre el estado cuántico. Cuando medimos un qubit obtenemos o bien el resultado $|0\rangle$, con probabilidad $|\alpha|$ o el resultado $|1\rangle$, con probabilidad $|\beta|$. Naturalmente, $|\alpha|^2 + |\beta|^2 = 1$, ya que las probabilidades deben sumar uno. La capacidad de un qubit de estar en un estado de superposición va en contra de nuestro "sentido



común del mundo físico que nos rodea. Un bit clásico es como
una moneda: o cara o cruz. En el caso de las monedas imperfectas, puede haber estados intermedios, como tenerla equilibrada en un borde, pero estos pueden ser ignorados en el caso ideal. En cambio, un qubit puede existir en un continuo de estados entre $|0\rangle$ y $|1\rangle$ hasta que es observado. Insistamos en que cuando se mide un qubit, el resultado de la medición es siempre "0" o "1". Por ejemplo, un qubit puede estar en el estado

$$|+\rangle := \frac{1}{\sqrt{2}}|0\rangle + \frac{1}{\sqrt{2}}|1\rangle \tag{3.2}$$

El cuál, cuando se mide, devuelve $|0\rangle$ el 50% del tiempo y $|1\rangle$ el otro 50% [4]. A primera vista, podría parecer que debería haber cuatro grados de libertad en $|\psi\rangle$ ya que α y β son números complejos con dos grados de libertad cada uno. Sin embargo, un grado de libertad se elimina por la restricción de normalización $|\alpha|^2 + |\beta|^2 = 1$. Esto significa que, con un cambio adecuado de coordenadas, se puede eliminar uno de los grados de libertad. Por lo tanto, uno puede visualizar un qubit como un vector en una esfera de radio 1. En la imagen 3.1 podemos observar esta visualización propuesta, donde el vector constituye un estado $|\psi\rangle = cos\frac{\theta}{2}|0\rangle + e^{i\phi}sin\frac{\theta}{2}|1\rangle$ donde θ y ϕ son las coordenadas que definen un punto en la superficie de la esfera de Bloch. Al medir un bit, obligamos al vector a colapsar al estado $|0\rangle$ con probabilidad $|cos\frac{\theta}{2}|^2$ y al estado $|1\rangle$ con probabilidad $|e^{i\phi}sin\frac{\theta}{2}|^2$. En esta imagen en particular, como el vector está justo a mitad de camino entre ambos, la probabilidad es 50/50.

3.1.1 Varios qubits

Los ordenadores con un sólo bit no resultan nada interesantes. De igual manera, se busca tener ordenadores con más de un qubit.
En primer lugar, veamos como se extendería nuestro sistema para tener dos qubits. Cuando tenemos 2 bits para representar la información se obtienen cuatro posibles combinaciones 00, 10, 01 y 11.
Consideremos por ejemplo la cadena 01, para combinar sistemas cuánticos se debe usar el producto tensorial. Podemos describir el bite 01 como:

$$|0\rangle \otimes |1\rangle \in \mathbb{C}^2 \otimes \mathbb{C}^2 \tag{3.3}$$

Un par de qubits puede escribirse como $|0\rangle \otimes |1\rangle$ o $|0 \otimes 1\rangle$, como notacion, también lo escribiremos como $|0\rangle\,|1\rangle$, $|0,1\rangle$ o $|01\rangle$.
En un sistema de dos qubits habrá cuatro estados computacionales básicos denotados por $|00\rangle$, $|01\rangle$, $|10\rangle$, $|11\rangle$, por lo que un vector de estado que describa estos dos qubits será de la forma:

$$|\psi\rangle = \alpha_{00}|00\rangle + \alpha_{01}|01\rangle + \alpha_{10}|10\rangle + \alpha_{11}|11\rangle \tag{3.4}$$

De manera análoga al caso de un único qbit, el resultado de medir un estado $|x\rangle$ ocurre con probabilidad $|\alpha_x|^2$ Ahora la noción de entrelazamiento también se extiende y es aquí dónde aparecen conclusiones sorprendentes.
Veamos como representar formalmente un estado de varios qubits, si tenemos dos qubits separados, podemos describir su estado conjunto usando el producto tensorial como ya vimos en la SECCION TENSORES. Si tenemos dos qubits,

$$|a\rangle = \begin{pmatrix} a_0 \\ a_1 \end{pmatrix}, |b\rangle = \begin{pmatrix} b_0 \\ b_1 \end{pmatrix} \tag{3.5}$$

Su estado combinado es:

$$|ba\rangle = |b\rangle \otimes |a\rangle = \begin{pmatrix} b_0 \times \begin{pmatrix} a_0 \\ a1 \end{pmatrix} \\ b_1 \times \begin{pmatrix} a_0 \\ a1 \end{pmatrix} \end{pmatrix} \tag{3.6}$$

Veamos un ejemplo de esto, sea un sistema que está en el estado

$$\frac{|00\rangle + |11\rangle}{\sqrt{2}} = \frac{|00\rangle}{\sqrt{2}} + \frac{|11\rangle}{\sqrt{2}} \tag{3.7}$$

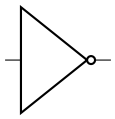
Este estado es conocido como Bell state o par EPR y es responsable de muchas sorpresas en computación cuántica. En este estado, los dos qubits están entrelazados, esto quiere decir que, si medimos el primer qubit y lo encontramos en el estado $|1\rangle$, entonces automáticamente sabemos que el estado del segundo qubit es $|1\rangle$ Como resultado, una medición del segundo qubit siempre da el mismo resultado que la medición del primer qubit. Es decir, los resultados de las mediciones están correlacionados. De hecho, resulta que se pueden realizar otros tipos de mediciones en el estado de Bell aplicando primero algunas operaciones al primer o segundo qubit, y aún siguen existiendo interesantes correlaciones interesantes entre el resultado de una medición en el primer y segundo qubit. Estas correlaciones han sido objeto de un intenso interés desde el famoso artículo de Einstein, Podolsky y Rosen, en el que señalaron por primera vez las extrañas propiedades de estados como el de Bell. Las ideas de EPR fueron retomadas y mejoradas por John Bell, que demostró un resultado sorprendente: las correlaciones de medición en el estado de Bell son más fuertes que las que podrían existir entre los sistemas clásicos.
De forma más general, podemos considerar un sistema de n qubits. Los estados base computacionales de este sistema son de la forma $|x_1 x_2 ... x_n\rangle$, por lo que un estado cuántico de dicho sistema está especificado por 2^n amplitudes. Para n = 500 este número es mayor que el número estimado de número de átomos en el Universo. Tratar de almacenar todos estos números complejos no sería sería posible en ningún ordenador clásico concebible. El espacio de Hilbert es, en efecto, un lugar muy grande. En principio, sin embargo, la Naturaleza manipula esas enormes cantidades de datos, incluso para sistemas que contienen sólo unos cientos de átomos. Es como si la Naturaleza guardara 2.500 trozos de papel papel de borrador escondido, en el que realiza sus cálculos a medida que el sistema evoluciona. Este enorme potencial de cálculo es algo que nos gustaría mucho aprovechar. Pero, ¿cómo podemos pensar en la mecánica cuántica como computación?

3.2 Puertas cuánticas

Los cambios que se producen en un estado cuántico pueden describirse utilizando el lenguaje de la computación cuántica. De forma análoga a como se construye un ordenador clásico a partir de un circuito eléctrico que contiene cables y puertas lógicas, un ordenador cuántico se construye a partir de un circuito cuántico que contiene cables y puertas cuánticas elementales para transportar y manipular la información cuántica. En esta sección, partiendo de los circuitos clásicos, describimos algunas puertas cuánticas sencillas y presentamos varios circuitos de ejemplo que ilustran su aplicación, incluido un circuito que teletransporta ¡qubits!

3.2.1 De las puertas clásicas a las puertas cuánticas

Las puertas lógicas clásicas son la manera que se utiliza para manipular bits, y la concatenación de puertas da lugar a un circuito. Uno podría preguntarse cómo podemos cambiar el estado de un sólo bit, consideremos por ejemplo la puerta lógica not:



Esta puerta toma como input un bit y devuelve un bit, podemos representar la puerta con una matriz cuadrada:

$$NOT = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \tag{3.8}$$

De esta manera, $NOT\,|0\rangle = |1\rangle$ y viceversa. Como hemos utilizado la representación de los bits a través de un vector, esto nos podría inducir de igual manera la puerta cuántica que representa a NOT, que llamaremos X. Nótese que estamos asumiendo que la puerta cuántica NOT actúa linealmente, es decir:

$$X(\alpha\,|0\rangle + \beta\,|1\rangle) = \alpha\,|1\rangle + \beta\,|0\rangle \tag{3.9}$$

O, si representamos en notación vectorial el estado $\alpha\,|0\rangle + \beta\,|1\rangle$:

$$X\begin{pmatrix} \alpha \\ \beta \end{pmatrix} = \begin{pmatrix} \beta \\ \alpha \end{pmatrix} \tag{3.10}$$

Por qué la puerta cuántica actúa linealmente y no de forma no lineal es una pregunta muy interesante, y la respuesta no es en absoluto obvia. Resulta que este comportamiento lineal es una propiedad general de la mecánica cuántica, y muy bien motivada empíricamente. Además, el comportamiento no lineal puede dar lugar a aparentes paradojas como los viajes en el tiempo, la comunicación más rápida que la luz y las violaciones de las segundas leyes de la termodinámica. En el caso de una puerta lógica clásica unitaria, la única puerta no trivial es la puerta NOT, sin embargo, cuando nos vamos a mundo cuántico, las posibilidades son mucho mayores. Primero, vamos a plantearnos que tiene que satisfacer una puerta cuántica. La condición de normalización nos obliga a que la matriz sea unitaria, es decir, que $UU^\dagger = I$ donde $U^\dagger$ es la adjunta de U resultado de transponer y despues conjugar en los complejos. Sorprendentemente, la única condición que deben satisfacer las puertas cuánticas es esta, por lo tanto, cualquier matriz unitaria puede ser una puerta cuántica. Vamos a mencionar dos de ellas con nombre propio y que serán de utilidad posteriormente.
En primer lugar vamos a considerar la puerta Z, esta puerta deja $|0\rangle$ igual y cambia el signo de $|1\rangle$:

$$Z = \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix} \tag{3.11}$$

Por otro lado, tenemos la puerta de Hadarmard, una de las más importantes y útiles.

$$H = \frac{1}{\sqrt{2}}\begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix} \tag{3.12}$$

Esta puerta es el equivalente a la "raiz cuadrada de AND" ya que convierte $|0\rangle$ en $(|0\rangle + |1\rangle)/\sqrt{2}$ y $|1\rangle$ en $(|0\rangle - |1\rangle)/\sqrt{2}$. Si lo visualizamos en la esfera de Bloch, lo que hace es rotar la esfera sobre el eje y 90º y después aplica una rotación sobre el eje x de 180º.
Igual que las puertas lógicas clásicas tienen su representación, la tienen las puertas cuánticas, veamos las funamentales:

Pauli-X	$q:$ —[H]—	$\begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}$
Pauli-Y	$q:$ —[Y]—	$\begin{pmatrix} 0 & -i \\ i & 0 \end{pmatrix}$
Pauli-Z	$q:$ —[Z]—	$\begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix}$
Hadamard	$q:$ —[H]—	$\frac{1}{\sqrt{2}}\begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix}$

Veamos más puertas interesantes para un único qubit: si tomamos la exponencial de las matrices de Pauli X,Y,Z llegamos a los operadores de rotación sobre los distintos ejes de la esfera de Bloch:

$$R_x(\theta) \equiv e^{-i\theta X/2} = cos(\frac{\theta}{2})I - i * sin(\frac{\theta}{2})X = \begin{pmatrix} cos(\frac{\theta}{2}) & -i \cdot sin(\frac{\theta}{2}) \\ -i \cdot sin(\frac{\theta}{2}) & cos(\frac{\theta}{2}) \end{pmatrix} \tag{3.13}$$

$$R_y(\theta) \equiv e^{-i\theta X/2} = cos(\frac{\theta}{2})I - i * sin(\frac{\theta}{2})Y = \begin{pmatrix} cos(\frac{\theta}{2}) & -sin(\frac{\theta}{2}) \\ sin(\frac{\theta}{2}) & cos(\frac{\theta}{2}) \end{pmatrix} \tag{3.14}$$

$$R_z(\theta) \equiv e^{-i\theta X/2} = cos(\frac{\theta}{2})I - i * sin(\frac{\theta}{2})Z = \begin{pmatrix} e^{-i\theta/2} & 0 \\ 0 & e^{i\theta/2} \end{pmatrix} \tag{3.15}$$

Veamos ahora que ocurre cuando aplicamos puertas unitarias en un sistema de varios qubits. Para empezar, supongamos que tenemos un sistema de 2 qubits, $|q_0\rangle$ y $|q_1\rangle$, el estado del sistema total es $|q_1 q_0\rangle$. Si queremos ver, por ejemplo, como actúa una puerta h sobre $|q_0\rangle$ y una puerta x sobre $|q_1\rangle$, que viene representado por el siguiente circuito:

$q_0:$ —[H]—
$q_1:$ —[X]—

Para representar el sistema final utilizamos el producto tensorial y, por sus propiedades:

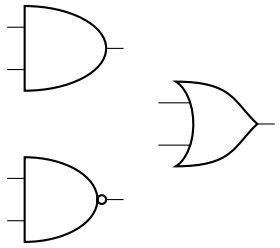
$$X|q_1\rangle \otimes H|q_0\rangle = (X \otimes H)|q_1 q_0\rangle \tag{3.16}$$

Para representar el productor tensorial de manera más clara, podemos utilizar la siguiente notación:

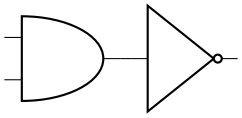
$$X \otimes H = \begin{pmatrix} 0 & H \\ H & 0 \end{pmatrix} \tag{3.17}$$

3.2.2 Puertas de varios qubits

En computación clásica, tenemos una serie de puertas lógicas que nos permiten hacer operaciones con varios bits, por ejemplo AND, OR y NAND:



Un resultado teórico fundamental es que cualquier operación sobre bits puede ser computada componiendo la puerta NAND, por lo que esta se conoce como la puerta universal. NAND es básicamente una puerta AND seguida de una puerta NAND:



En términos de matrices:

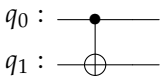
$$NAND = NOT \times AND = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{3.18}$$

Sin embargo, todas las puertas que hemos presentado para múltiples qubits tienen algo en común y es que no son reversibles. Esto nos dice que no funcionarán en el mundo cuántico ya que todas las puertas cuánticas tienen que ser reversibles y representadas por matrices unitarias. Por ejemplo, la operación AND no es reversible, ya que si recibieramos el bit $|0\rangle$ no podríamos determinar en qué estado estaban exactamente los dos bits iniciales.

Las puertas reversibles tienen una historia anterior a la computación cuántica. Nuestros ordenadores cotidianos pierden energía y generan una enorme cantidad de calor. En la década de 1960, Rolf Landauer analizó los procesos computacionales y demostró que la pérdida de energía y el calor se producen al borrar información, en lugar de escribirla. Esta noción se conoce como el principio de Landauer.

Hemos comprobado que borrar la información es una operación irreversible, que disipa la energía. En los años 70, Charles H. Bennett continuó con esta línea de pensamiento. Si borrar información es la única operación que utiliza energía, entonces un ordenador que ea reversible y no borre, no consumirá energía. Bennett comenzó a trabajar en circuitos y programas reversibles.

Para ver un ejemplo de circuito cuántico, vamos a considerar una puerta reversible que afecte a más de un qubit. Por ejemplo, podemos considerar la puerta controlled-NOT:



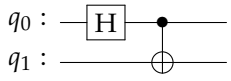
Esta puerta tiene dos inputs y dos outputs, el input de arriba, denotado por $|x\rangle$ es el qubit de control. Si $|x\rangle = |0\rangle$ entonces $|y\rangle$ será igual en la salida que en la entrada, de lo contrario, actúa como un not sobre $|y\rangle$. Es decir la puerta CNOT lleva el estado $|x, y\rangle$ a $|x, x \oplus y\rangle$ donde $\oplus$.

La matriz que corresponde a esta puerta es:

$$CNOT = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \end{pmatrix} \tag{3.19}$$

Con un poco de álgebra elemental es trivial comprobar que esta matriz es unitaria y que su inversa es ella misma.

Veamos como actúa sobre qubits en superposición. Consideremos el circuito de Bell, integrado por puertas que ya conocemos y con consecuencias muy profundas, como veremos en la sección 3.3.2.



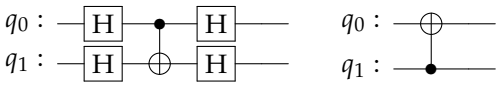
El resultado de aplicar en primer lugar la puerta H a el estado conjunto es

$$I|0\rangle \otimes H|0\rangle = \frac{1}{\sqrt{2}}(|00\rangle + |01\rangle) = |0+\rangle \tag{3.20}$$

Ahora, al aplicar CNOT:

$$CNOT|01\rangle = \frac{1}{\sqrt{2}}(|00\rangle + |11\rangle) \tag{3.21}$$

Y de esta manera llegamos al estado de Bell, que ya hemos expuesto en secciones anteriores. Si seguimos jugando con estas puertas llegamos a la siguiente igualdad, llamada "CNOT circuit identity"

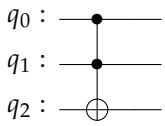


Este es un ejemplo de kickback (o phase kickback) que es muy importante y se utiliza en casi todos los algoritmos cuánticos. El kickback consiste en que el valor propio añadido por una puerta a un qubit se "devuelve" a un qubit diferente mediante una operación controlada.

Además, las operaciones controladas son una de las operaciones más útiles en computación, tanto clásica como cuántica. Podemos variar tanto la cantidad de bits de control como la puerta que se aplica en caso de que se cumplan las condiciones. Por ejemplo, para aplicar un operador unitario cualquiera U, tenemos la operación "U-controlada" (controlled-U), que también denotaremos como ${}^{C}U$. En este caso, representa la siguiente operación:

$$|c\rangle|t\rangle \rightarrow |c\rangle U^c|t\rangle$$

Por otro lado, otra puerta fundamental de puerta cuántica es la puerta de Toffoli, que es como una extensión de CNOT pero con dos qubits de control:



Fundamentalmente, la operación que realiza es:

$$|x, y, z\rangle \rightarrow |x, y, z \oplus (x \wedge y)\rangle \tag{3.22}$$

de nuevo, es un ejercicio elemental comprobar que la puerta de Toffoli es su propia inversa, basta con ponerlo en términos de matrices.
Una de las razones por las que la puerta de Toffoli es interesante es que es universal en computación clásica. En otras palabras, con copias de la puerta de Toffoli, se puede hacer cualquier puerta lógica clásica. En particular, se puede hacer un ordenador reversible utilizando sólo puertas de Toffoli. Tal ordenador en teoría, no utilizaría ninguna energía ni emitiría ningún calor.

3.2.3 Puertas universales

Como acabamos de ver, la puerta de toffoli constituye, en computación clásica, una puerta univeersal. En lo que respecta a computación cuántica, un conjunto de puertas se dice universal para computación cuántica

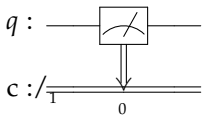
si cualquier operación unitaria se puede aproximar con precisión arbitraria por un circuito cuántico. Un conjunto universal es

$$\{H, {}^{C}NOT, R(cos^{-1}(\frac{3}{5}))\} \tag{3.23}$$

Sin embargo, la demostración de este resultado se escapa de los objetivos del trabajo. Para profundizar en las puertas universales y la demostración, se puede usar [3][4].

3.2.4 Medición

El elemento que nos falta para construir circuitos cuánticos es la medición, en un circuito, denotaremos una medida proyectiva en la base computacional utilizando el siguiente símbolo:



Donde la línea que sale de q representa un qubit y la línea doble que sale de c representa 1 bit clásico.

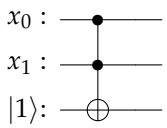
Respecto a las mediciones, hay dos principios fundamentales a tener en cuenta.
Principio de medición diferida: Las mediciones siempre se pueden trasladar desde una etapa intermedia de un circuito cuántico al final del circuito; si los resultados de las mediciones se utilizan en cualquier etapa del circuito, entonces las operaciones controladas clásicamente controladas clásicamente pueden ser sustituidas por operaciones cuánticas condicionales. **Principio de medición implícita**: Sin pérdida de generalidad, cualquier cables cuánticos no terminados (qubits que no se miden) al final de un de un circuito cuántico puede suponerse medido.

3.3 Circuitos cuánticos

Una vez vistos los elementos básicos que componen un circuito, es interesante desarrollar como combinarlos para sacarles partido, por lo que en esta sección vamos a ver algunos ejemplos que nos den una idea de cómo funciona un circuito cuántico. Para esta sección, nos basamos fundamentalmente en el trabajo realizado en materia de circuitos en [5].

3.3.1 Computación clásica en un ordenador cuántico

Si el objetivo de la computación cuántica es superar a la computación clásica, una pregunta que uno se podría hacer es si es posible simular un circuito lógico cláisco usando un circuito cuántico. La respuesta resulta se que si, lo cual es congruente con la creencia que ostentan los físicos sobre la universalidad de la mecánica cuántica. Por lo tanto, los circuitos lógicos clásicos pueden ser simulados en un ordenador cuántico. Sin embargo, ya hemos visto que esto al revés no es posible ya que las puertas como NAND, AND o XOR no son reversibles, luego no podemos usarlas directamente y, de igual manera, tampoco se pueden simular exactamente con circuitos cuánticos al ser todos los operadores uniterios y reversibles.
Como ya mencionamos en la sección 3.2.2, la puerta de Toffoli es universal en computación clásica y como ya hemos visto, también podemo aplicar la puerta de Toffoli en computación cuántica. Vamos ver esto, sabemos que la puerta NAND es universal, es decir, cualquier función booleana se puede generar con la combinación de puertas NAND [6]. Veamos que con la puerta de Toffoli podemos generar puertas lógicas clásicas equivalentes a NAND.



En este circuito, si medimos el tercer qubit, nos devolverá $NAND(x_0, x_1)$ por lo tanto, concatenando estas puertas ya somos capaces de generar cualquier circuito lógico clásico.

3.3.2 Intercambiar qubits

Una vez que hemos visto ejemplos de puertas de uno y varios qubits, sólo queda ver cómo se pueden concatenar estas puertas para formar circuitos, que es la base de la computación. Vamos a planternos la posibilidad de intercambiar qubits de sitio. Lo que buscamos es una operación lineal unitaria T que, dado un sistema cuántico $\mathbb{V}$ verifique

$$T : \mathbb{V} \otimes \mathbb{V} \to \mathbb{V} \otimes \mathbb{V} \tag{3.24}$$

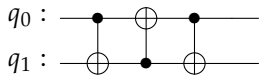
Y el objetivo es que lleve

$$T(|x\rangle \otimes 0) = (0 \otimes |x\rangle) \tag{3.25}$$

Si transportamos una superposición de estados:

$$T(\frac{|x\rangle + |y\rangle}{\sqrt{2}} \otimes 0) = \frac{1}{\sqrt{2}}(T(|x\rangle \otimes 0) + T(|y \otimes 0\rangle) = \frac{0 \otimes (|x\rangle + |y\rangle)}{\sqrt{2}} = 0 \otimes \frac{|x\rangle + |y\rangle}{\sqrt{2}} \tag{3.26}$$

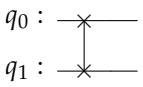
Podemos genrar un circuito que intecambia qubits con las siguientes puertas:



Vamos a entender este circuito a fondo:
El circuito se lee de izquierda a derecha y cada linea representa un estado. En este circuito se concatenan tres operaciones CNOT de la siguiente manera:

$$|a, b\rangle \to |a, a\rangle \to |a \oplus (a \oplus b), a \oplus b\rangle = |b, a \oplus b\rangle \to |b, (a \oplus b) \oplus b\rangle = |b, a\rangle \tag{3.27}$$

Por lo tanto, el efecto de este circuito es intercambiar el estado de dos qubits, que a partir de ahora representaremos como



3.3.3 Copiar qubits

Supongamos que nos interesa un circuito cuántico que nos permita copiar qubits. En el caso de transportar qubits, Sin embargo, vamos a tratar de encontrar ahora una función que pueda copiar qubits, buscamos

$$V : \mathbb{V} \otimes \mathbb{V} \to \mathbb{V} \otimes \mathbb{V} \tag{3.28}$$

Que realice la operación

$$C(|x\rangle \otimes 0) = |x\rangle \otimes |x\rangle \tag{3.29}$$

Parece una operación igual de factible que la que hemos programado en 3.3.2. Vamos a comprobarlo, si C es un candidato para operador de clonación, entonces en los estados de la base se comporta como

$$C(|0\rangle \otimes |0\rangle) = |0\rangle \otimes |0\rangle \tag{3.30}$$

y

$$C(|1\rangle \otimes |0\rangle) = |1\rangle \otimes |1\rangle$$

Si queremos copiar un estado en superposición, tenemos que, necesariamente

$$C(\frac{|x\rangle + |y\rangle}{\sqrt{2}} \otimes 0) = \frac{|x\rangle + |y\rangle}{\sqrt{2}} \otimes \frac{|x\rangle + |y\rangle}{\sqrt{2}} \tag{3.31}$$

Ahora, como C tiene que ser una operación lineal, por lo tanto:

$$C(\frac{|x\rangle + |y\rangle}{\sqrt{2}} \otimes 0) = \frac{1}{\sqrt{2}} C((|x\rangle + |y\rangle) \otimes 0) = \frac{1}{\sqrt{2}}(C(|x\rangle \otimes 0) + C(|y \otimes 0\rangle) = \\ = \frac{1}{\sqrt{2}}((|x\rangle \otimes |x\rangle) + (|y\rangle \otimes |y\rangle)) = \frac{(|x\rangle \otimes |x\rangle) + (|y\rangle \otimes |y\rangle)}{\sqrt{2}} \tag{3.32}$$

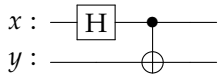
Pero,

$$\frac{(|x\rangle \otimes |x\rangle) + (|y\rangle \otimes |y\rangle)}{\sqrt{2}} \neq \frac{|x\rangle + |y\rangle}{\sqrt{2}} \otimes \frac{|x\rangle + |y\rangle}{\sqrt{2}} \tag{3.33}$$

Por lo tanto, hemos demostrado que, aunque C es una aplicación de conjuntos, no es una aplicación lineal, por lo tanto no está permitida.
Esta afirmación y esta demostración rápida tiene raices verdaderamente profundas en el campo de la computación cuántico ya que es consecuencia del No-Cloning Theorem que de hecho tiene una prueba muy sencilla que se recomienda echar un vistazo en [7]. Hay otra forma de ver el fallo del circuito para copiar bits, basada en la intuición de que un qubit contiene de algún modo información "oculta" no accesible directamente por la medición. Una vez que se mide un qubit, el estado del otro está completamente completamente determinado, y no se puede obtener información adicional sobre a y b. En este sentido, la información extra oculta que llevaba el qubit original se perdió en la primera medición, y no puede recuperarse. Sin embargo, si el qubit ha sido copiado, entonces el estado del debería seguir conteniendo parte de esa información oculta. Por lo tanto, una copia no puede haber sido creada.

3.3.4 Estados de bell

Vamos a considerar ahora los estados de Bell, son estados muy útiles, particularmente de cara a la creación de algoritmos como veremos más adelante y, además, contienen propiedades muy particulares de la computación cuántica que dan lugar a resultados sorprendentes [8]. El circuito que usamos para crear los estados de Bell es



Este circuito, da lugar a los estados β_{xy} que vienen dados por

$$|00\rangle \to \frac{|00\rangle + |11\rangle}{\sqrt{2}} = |\beta_{00}\rangle \tag{3.34}$$

$$|01\rangle \to \frac{|01\rangle + |10\rangle}{\sqrt{2}} = |\beta_{01}\rangle \tag{3.35}$$

$$|10\rangle \rightarrow \frac{|00\rangle - |11\rangle}{\sqrt{2}} = |\beta_{10}\rangle \tag{3.36}$$

$$|11\rangle \rightarrow \frac{|01\rangle - |10\rangle}{\sqrt{2}} = |\beta_{11}\rangle \tag{3.37}$$

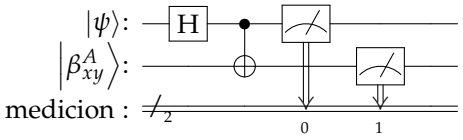
Y la ecuación general es

$$|\beta_{xy}\rangle = \frac{|0, y\rangle + (-1)^x |1, \bar{y}\rangle}{\sqrt{2}} \tag{3.38}$$

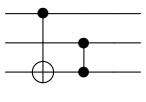
Como hemos mencionado, estos estados serán recurrentes en la generación de algortimos, también son conocidos como pares EPR.

3.3.5 Teleportación cuántica

Ahora, vamos a apicar las técnicas que acabamos de aprender para entender un resultado bastante profundo de la mecánica cuántica muy sorprendente. Ahora que conocemos los estados de Bell, vamos a ver una aplicación de ellos. Conozcamos a dos amigos de cualquier persona que haya estudiado comunicaciones, criptografía o información, Alice y Bob. Alice y Bob viven muy lejos pero un día hace tiempo que estuvieron juntos generaron un par EPR y cada uno se llevó un qubit distinto.
Pasados unos años, Bob está escondido y Alice quiere enviarle un nuevo qubit $|\psi\rangle$ a Bob, ella no sabe cuál es el estado del qubit y, además, solo puede enviar información *clásica* a Bob. Por suerte para Alice, la teleportación cuántica es una manera de usar un par EPR para enviar $|\psi\rangle$ a Bob, combinado con el envío de un poco de información clásica. Veamos cómo se puede llevar esto a cabo.
Alice hace interactuar el qubit $|\psi\rangle$ con su mitad del par EPR usando el circuito que hemos visto en la sección 3.3.4 y realiza una medición de los dos qubits en su posesión, obteniendo uno de los cuatro posibles estados de la base. Le envía a bob cuál de los cuatro estados es el que ha medido y con esa información Bob realiza cierta operación sobre su qubit del par EPR dependiendo de la medición de Alice. De esa manera, Bob puede recuperar el qubit $|\psi\rangle$. Vamos a denotar como $\left|\beta^A_{xy}\right\rangle$ al qubit del par EPR que pertenece a Alice y como $\left|\beta^B_{xy}\right\rangle$ al que pertenece a Bob. Como ya hemos comentado, $|\psi\rangle$ es el qubit que queremos teleportar. Alice por su cuenta tiene que realizar el siguiente circuito



La medición de este circuito devuelve $|00\rangle$, $|01\rangle$, $|10\rangle$ o $|11\rangle$, resultado el cuál Alice tiene que enviar a Bob. Una vez que Bob esté en posesión de esta información, tiene que realizar la siguiente operación, que depende de la medición de Alice. Si la medición de Alice es $|00\rangle$, entonces Bob no tiene que hacer nada, su qubit será directamente el estado $|\psi\rangle$, si Alice mide el estado $|01\rangle$, a Bob le basta aplicar la puerta X a su qubit para obtener $|\psi\rangle$. Si la medición fuera $|10\rangle$, tiene que aplicar la puerta Z y, si la medición es $|11\rangle$, aplicará primero la puerta X y después la Z. De esta manera, una vez Alice ha aplicado el circuito anterior, Bob debe aplicar el circuito



Donde lo m1 es la medición del primer qubit y m2 la del segundo, luego serán seguro $|0\rangle$ o $|1\rangle$, por lo tanto, se aplican el orden las puertas Controlled-X y Controlled-Z, cumpliendo de esta manera que ahora el qubit que inicialmente era $\left|\beta^B_{xy}\right\rangle$, ahora es $|\psi\rangle$.

Medición de Alice	Estado del qubit de Bob
00	$(\alpha \left\|0\right\rangle + \beta \left\|1\right\rangle)$
01	$(\alpha \left\|1\right\rangle + \beta \left\|0\right\rangle)$
10	$(\alpha \left\|0\right\rangle - \beta \left\|1\right\rangle)$
11	$(\alpha \left\|1\right\rangle - \beta \left\|0\right\rangle)$

Por último, vamos a ver esto matemáticamente. Vamos a considerar el circuito conjunto en el que tenemos tres qubits, $|\psi\rangle$, $\left|\beta_{xy}^{A}\right\rangle$ y $\left|\beta_{xy}^{B}\right\rangle$. Por simplicidad en el ejemplo, vamos a suponer que

$$\left|\beta_{xy}\right\rangle = \left|\beta_{xy}^{A}\right\rangle \left|\beta_{xy}^{B}\right\rangle = \left|\beta_{00}\right\rangle \tag{3.39}$$

Por lo tanto, el estado inicial es

$$|\psi_0\rangle = |\psi\rangle \left|\beta_{00}\right\rangle = \frac{1}{\sqrt{2}}(\alpha \left|0\right\rangle (|00\rangle + |11\rangle) + \beta \left|1(|10\rangle + |01\rangle)\right\rangle) \tag{3.40}$$

Ahora, Alice aplica el circuito que hemos visto a la parte que posee ella, aplicando la puerta CNOT y luego Hadamard, de esta manera, nos queda el siguiente estado:

$$|\psi_2\rangle = \frac{1}{2}(\alpha(|0\rangle + |1\rangle)(|00\rangle + |11\rangle) + \beta(|0\rangle - |1\rangle)(|10\rangle + |01\rangle)) \tag{3.41}$$

De esta ecuación, obtenemos el estado en el que estará el qubit de Bob después de ser medido, aunque depende de la medición que haya realizado Alice. Es por esto, que ahora para poder obtener Bob el qubit original $|\psi\rangle$ tiene que realizar una transformación que depende de la medición que haya hecho Alice. Es necesario incidir en el hecho de que sin la medición de Alice, Bob nunca puede estar seguro de que tenga el estado $|\psi\rangle$, esto es lo que impide que se utilice la teleportación para transmitir información más rápido que la velocidad de la luz. Como sabemos, la teoría de la relatividad nos dice que si consiguieramos esto, podríamos enviar información al pasado. Es clave la observación de que, sin el pedacito de información cuántica, la teleportación no aporta ninguna información. Al explotar el recurso físico del entrelazamiento, el teletransporte cuántico sirve como primitivo clave en una variedad de tareas de información cuántica y representa un importante bloque de construcción para las tecnologías cuánticas, con un papel fundamental en el progreso continuo de la comunicación cuántica, la computación cuántica y las redes cuánticas [9].

3.3.6 Simulación cuántica

Una de las aplicaciones prácticas más importantes de la computación es la simulación de sistemas físicos. Por ejemplo, en el diseño de ingeniería de un nuevo, el análisis de elementos finitos y el modelado se utilizan para garantizar la seguridad y minimizar el coste. Los coches se hacen ligeros, estructuralmente sólidos, atractivos y baratos, utilizando diseño asistido por ordenador. La ingeniería aeronáutica moderna depende en gran medida de las simulaciones de dinámica de fluidos computacional para el diseño de aviones. Las armas nucleares ya no se (en su mayor parte), sino que se prueban mediante una exhaustiva modelización computacional. Sin embargo, simular un sistema cuántico totalmente general en un ordenador clásico sólo es posible para sistemas muy pequeños, debido a la escala exponencial del espacio de Hilbert con el tamaño del sistema cuántico. Para apreciar lo rápido que esto nos lleva más allá de los recursos computacionales razonables, consideremos la memoria clásica necesaria para almacenar un estado completamente general $|\psi_n\rangle$ de n qubits (sistemas cuánticos de dos estados). El espacio de Hilbert para n qubits se extiende por 2^n estados ortogonales, etiquetados $|j\rangle$ con $0 \leq j \leq 2^n$. Los n qubits pueden estar en una superposición de todos ellos en diferentes proporciones.
Para almacenar esta descripción del estado en un ordenador clásico, necesitamos almacenar todos los números complejos cj. Cada uno de ellos requiere dos números de coma flotante (partes real e imaginaria). Utilizando 32 bits (4 bytes) para cada número en coma flotante, un estado cuántico de n=27 qubits requerirá 1 Gbyte de memoria -un ordenador de sobremesa nuevo en 2010 probablemente tenga entre 2 y 4 Gbytes de memoria en total-. Cada qubit adicional duplica la memoria, por lo que 37 qubits necesitarían

un Terabyte de memoria -un ordenador de sobremesa nuevo en 2010 probablemente tenga un disco duro de este tamaño-. El tiempo que se necesitaría para realizar cualquier cálculo útil sobre este tamaño de datos es, en realidad, lo que se convierte en el factor limitante. Como mencionábamos en 1.1. Richard Feynman ya comentaba que lo mejor para simular sistemas cuánticos sería ordenadores cuánticos. De hecho, tanto él como Yuri Manin propusieron el simulador cuántico en [10][11].
La tarea fundamental de la simulación cuántica será normalmente evaluar la evolución en el tiempo de un sistema cuántico para un determinado Hamiltoniano. A raiz de los postulados de la mecánica cuántica y resolviendo la ecuación de Schrödinger

$$i\hbar\frac{d}{dt}|\Psi\rangle = H|\Psi\rangle \tag{3.42}$$

Que nos lleva a simular la siguiente ecuación

$$|\Psi(t)\rangle = exp(i\hat{H}t)|\Psi(0)\rangle \tag{3.43}$$

Para cierto estado inicial $|\Psi(0)\rangle$ y el Hamiltoniano $\hat{H}$, que puede ser independiente del método. En muchos casos, lo que se busca son las propiedades de un sistema gobernado por el Hamiltoniano particular, y la evolución cuántica pura es suficiente. Para los sistemas cuánticos abiertos en los que el acoplamiento a otro sistema o entorno desempeña un papel, se utilizará en su lugar la ecuación maestra apropiada. Existen muchos métodos y técnicas de simulación que se pueden profundizar, una referencia muy buena para ello es [12].

Chapter 4

Algoritmos cuánticos

Ya hemos construido la arquitectura matemática que da lugar a la computación cuántica, ahora llega qué hacer con ella. Las ideas y objetivos del trabajo se solidifican en los algoritmos como veremos en el siguiente capítulo.
Los algoritmos suelen estar por delante del desarrollo del hardware y la computación cuántica no es una excepción en ello, existieron muchos algoritmos cuánticos antes de que cualquier ordenador cuántico existiera. Los algoritmos manipulan qubits para resolver problemas y, en general, resuelven estos problemas de manera más eficiente que los ordenadores clásicos. No se pretende dar una explicación detallada de cada algoritmo pues esto es algo verdaderamente extenso, sin embargo, una de las mejores maneras de aprender es haciendo, por lo que la mejor manera de prepararse para entender un poco más a fondo lo que es un programa cuántico es con estos ejemplos. Por ello, cada algoritmo lo podemos encontrar en el Jupyter Notebook que hemos programado lo que puede resultar muy didáctico para entender cómo se usan.

4.1 Algoritmo de Deutsch's

El algoritmo más simple que podemos encontrar, y que nos sirve para tomar cierta perspectiva para desarrollar cierta intuición es el algoritmo de Deutsch, el algoritmo surge como respuerta a un problema artificial. Esta parte es muy interesante para entender el algoritmo de Deutsch-Jozsa que veremos más adelante y que tiene mayor interés matemático.
Consideremos una función

$$f : \{0,1\} \to \{0,1\} \tag{4.1}$$

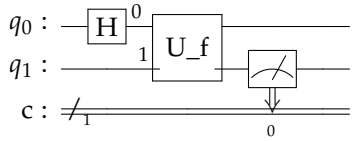
Existen cuatro funciones posibles, llamaremos balanceadas a las que $f(0) \neq f(1)$ y constantes a las que $f(0) = f(1)$. De las cuatro, dos son balanceadas y dos son constantes.
Supongamos ahora que nos dan a función como una caja negra, es decir, nos sabemos como está definida la función, sólo podemos dar inputs y ver cuáles son los outputs. Queremos diseñar un programa que dada una función como una caja negra, evalúe si es balanceada o constante.
Veamos un ejemplocon Python de cómo se podría resolver este problema en computación clásica.

```
if f(0) = 0:
  if f(1) = 0:
    print("Constant")
  else:
    print("Balanced")
else:
  if f(1) = 0:
    print("Balanced")
  else:
    print("Constant")
```

Un vez visto esto, pensemos en la implementación en computación cuántica. El manejo e implementación de la función f en computación clásica no nos supone ningún problema, sin embargo, en computación clásica, tenemos que exigir que cualquier puerta sea reversible y unitaria. Para evaluar una función f, la caja negra U_f será la puerta cuántica que usaremos para evaluar un input cualquiera.

$$U_f : |x\rangle |y\rangle \rightarrow |x\rangle |y \oplus f(x)\rangle \tag{4.2}$$

La clave de este algoritmo cuántico es que en vez de evaluar la función dos veces, ponemos en superposición los estados y una sóla evaluación de f, con algo de manipulación cuántica, nos da la solución.
Vamos a desarrollar el algoritmo poco a poco.
Comenzamos poniendo el estado de arriba en superposición de estados, como ya hemos visto en la sección **??**, todos los qubits comienzan en el estado $|0\rangle$ por defecto, luego basta con aplicar una puerta H a este estado, de esta manera nos queda el siguiente circuito:

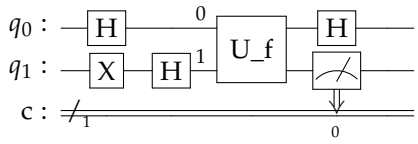


Veamos los distintos estados que atraviesa el circuito:
En álgebra, el circuito se corresponde con:

$$U_f(H \otimes I)(|0\rangle \otimes |0\rangle) \tag{4.3}$$

Comienza con $|\psi_0\rangle = |0\rangle |0\rangle$, al aplicar la puerta H obtenemos el estado $|\psi_1\rangle = \frac{|0,0\rangle + |1,0\rangle}{\sqrt{2}}$.
Después, multiplicamos por U_f y nos queda: $|\psi_2\rangle = \frac{|0,f(0)\rangle + |1,f(1)\rangle}{\sqrt{2}}$. Sin embargo, esto nos devolverá un estado en superposición que al medirlo nos dará un resultado no determinista, por lo que aún tenemos que refinar el algoritmo. Tras cierta prueba y error se encuentra un algortimo que sea capaz de darnos una respuesta determinista a nuestro problema. Véase el siguiente circuito que da lugar al algoritmo:

(4.4)



En términos matriciales:

$$(H \otimes I)U_f(H \otimes H)\,|0,1\rangle \tag{4.5}$$

Nótese que hemos omitido la puerta X y hemos tomado como estado inicial el estado $|0,1\rangle$.

De nuevo, vamos a recorrer este circuito:

$$|\psi_1\rangle = (\frac{|0\rangle + |1\rangle}{\sqrt{2}})(\frac{|0\rangle - |1\rangle}{\sqrt{2}}) = \frac{|0,0\rangle - |0,1\rangle + |1,0\rangle - |1,1\rangle}{2} = \begin{pmatrix} \frac{1}{2} \\ -\frac{1}{2} \\ \frac{1}{2} \\ -\frac{1}{2} \end{pmatrix} \tag{4.6}$$

Una vez que multiplicamos por f, nos queda un resultado muy interesante [4]:

$$|\psi_2\rangle = (\frac{(-1)^{f(0)}\,|0\rangle + (-1)^{f(1)}\,|1\rangle}{\sqrt{2}})(\frac{|0\rangle - |1\rangle}{\sqrt{2}}) \tag{4.7}$$

Por lo tanto el resultado depende de si f es balanceada o constante:

$$|\psi_2\rangle = \begin{cases} (\pm 1)(\frac{|0\rangle+|1\rangle}{\sqrt{2}})(\frac{|0\rangle-|1\rangle}{\sqrt{2}}) & f \quad constante \\ \\ (\pm 1)(\frac{|0\rangle-|1\rangle}{\sqrt{2}})(\frac{|0\rangle-|1\rangle}{\sqrt{2}}) & f \quad constante \end{cases} \tag{4.8}$$

Como la puerta H lleva $\frac{|0\rangle+|1\rangle}{\sqrt{2}}$ a $|0\rangle$ y lleva $\frac{|0\rangle-|1\rangle}{\sqrt{2}}$ a $|1\rangle$ por lo tanto, al medir el qubit de arriba nos quedará $|0\rangle$ si la función es constante y $|1\rangle$ si es balanceada. Nótese que la medición de $-|1\rangle$ es la misma que de $|1\rangle$. Con esto, completaríamos el algoritmo de Deutsch, igual que lo hemos escrito en Python para computación clásica, veámoslo para computación cuántica con qiskit. Este nos devuelve el circuito con el oráculo, pero no dice aún si la función es balanceada o constante. Esto se debe a que, si lo simularamos, tendríamos una respuesta unívoca y de hecho, en el ejemplo de uso con el simulador se ve claramente que tendríamos la respuesta. Sin embargo, en caso de ejecutarlo en un ordenador cuántico real, el resultado puede dar errores y habría que definir cuantos intentos realizar asi como nuestra tolerancia a los errores. Se puede ver dicho ejemplo de uso en el anexo de Jupyter Notebook. No lo simularemos en ordenadores cuánticos reales ya que nuestro objetivo está en el software y para estos programas la simulación es suficiente.

```
def Deutsch_Algorithm(oracle):
    """

    Parameters
    ----------
    oracle : Circuito cu ntico U_f, dos qubits de entrada y dos qubits de salida

    Returns
    -------
    f balanceada o constante

    """
    qc = QuantumCircuit(2,1)
    qc.h(0)
    qc.x(1)
    qc.h(1)
    qc.append(oracle,[0,1])
    qc.h(0)
    qc.measure(0,0)
    return qc
```

Lo que hace el circuito es básicamente un cambio de base, comenzamos en la base canónica. La primera matriz de Hadamard se utiliza como matriz de cambio de base para pasar a una superposición equilibrada de estados básicos. Mientras que en esta base no canónica, evaluamos f con el qubit inferior en superposición. La última matriz de Hadamard se utiliza como una matriz de cambio de base para volver a la base canónica.

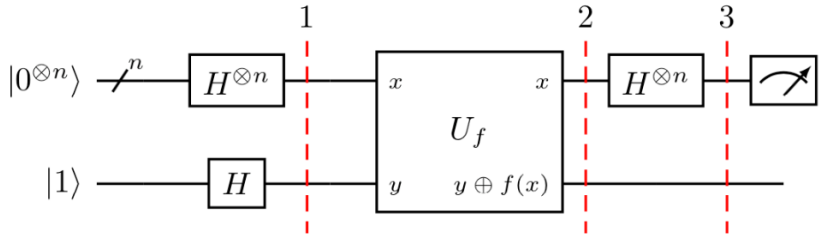
4.2 Algoritmo de Deutsch-Jozsa

El algoritmo de Deutsch-Jozsa generaliza el algoritmo de Deutsch que ya hemos visto, el de Deutsch-Jozsa es un ejemplo claro de problema que resulta sencillo de resolver para un ordenador cuántico pero muy complejo para un ordenador clásico [13] que en el peor de los casos necesita $2^{n-1}+1$ iteraciones para determinar si la función es constante o balanceada. En el año 1992 Deutsch y Jozsa encontraron la solución al problema para un número arbitrario de bits de entrada. Sea una función

$$f : \{0,1\}^n \to \{0,1\} \tag{4.9}$$

Diremos que f es balanceada si exactamente la mitad de los inputs tienen como imagen el 0 y que es constante si *todos* los inputs tienen imagen o bien 0 o bien 1.

Clásicamente, en el mejor de los casos, dos consultas al oráculo pueden determinar si la función booleana oculta $f(x)$ está equilibrada. En el peor de los casos, si seguimos viendo la misma salida para cada entrada que probamos, tendremos que comprobar exactamente la mitad de todas las entradas posibles más una para estar seguros de que $f(x)$ es constante. Como el número total de entradas posibles es 2^n esto implica que necesitamos $2^{n-1}+1$ entradas de prueba para tener la certeza de que $f(x)$ es constante en el peor de los casos. [14]
Veamos ahora el caso cuántico, la clave del algoritmo de Deutsch consistía en la superposición, para este algoritmo generalizaremos esas mismas ideas [15]. El circuito correspondiente al algoritmo es



Vamos a recorrer ahora los distintos pasos del algoritmo.
Paso 1 : Preparamos el estado incial para que sea

$$|\psi_0\rangle = |0\rangle^{\otimes n} |1\rangle \tag{4.10}$$

Paso 2 : Aplicamos la puerta de Hadamard a cada qubit:

$$|\psi_1\rangle = \frac{1}{\sqrt{2^{n+1}}} \sum_{x=0}^{2^n-1} |x\rangle \left(|0\rangle - |1\rangle\right) \tag{4.11}$$

Paso 3 : Aplicar el oráculo U_f a los n+1 qubits. Recordamos que $U_f |x\rangle |y\rangle = |x\rangle |x(x)\rangle$ y tenemos:

$$|\psi_2\rangle = \frac{1}{\sqrt{2^{n+1}}} \sum_{x=0}^{2^n-1} |x\rangle \left(|f(x)\rangle - |1 \oplus f(x)\rangle\right) = \frac{1}{\sqrt{2^{n+1}}} \sum_{x=0}^{2^n-1} (-1)^{f(x)} |x\rangle \left(|0\rangle - |1\rangle\right) \tag{4.12}$$

Ya que para cada x, $f(x)$ es o bien 0 o bien 1. **Paso 4** : Ahora, aplicamos de nuevo las puertas de Hadamard

pero sólo a los n primeros qubits, el último ya no nos interesa y podemos ignorarlo.

$$|\psi_3\rangle = \frac{1}{2^n} \sum_{x=0}^{2^n-1} (-1)^{f(x)} \left(\sum_{y=0}^{2^n-1} (-1)^{x \cdot y} |y\rangle \right) = \frac{1}{2^n} \sum_{y=0}^{2^n-1} \left(\sum_{x=0}^{2^n-1} (-1)^{f(x)} (-1)^{x \cdot y} \right) |y\rangle \tag{4.13}$$

Paso 4 : Por último, medimos los n primeros qubits, la probabilidad de medir $|0\rangle^{\otimes n}$ es $|\frac{1}{2^n} \sum_{x=0}^{2^n-1} (-1)^{f(x)}|^2$

lo cual vale 1 si $f(x)$ es constante y 0 si es balanceada.
La razón por la que esto funciona es muy similar al anterior algoritmo, cuando el oráculo es constante, aplicarlo a los qubits de entrada no tiene efecto, por lo que como H es su propia inversa, en el paso 4

invertimos el paso 2 y obtenemos el mismo estado.

$$H^{\otimes n} \begin{pmatrix} 1 \\ 0 \\ 0 \\ . \\ . \\ . \\ 0 \end{pmatrix} = \frac{1}{\sqrt{2^n}} \begin{pmatrix} 1 \\ 1 \\ 1 \\ . \\ . \\ . \\ 1 \end{pmatrix} \tag{4.14}$$

Al aplicar U_f al resultado de 4.14, obtenemos el mismo vector, luego

$$H^{\otimes n} \frac{1}{\sqrt{2^n}} \begin{pmatrix} 1 \\ 1 \\ 1 \\ . \\ . \\ . \\ 1 \end{pmatrix} = \begin{pmatrix} 1 \\ 0 \\ 0 \\ . \\ . \\ . \\ 0 \end{pmatrix} \tag{4.15}$$

Sin embargo, si el oráculo es balanceado, a causa del phase kickback, se añade un fase negativa a la mitad de los estados.

$$U_f \frac{1}{\sqrt{2^n}} \begin{pmatrix} 1 \\ 1 \\ 1 \\ . \\ . \\ . \\ 1 \end{pmatrix} = \frac{1}{\sqrt{2^n}} \begin{pmatrix} -1 \\ 1 \\ -1 \\ . \\ . \\ . \\ 1 \end{pmatrix} \tag{4.16}$$

Para entender el algoritmo mejor, se recomienda trabajar en un ejemplo sencillo como podría ser la función $f(x_0, x_1) = x_0 \oplus x_1$ donde el oráculo correspondiente a esta función es $U_f |x_1, x_0\rangle = (-1)^{f(x_0,x_1)} |x\rangle$. Se puede encontrar el algoritmo programado en qiskit asi como ejemplos de uso en el anexo de Jupyter Notebook.

4.3 Algoritmo de Periodicidad de Simon

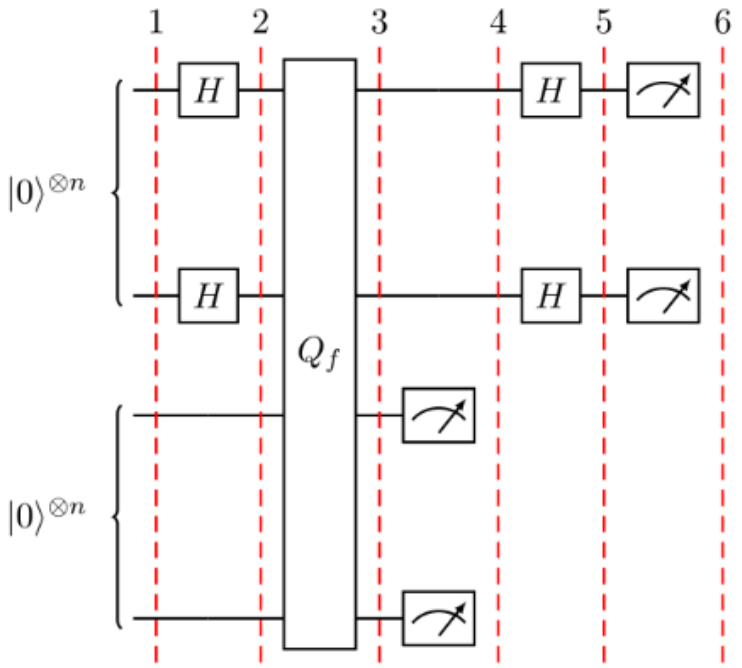
El algoritmo de Simon, introducido por primera vez en la referencia [16], fue el primer algoritmo cuántico que mostró un aumento de velocidad exponencial frente al mejor algoritmo clásico en la resolución de un problema específico. Esto inspiró los algoritmos cuánticos basados en la transformada cuántica de Fourier, que se utiliza en el algoritmo cuántico más famoso: El algoritmo de factorización de Shor. Los cuales veremos en las siguientes secciones

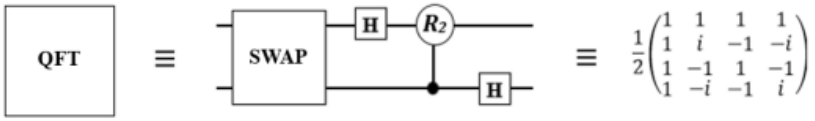
4.4 Quantum Fourier Transform

[PENDIENTE]

4.5 Algoritmo de factorización de Shor

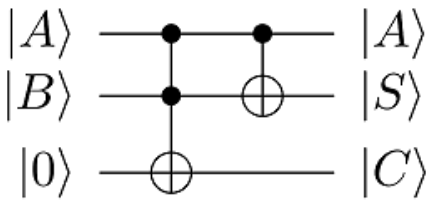
[PENDIENTE]



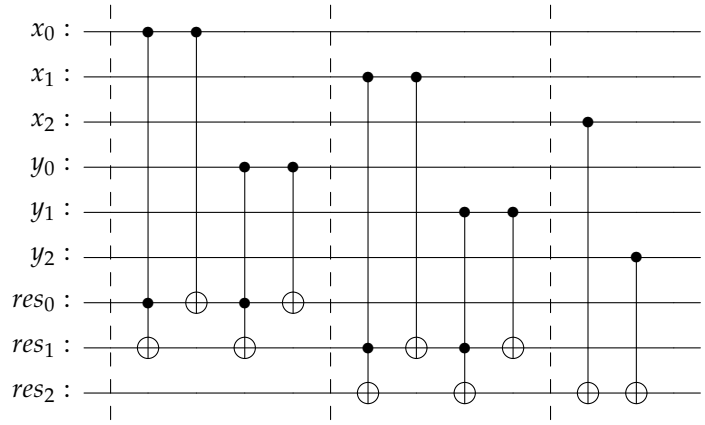


4.6 Algoritmo para sumar bits

El articulo que dió lugar a este TFG [17], está basado en metamorphic testing para un algoritmo cuántico que implementa la suma modular, por lo que es necesario, al menos, mencionar dicho algoritmo y entenderlo, ya que luego será también mencionado en la sección **??**.
Veamos, en primer lugar, como podemos construir un circuito que realice la suma modular de dos números de un sólo bit. Esto también es conocido como *Half-adder*. Toma las entradas A y B y devuelve la suma $S = A \oplus B$ y el resto $C = A \cdot B$ el circuito que corresponde a este algoritmo es

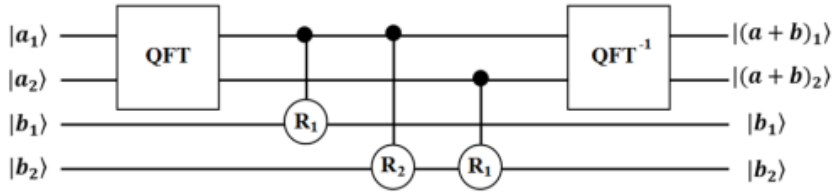


En este circuito, implementamos en primer lugar una puerta de Toffoli seguida de una puerta CNOT, y en la salida obtenemos $|A\rangle$, la suma de los bits y el resto. Basándonos en esta idea para un bit y siguiendo el trabajo realizado en el artículo [17], exponemos el circuito que calcula la suma modular de números de 3 bits.



En este circuito, el resultado de la suma modular se encuentra almacenado en res_0, res_1 y res_2 y uno de los detalles es que la generalización para n bits es muy sencilla ya que basta replicar las puertas que encontramos entre las dos primeras barreras n-1 veces, aplicándolo a los bits correspondientes, y para la última aplicar tan sólo dos puertas cnot. Se puede encontrar esta generalización en el código anexo, justo debajo del caso de 3 bits. Ahí podemos encontrar la generalización para la suma modular de n bits, su simulación y código para sumar dos números arbitrarios.

Otro algoritmo muy interesante es el propuesto en [18], que no vamos a desarrollar en detalle, pero exponemos el circuito utilizado:



Chapter 5

Introducción a metamorphic testing y mutantes

Software testing es un método para comprobar si el software real se ajusta a los requisitos esperados y para garantizar que el producto de software está libre de defectos. Implica la ejecución de componentes de software/sistema mediante herramientas manuales o automatizadas para evaluar una o varias propiedades de interés. El objetivo del software testing es identificar errores, lagunas o falta de requisitos en contraste con los requisitos reales. Consiste un enfoque generalizado para el aseguramiento y la verificación de la calidad del software. Un elemento clave del testing es que permite encontrar errores pero no demostrar la ausencia de ellos.

5.1 Software testing

Las pruebas de software son un enfoque principal para el aseguramiento y la verificación de la calidad del software. Sin embargo, se enfrenta a dos problemas fundamentales: el problema del oráculo y el del conjunto de pruebas fiable [19].
Aunque el software testing puede determinar la corrección del software bajo la suposición de algunas hipótesis específicas (véase la jerarquía de la dificultad de las pruebas más adelante), las pruebas no pueden identificar todos los fallos del software [20], sino que proporcionan una crítica o comparación que compara el estado y el comportamiento del producto con los oráculos de las pruebas, es decir, los principios o mecanismos por los que alguien podría reconocer un problema. Estos oráculos pueden incluir (pero no se limitan a) especificaciones, contratos, productos comparables, versiones anteriores del mismo producto, inferencias sobre el propósito previsto o esperado, expectativas de los usuarios o clientes, normas pertinentes, leyes aplicables u otros criterios.

Uno de los principales objetivos de las pruebas es detectar los fallos del software para poder descubrir y corregir los defectos. Las pruebas no pueden establecer que un producto funciona correctamente en todas las condiciones, sino sólo que no funciona correctamente en condiciones específicas[4] El alcance de las pruebas de software puede incluir el examen del código, así como la ejecución de ese código en varios entornos y condiciones, así como el examen de los aspectos del código: si hace lo que se supone que debe hacer y si hace lo que necesita hacer. En la cultura actual de desarrollo de software, una organización de pruebas puede estar separada del equipo de desarrollo. Los miembros del equipo de pruebas desempeñan diversas funciones.
La información derivada de las pruebas de software puede utilizarse para corregir el proceso de desarrollo de software [21]. Los fallos del software se producen a través del siguiente proceso: Un programador comete un error (mistake), que da lugar a un fallo (defect, bug) en el código fuente del software. Si este fallo se ejecuta, en determinadas situaciones el sistema producirá resultados erróneos, provocando un fallo

(failure).

5.2 Mutation testing

La prueba de mutación es un tipo de prueba de software en la que se cambian/mutan ciertas declaraciones del código fuente para comprobar si los casos de prueba son capaces de encontrar errores en el código fuente. El objetivo de las pruebas de mutación es garantizar la calidad de los casos de prueba en términos de robustez que debe fallar el código fuente mutado.

Los cambios realizados en el programa mutante deben ser extremadamente pequeños para que no afecten al objetivo general del programa. Mutation testing también se denomina estrategia de prueba basada en fallos, ya que implica la creación de un fallo en el programa y es un tipo de white-box testing [22] que se utiliza principalmente para las Unit Testing [23].
Cada versión mutada se llama mutante y las pruebas detectan y rechazan los mutantes haciendo que el comportamiento de la versión original difiera del mutante. Esto se llama matar al mutante. Los conjuntos de pruebas se miden por el porcentaje de mutantes que matan. Se pueden diseñar nuevas pruebas para eliminar otros mutantes. Los mutantes se basan en operadores de mutación bien definidos que imitan errores típicos de programación (como el uso de un operador o un nombre de variable incorrecto) o fuerzan la creación de pruebas relevantes (como la división de cada expresión por cero).

La mutación se propuso originalmente en 1971, pero perdió fervor debido a los altos costes que implicaba. Ahora, de nuevo, ha cobrado fuerza y se utiliza ampliamente en lenguajes como Java y XML.
A continuación describimos el proceso para llevar a cabo mutation testing [24]:

Paso 1: Los fallos se introducen en el código fuente del programa creando muchas versiones llamadas mutantes. Cada mutante debe contener un único fallo, y el objetivo es hacer que la versión mutante falle, lo que demuestra la eficacia de los casos de prueba.

Paso 2: Los casos de prueba se aplican al programa original y también al programa mutante. Los casos de prueba deben ser adecuados y se ajustan para detectar fallos en el programa.

Paso 3: Se comparan los resultados del programa original y del mutante.

Paso 4: Si el programa original y el mutante generan una salida diferente, entonces el mutante es eliminado por el caso de prueba. Por lo tanto, el caso de prueba es lo suficientemente bueno para detectar el cambio entre el programa original y el mutante.

Paso 5: Si el programa original y el programa mutante generan la misma salida, el mutante se mantiene vivo. En estos casos, es necesario crear casos de prueba más eficaces que maten a todos los mutantes.

Para ver un ejemplo de mutation testing muy simple, consideremos el siguiente código en python.

```
def program(x):
    if x<1:
        print('true')
    else:
        print('false')
```

Si a este programa le aplicamos un *operador de mutación*, que es lo que pretendemos estudiar en este trabajo, nos devolverá el mutante, que es el programa modificado. Si por ejemplo aplicamos un operador de mutación que cambie "<" por ">", tendríamos el siguiente mutante:

```
def program(x):
    if x>1:
        print('true')
    else:
        print('false')
```

El objetivo de este trabajo surge a raiz del metamorphic testing, (dar intro de la wikipedia y citar paper de Luis) que usa mutantes

5.2.1 Metamorphic testing

Como ya se ha comentado anteriormente, este trabajo surge a raíz del artículo [17], en el que el objetivo fundamental son las pruebas usando metamorphic testing para computación cuántica. Metamorphic testing es un método de testing basado en propiedades que consiste en un enfoque tanto para la generación de casos de prueba como para la verificación de los resultados de las pruebas. Un elemento central es un conjunto de relaciones metamórficas, que son propiedades necesarias de la función o el algoritmo objetivo en relación con múltiples entradas y sus salidas esperadas. Desde su primera publicación, hemos asistido a un rápido de trabajos que examinan las pruebas metamórficas desde varias perspectivas, como la de relación metamórfica, la generación de casos de prueba, la integración con otras técnicas de ingeniería del software y la validación y evaluación de los sistemas de software [25]. Veamos dos ejemplos de esto. El primero es el ejemplo clásico, supongamos que tenemos un programa con una implementación que calcula el seno de x hasta 100 cifras significativas, una relación metamórfica del seno es $sin(\pi - x) = sin(x)$. De esta manera, aunque no sepamos si el programa calcula bien un caso de prueba como $x_1 = 1$, sí podemos contrastarlo con el resultado de $x_2 = \pi - 1$. Cualquier inconsistencia, teniendo en consideración los distintos errores de redondeo del propio ordenador, indica un fallo (failure) del programa, causado por una falta (fault) de la implementación. [26].
Otro ejemplo que es verdaderamente interesante extraido de [27] y muestra lo que es el metamorphic testing de una manera muy cotidiana es el siguiente:
Supongamos que queremos ayudar a nuestro hijo con los deberes. Le preguntamos cuántos ejercicios tiene que hacer en total y nos dice que 2. Como no estamos seguros de si la respuesta es cierta, pasado un rato le preguntamos cuántos ejercicios de matemáticas tiene que hacer y nos dice que son cuatro. Nuestra regla metamórfica es que los ejercicios de matemáticas tienen que ser menos que el total de ejercicios de todas las asignaturas. Por lo tanto, nuestro test metamórfico nos ha servido para detectar que nuestro hijo es un poco *olvidadizo*. Notese que no hemos tenido que comprobar que ninguna de las respuestas era correcta y, aún más importante, el chico se ha expuesto sólo.

5.3 Quantum Testing

La computación cuántica es una disciplina que está aún por desarrollar y es muy joven aún, por lo que más aún lo es el Quantum Testing. Sin embargo, se han realizado algunos intentos para comenzar a definir procesos de ingiería de software aplicados a las computación cuántica [17][28]. La motivación para su desarrollo es bastante alta. Esto es debido a que durante la computación no podemos saber el estado de un circuito y sólo podemos saberlo realizando una medición, lo cual colapsa el estado como ya hemos visto. Por ello, el debbuging interactivo no es algo posible, es decir, cuándo programamos un programa clásico usualmente el programador realiza el proceso de debbuging interactuando directamente con los algoritmos y con las distintas etapas comprabando que funcionan como se espera. Sin embargo, en computación cuántica esto no funciona así, en secciones anteriores hemos visto ejemplos de simulación cuántica donde podíamos simular un circuito y con qiskit calculábamos el vector de estado, pero cuando el tamaño de los circuitos comienza a ser mayor, el coste computacional es desproporcionado. Es por eso que las técnicas de testing que hemos descrito anteriormente pueden resultar particularmente efectivas para quantum testing, de hecho, utilizando algoritmos como el algoritmo de búsqueda de Grover para buscar en conjuntos no estructurados, requiriendo menor potencia computacional que el algoritmo clásica como ya hemos visto [17].

Chapter 6

Operadores de mutación cuántica en Qiskit

Llegados a este punto, a lo largo del trabajo nos hemos familiarizado con la computación cuántica, hemos hecho un viaje desde la base matemática y los postulados hasta conocer el concepto de bit, jutar bits en circuitos y hacer algoritmos cuánticos y entenderlos. En paralelo, hemos conocido una introducción de lo que es el software testing fijándonos particularmente en metamorphic y mutant testing. Ya sabemos entender el título del trabajo. Sabemos lo que son los operadores de mutación, queremos aplicarlo en computación cuántica y en particular en el leguaje qiskit.

6.1 Qiskit

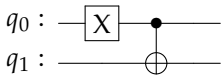
Qiskit es un software development toolkit (SDK) de código abierto para trabajar con ordenadores cuánticos a nivel de pulsos, circuitos y módulos de aplicación. Qiskit fue fundado por IBM Research para permitir el desarrollo de software para su servicio de computación cuántica en la nube, IBM Quantum Experience[29]. Se compone de elementos que trabajan juntos para hacer posible la computación cuántica. El objetivo central de Qiskit permite a los usuarios diseñar experimentos y aplicaciones y ejecutarlos en ordenadores cuánticos reales y/o simuladores clásicos. Ofrece la posibilidad de desarrollar software cuántico tanto a nivel de código máquina de OpenQASM, como a niveles abstractos adecuados para usuarios sin experiencia en computación cuántica. Esta funcionalidad es proporcionada por distintos componentes que se pueden ver en [30]. Se puede observar sus principales diferencias y el contraste con otros lenguajes desarrollados en el artículo [31].
A lo largo de este trabajo se ha tratado de dar una introducción a qiskit con la creación de los circuitos y el desarrollo de algoritmos en Qiskit por lo que se recomienda enormemente seguir el Jupyter Notebook de este trabajo y probar con los diferentes comandos expuestos para aprender. Si se quiere profundizar más, no hay mejor sitio que [32]

6.2 Operadores de mutación cuánticos

Ahora que sabemos lo que son los operadores de mutación y hemos visto ejemplos en computación clásica, vamos a extender el concepto a la computación cuántica. Vamos a hacer un ejemplo que sea más o menos análogo al de la sección 5.2 pero para un circuito cuántico. Consideremos un ejemplo sencillo de circuito cuántico:

```
qc = QuantumCircuit(2)
qc.x(0)
qc.cx(0,1)
```

Que genera el circuito:



Como los bits siempre se incian en $|0\rangle$, el circuito nos devolverá siempre el estado $|11\rangle$. Ahora, podríamos desarrollar a mano mutantes del circuito, por ejemplo, dos mutantes serían:

En ambos circuitos, se ha sustituido una puerta por otra, en el primer mutante, se ha aplicado un operador de mutación que sustituye la puerta h por una puerta x, y en el segundo, el operador de mutación cambia una puerta CNOT por una puerta swap.
El objetivo de este trabajo se sitúa en encontrar una manera sistemática de crear mutantes, es decir, crear funciones que hagan la función de un operador de mutación, las cuales nos permitan generar un mutante a partir de un circuito.

6.2.1 Qué cambia en un programa cuántico

Ahora, el lector se puede preguntar qué es lo susceptible de ser mutado en un programa cuántico. Los operadores de mutación que vamos a crear se basarán en el intercambio de puertas, habrá varios tipos de operadores dependiendo de si queremos permitir que la puerta que cambiamos afecte a un distinto número de qubits, que sea paramétrica, por ejemplo.
Para ello, en primer lugar, tenemos que saber cuáles son todas las puertas que hay en Qsikit y como se representan internamente por Qiskit. Las puertas son infinitas ya que dado cualquier circuito, podemos convertirlo en una puerta en Qiskit a través de la función circuit_to_gatecircuit(parameter_map=None, equivalence_library=None, label=None) [33]. De primeras, una vez que tenemos un circuito creado, sustituir una puerta por otra esa una tarea complicada en qiskit. Tras prueba y error así como investigación en la documentación de qiskit, se encuentra que la manera de modificar la información esencial de un circuito es a través del atributo *QuantumCircuit.data* [34]. Esta lista contiene tuplas con la información de cada una de las puertas, clásicas o cuánticas. Trabajaremos con las puertas como instrucciones.

6.2.2 Funciones para los operadores

El objetivo es crear un generador de mutantes, es decir, un operador de mutación que le demos un circuito y devuelva el mutante. Por ello, se ha creado el módulo MutantOperators. En este módulo, subido para descarga en https://github.com/JaimeDelaVegaFornie/TFG-2022.git. Para decidir qué funciones son convenientes, hay que tener en cuenta cuáles podrían ser los errores habituales y de esta manera generar los diversos mutantes.
Una de las principales tareas conseguidas a través del desarrollo de este trabajo es la de ganar un entendimiento suficiente como para considerar cuáles podrían ser los operadores de mutación principales. De esta manera, podemos completar nuestro módulo lo máximo posible sin dejarnos ninguno importante. De todas maneras, estos operadores son propuestas iniciales que tendrán que ser ampliadas según el uso que se quiera dar. Se han creado cuatro operadores fundamentales para la creación de distintos tipos de mutantes:
Operador 1:

```
gate_mutant(circ1,input_gate=None,output_gate=None)
```

Este operador tiene como función principal intercambiar puertas cuánticas. Por defecto y sin parámetros, escoge una puerta aleatoria y la cambia por otra que afecte a los mismos qubits que la anterior. También

podemos dar como argumento la puerta que queremos que sea sustituida y la nueva puerta a poner en el mutante. Devuelve un mutante del circuito y no modifica el original. Un error frecuente puede ser aplicar una puerta que no es exactamente la que queremos, por ejemplo aplicar a un qubit la puerta X en lugar de la puerta H, este operador pretende simular este tipo de errores.
Operador 2:

```
targetqubit_mutant(circ1,gate=None,target_qubit=None,final_qubit=None)
```

Este operador tiene como función principal modificar los qubits a los que afecta una puerta multiqubit. Por defecto y sin parámetros, escoge una puerta aleatoria y modifica uno del los qubits a los que afecta cambiándolo por otro cualquiera del circuito. Si se le provee una puerta en específico, lo hará sobre esta puerta. También se puede pasar como parámetro el qubit a cambiar y el qubit donde deseamos que afecte el mutante. Es fácil equivocarse cuando tenemos una puerta que afecta a varios qubits, por ejemplo, en el algoritmo de la suma, si una de las puertas de Toffoli afecta a un qubit equivocado, esto modificaria el resultado por completo. Este tipo de errores del programador son los que pretende crear este operador
Operador 3:

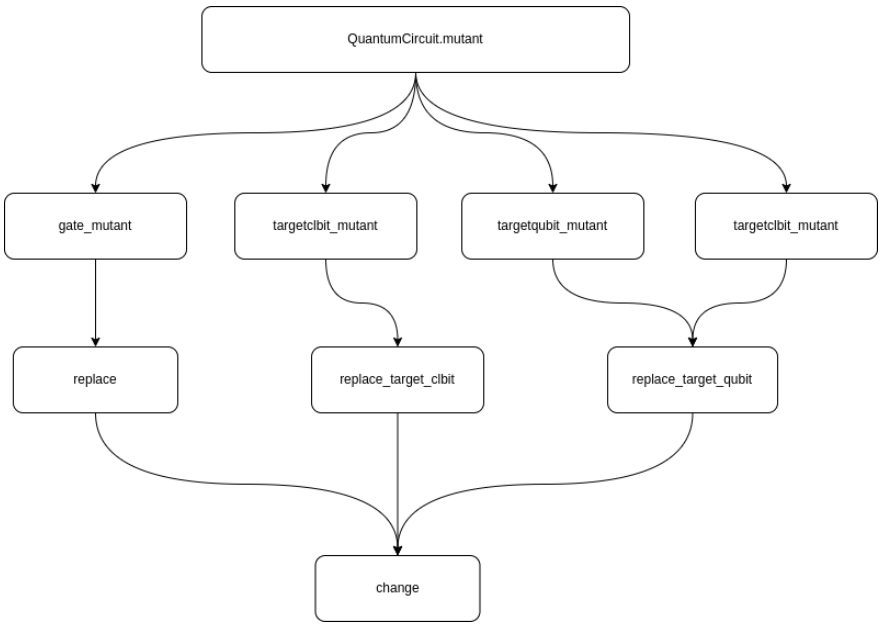
```
targetclbit(circ1,gate=None,target_clbit=None,final_clbit=None)
```

El objetivo de este operador es muy similar al anterior, sin embargo, ahora afecta a qubits clásicos. Es fácil, por ejemplo al medir, llevar la medición al bit clásico equivocado y que el resultado no tenga sentido, este tipo de errores se generar con mutantes a través de este operador.
Operador 4:

```
measure_mutant(circ1,target_qubit=None,final_qubit=None)
```

También es un error común medir el qubit equivocado, sobretodo en circuitos de muchos qubits, por ello, este operador genera un mutante del circuito original en el que una de las puertas de medición mide un qubit distinto que la puerta original. De igual forma, se puede especificar el target_qubit, lo que hará que se busque una puerta que mida este qubit, y también un final_qubit, que será el qubit que medirá la puerta nueva en el circuito mutante.

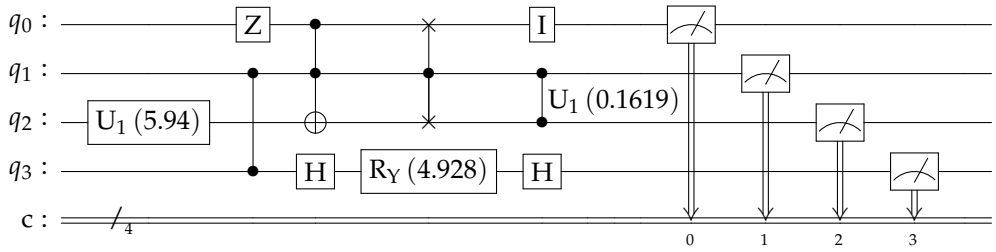
Con la creación de estos operadores se cumplen dos objetivos. A partir de cualquier circuito se pueden generar cuatro tipos de mutante de él. Este mutante puede ser aleatorio o, si tenemos alguna preferencia podemos mutar el circuito exactamente como necesitemos.
El código para este módulo se puede encontrar tanto en el anexo de Jupyter Notebook como en GitHub como módulo que se puede descargar e importar en Python.
Las funciones del módulo están relacionadas de la siguiente manera:



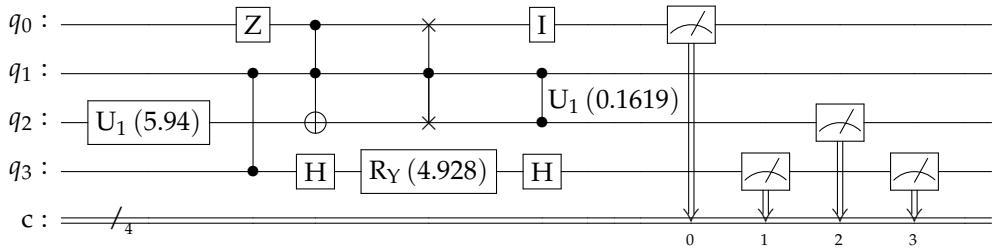
6.3 Aplicación del módulo

6.3.1 Ejemplo de uso de las funciones

Vamos a dar ejemplos de uso para que, creemos un circuito aleatorio y vamos a generar diversos mutantes, usando random_circuit obtenemos



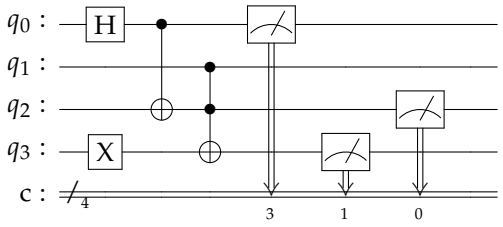
Hay muchos ejemplos de uso en el Jupyter Notebook, vamos a explicar dos casos sencillos un poco más ampliados. Imaginemos que queremos generar un mutante cualquier para nuestro circuito pero no sabemos cuál, nos vale cualquiera, en dicho caso, podemos llamar al método mutant, y obtenemos, por ejemplo, el siguiente circuito, en el que, como la función nos indica, se cambia el qubit al que afecta la medición.



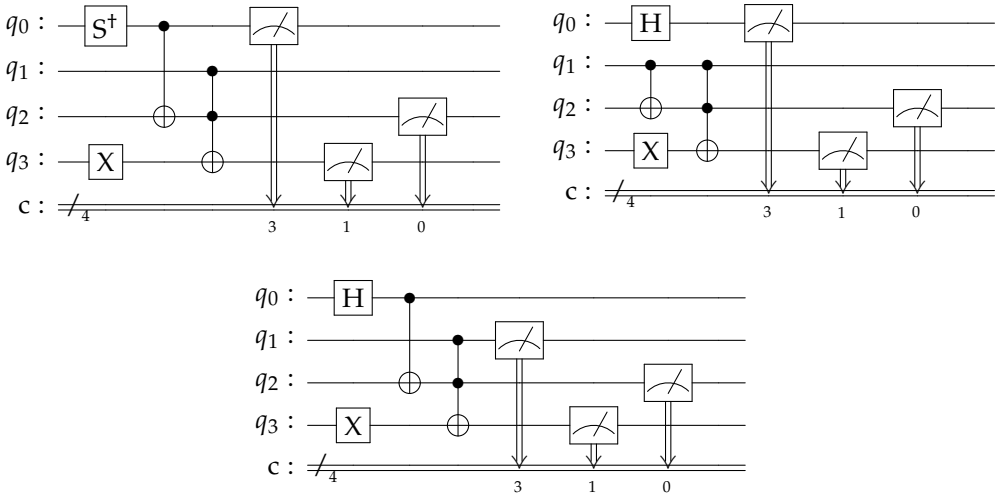
Esto se realiza con el siguiente código

```
from qiskit.circuit.random import random_circuit

circ = random_circuit(4, 4, measure=True)
c1 = circ.mutant()
c1.draw(output='mpl')
```

Se puede observar que la creación de mutantes aleatorios es tremendamente sencilla. Imaginemos ahora que, dado otro circuito, por ejemplo



Para este circuito vamos a generar tres mutantes diferentes



El primer mutante se ha creado cambiando la puerta H por la puerta $Sdg = \begin{pmatrix} 1 & 0 \\ 0 & -i \end{pmatrix}$. En el segundo, se modifica el bit de control de la puerta CNOT y en el último se cambia la medición, en lugar de medir q_0 se mide q_1. Todos estos mutantes se obtienen con las siguientes lineas de código.

```
#Creamos el circuito para mutarlo
qc = QuantumCircuit(4,4)
qc.h(0)
qc.x(3)
qc.cx(0,2)
qc.ccx(1,2,3)
qc.measure([0,3,2],[3,1,0])
#Ahora creamos tres mutantes del circuito
m1 = qc.gate_mutant(HGate(),SdgGate())
m2 = qc.targetqubit_mutant(CXGate(),qc.qubits[0],qc.qubits[1])
m3 = qc.measure_mutant(qc.qubits[0],qc.qubits[1])
```

Por lo tanto, es notable que se simplifica enormemente la creación de mutantes. Algo que remarcar es el hecho de que para seleccionar los qubits sobre los que pretendemos trabajar, lo mejor es tomarlos directamente de la lista que genera qc.qubits.

6.3.2 Aplicación en Metamorphic Testing

Veamos ahora un caso de uso del módulo a través de un ejemplo trivial de Metamorphic Testing. Para ello, vamos a utilizar el algoritmo de suma modular. Para la suma modular tenemos dos reglas metamórficas sencillas, la primera es la conmutatividad $a \oplus b = b \oplus a$ y la segunda es el elemento neutro $a \oplus 0 = a$. La primera la usaremos para testear $sumar\_bits()$ y la segunda para la función $suma\_modular()$. Para la primera regla, creamos un conjunto test y comprobamos que se cumple. A partir de aquí tenía pensado testear el algoritmo de la suma con las reglas metamórficas y luego introducir mutantes para ver que no se cumplían dichas reglas pero no estoy muy seguro de como hacerlo.

Chapter 7

Conclusión

El foco principal del trabajo era el entendimiento profundo de la computación cuántica con sus algoritmos, circuitos y puertas para poder desarrollar en el lenguaje qiskit operadores de mutación de manera análoga a los que existen en computación clásica. Este trabajo surge desde una necesidad investigadora real, como ya hemos mencionado antes, en el paper [17] encontramos generación de mutantes de manera manual para realizar metamorphic testing. Dada la proliferación de la computación cuántica, este trabajo busca de manera fundamental la creación de una libreria para qiskit que pueda ser usada para crear mutantes en general de manera automática.

7.1 Trabajo futuro

En primer lugar, algo que sería muy interesante es tratar de convertir el módulo que se ha creado en una librería estandar de qiskit que todo el mundo pueda descargar y usar.
Por otro lado, investigador que trabaje en computación cuántica en un área de testing podría alegrarse enormemente al encontrar la manera aquí desarrollada de fabricar mutantes de una manera tan sencilla sin tener que hacerlos de uno en uno. Podría realizar los mutantes que necesite o incluso dejarlo en manos del azar con la herramienta de crear mutantes aleatorios.
Por esta parte, la parte de un mutante que corresponde a los propios entresijos de qiskit quedaría solucionada. Sin embargo, falta algo fundamental, ¿Cuáles son los mejores mutantes? ¿Qué mutante elijo para cada circuito? Aunque en la computación clásica lleva muchos años de desarrollo y tiene estudios fuertes, en la computación cuántica esto es un área completamente nueva y sin investigación aún.
Al no tener aún respuesta a estas preguntas planteadas, se ha tratado de dar una herramienta que genere mutantes aleatorios en un intento de sustituir una aún por desarrollar que pudiera decidir cuáles son los mejores mutantes en cada caso.
Hay muchas líneas en las que desarrollar este trabajo futuro, ya que la computación cuántica, como hemos comentado, añade superposición, por lo que introducir mutantes puede dar lugar a resultados muy sorprendentes. Sin embargo, aunque uno se quede con muchas ganas de encontrar cuáles son los mejores mutantes, se trata de un trabajo verdaderamente costoso y que excede con creces los objetivos de este trabajo.

Bibliography

[1] V. Dorobantu, "The postulates of quantum mechanics," *arXiv preprint physics/0602145*, 2006.

[2] R. Feynman, "Feynman lectures on physics. vol. 3. quantum mechanics. le cours de physique de feynman. vol. 3. mecanique quantique," 1979.

[3] M. A. Nielsen and I. Chuang, "Quantum computation and quantum information," 2002.

[4] N. S. Yanofsky and M. A. Mannucci, *Quantum computing for computer scientists*. Cambridge University Press, 2008.

[5] A. Galindo and M. A. Martin-Delgado, "Information and computation: Classical and quantum aspects," *Reviews of Modern Physics*, vol. 74, no. 2, p. 347, 2002.

[6] M. M. Mano, C. R. Kime, and T. Martin, "Logic and computer design fundamentals," 2000.

[7] W. K. Wootters and W. H. Zurek, "The no-cloning theorem," *Physics Today*, vol. 62, no. 2, pp. 76–77, 2009.

[8] N. Gisin and H. Bechmann-Pasquinucci, "Bell inequality, bell states and maximally entangled states for n qubits," *Physics Letters A*, vol. 246, no. 1-2, pp. 1–6, 1998.

[9] S. Pirandola, J. Eisert, C. Weedbrook, A. Furusawa, and S. L. Braunstein, "Advances in quantum teleportation," *Nature photonics*, vol. 9, no. 10, pp. 641–652, 2015.

[10] R. P. Feynman, "Simulating physics with computers," in *Feynman and computation*, pp. 133–153, CRC Press, 2018.

[11] I. I. Manin, *Vychislimoe i nevychislimoe*. Sov. radio,, 1980.

[12] K. L. Brown, W. J. Munro, and V. M. Kendon, "Using quantum computers for quantum simulation," *Entropy*, vol. 12, no. 11, pp. 2268–2307, 2010.

[13] N. Johansson and J.-Å. Larsson, "Efficient classical simulation of the deutsch–jozsa and simon's algorithms," *Quantum Information Processing*, vol. 16, no. 9, pp. 1–14, 2017.

[14] D. David and J. Richard, "Rapid solution of problems by quantum computation," 1992.

[15] Z. Li, J. Dai, S. Pan, W. Zhang, and J. Hu, "Synthesis of deutsch-jozsa circuits and verification by ibm q," *International Journal of Theoretical Physics*, vol. 59, no. 6, pp. 1668–1678, 2020.

[16] D. R. Simon, "On the power of quantum computation," *SIAM Journal on Computing*, vol. 26, no. 5, pp. 1474–1483, 1997.

[17] L. L. Rui Abreu, João Paulo Fernandes and G. Tavares, "Metamorphic testing of oracle quantum programs," *Proceedings of The 3rd International Workshop on Quantum Software Engineering (Q-SE 2022) ACM, New York, NY, USA, 8 pages. https://doi.org/XXXXXXX.XXXXXXX*, 2022.

[18] A. Cherkas and S. Chivilikhin, "QUANTUM ADDER OF CLASSICAL NUMBERS," *Journal of Physics: Conference Series*, vol. 735, p. 012083, aug 2016.

[19] T. Y. Chen, F.-C. Kuo, H. Liu, P.-L. Poon, D. Towey, T. Tse, and Z. Q. Zhou, "Metamorphic testing: A review of challenges and opportunities," *ACM Computing Surveys (CSUR)*, vol. 51, no. 1, pp. 1–27, 2018.

[20] H. A. B. Ateya and S. M. Baneamoon, "Software bug prediction using static analysis with abstract syntax trees," *International Journal of Engineering and Artificial Intelligence*, vol. 1, no. 4, pp. 57–64, 2020.

[21] D. Huizinga and A. Kolawa, *Automated defect prevention: best practices in software management*. John Wiley & Sons, 2007.

[22] T. Ostrand, "White-box testing," *Encyclopedia of Software Engineering*, 2002.

[23] D. Huizinga and A. Kolawa, *Automated defect prevention: best practices in software management*. John Wiley & Sons, 2007.

[24] A. J. Offutt and R. H. Untch, "Mutation 2000: Uniting the orthogonal," *Mutation testing for the new century*, pp. 34–44, 2001.

[25] T. Chen, S. Cheung, and S. Yiu, "Metamorphic testing: a new approach for generating next test cases. technical report hkust-cs98-01," *Hong Kong Univ. of Science and Technology*, 1998.

[26] "Ieee standard glossary of software engineering terminology," *IEEE Std 610.12-1990*, pp. 1–84, 1990.

[27] S. Segura, D. Towey, Z. Q. Zhou, and T. Chen, "Metamorphic testing: Testing the untestable," *IEEE Software*, vol. PP, pp. 1–1, 12 2018.

[28] D. Gachechiladze, F. Lanubile, N. Novielli, and A. Serebrenik, "Anger and its direction in collaborative software development," in *2017 IEEE/ACM 39th International Conference on Software Engineering: New Ideas and Emerging Technologies Results Track (ICSE-NIER)*, pp. 11–14, IEEE, 2017.

[29] A. Cross, "The ibm q experience and qiskit open-source quantum computing software," in *APS March meeting abstracts*, vol. 2018, pp. L58–003, 2018.

[30]

[31] G. Carrascal, A. A. Del Barrio, and G. Botella, "First experiences of teaching quantum computing," *The Journal of Supercomputing*, vol. 77, no. 3, pp. 2770–2799, 2021.

[32] Q. D. Team, "Introduction to qiskit." https://qiskit.org/documentation/intro_tutorial1.html, 2021.

[33] Q. D. Team, "Circuit to gate." https://qiskit.org/documentation/stubs/qiskit.converters.circuit_to_gate.html, 2021.

[34] Q. D. Team, "Quantumcircuit.data." https://qiskit.org/documentation/stable/0.19/stubs/qiskit.circuit.QuantumCircuit.data.html, 2021.

Chapter 8

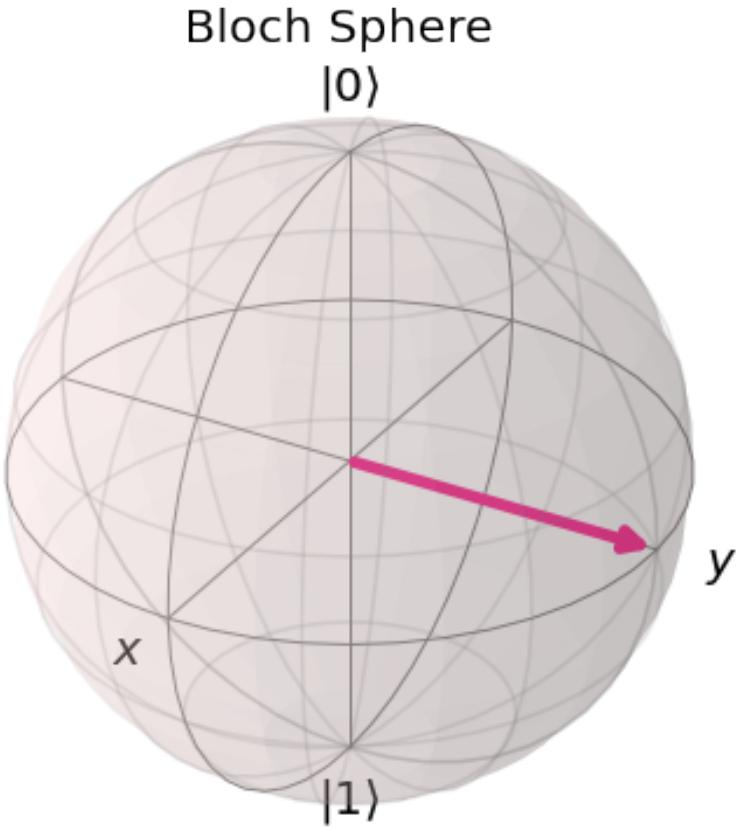
Anexo

8.1 Capítulo 3

Esfera de Bloch

[1]:
```
from qiskit.visualization import plot_bloch_vector
%matplotlib inline

plot_bloch_vector([0,1,0], title="Bloch Sphere")
```

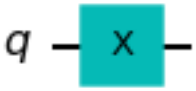
[1]:



8.1.1 3.1 Puertas cuánticas

[2]:
```
#Pauli-X
from qiskit import QuantumCircuit
from qiskit.tools.visualization import circuit_drawer

qc = QuantumCircuit(1)
qc.x(0)

#print(qc.draw(output='latex_source'))
qc.draw(output='mpl')
```

[2]:

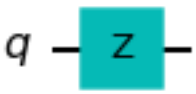


[4]:
```
#Pauli-Y
qc = QuantumCircuit(1)
qc.y(0)

#print(qc.draw(output='latex_source'))
qc.draw(output='mpl')
```

[4]:

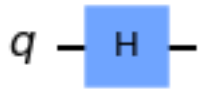
[5]:
```
#Pauli-Z
qc = QuantumCircuit(1)
qc.z(0)

#print(qc.draw(output='latex_source'))
qc.draw(output='mpl')
```

[5]:



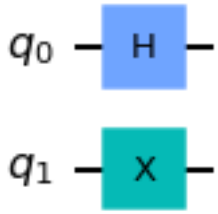
[6]:
```
#Hadamard
qc = QuantumCircuit(1)
qc.h(0)

#print(qc.draw(output='latex_source'))
qc.draw(output='mpl')
```

[6]:



[9]:
```
qc = QuantumCircuit(2)
qc.h(0)
qc.x(1)

#print(qc.draw(output='latex_source'))
qc.draw(output='mpl')
```

[9]:

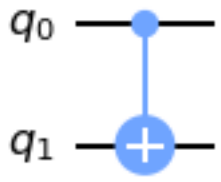


8.1.2 3.2 Puertas de varios qubits

CNOT

[14]:
```
qc = QuantumCircuit(2)
qc.cnot(0,1)

#print(qc.draw(output='latex_source'))
qc.draw(output='mpl')
```

[14]:



[15]:
```
#Vamos a simular la puerta CNOT para encontrar la matriz asociada


from qiskit import QuantumCircuit, Aer, assemble

import numpy as np
from qiskit.visualization import plot_histogram, plot_bloch_multivector

usim = Aer.get_backend('aer_simulator')
qc.save_unitary()
qobj = assemble(qc)
unitary = usim.run(qobj).result().get_unitary()
```

[16]:
```
from qiskit.visualization import array_to_latex
array_to_latex(unitary, prefix="\\text{Circuit = }\n")
```

[16]:

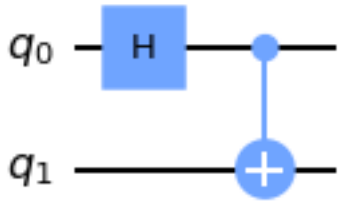
$$\text{Circuit =}$$

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix}$$

Circuito de Bell

[17]:
```
qc = QuantumCircuit(2)
qc.h(0)
qc.cnot(0,1)

#print(qc.draw(output='latex_source'))
qc.draw(output='mpl')
```

[17]:



CNOT circuit identity

[20]:
```
qc = QuantumCircuit(2)
qc.h(0)
qc.h(1)
qc.cx(0,1)
qc.h(0)
qc.h(1)
#print(qc.draw(output='latex_source'))
display(qc.draw())

qc.save_unitary()
usim = Aer.get_backend('aer_simulator')
qobj = assemble(qc)
unitary = usim.run(qobj).result().get_unitary()
array_to_latex(unitary, prefix="\\text{Circuit = }\n")
```

```
q_0:  H  H

q_1:  H  X  H
```

[20]:

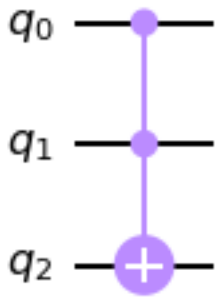
Circuit =

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix}$$

Toffoli gate

[21]:
```
qc=QuantumCircuit(3)
qc.ccx(0,1,2)
#print(qc.draw(output='latex_source'))
qc.draw(output='mpl')
```

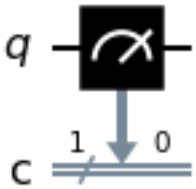
[21]:



Medición

[22]:
```
qc = QuantumCircuit(1,1)
qc.measure(0,0)
#print(qc.draw(output='latex_source'))
qc.draw('mpl')
```

[22]:

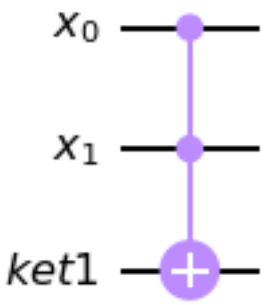


8.1.3 Circuitos cuánticos

NAND con Toffoli

[32]:
```
from qiskit import QuantumRegister

A = QuantumRegister(1,name='x_0')
B = QuantumRegister(1,name='x_1')
In = QuantumRegister(1,name='ket{1}')
qc = QuantumCircuit(A,B,In)
qc.ccx(A,B,In)
#print(qc.draw(output='latex_source'))
qc.draw(output='mpl')
```

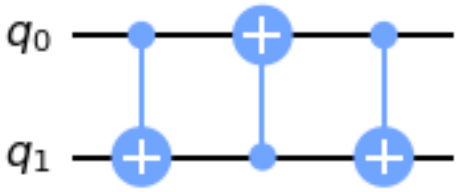
[32]:



Intercambiar qubits

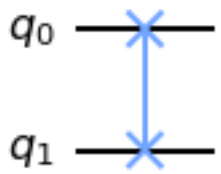
[31]:
```
qc=QuantumCircuit(2)
qc.cx(0,1)
qc.cx(1,0)
qc.cx(0,1)
#print(qc.draw(output='latex_source'))
qc.draw(output='mpl')
```

[31]:



[33]:
```
q = QuantumCircuit(2)
q.swap(0,1)
q.draw(output='mpl')
```

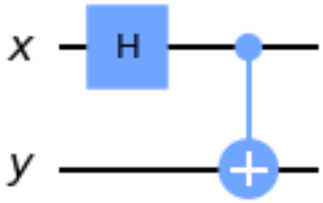
[33]:



Bell states

[36]:
```
#Circuito de Bell
x = QuantumRegister(1,name='x')
y = QuantumRegister(1,name='y')
qc = QuantumCircuit(x,y)
qc.h(0)
qc.cx(0,1)
#print(qc.draw(output='latex_source'))
qc.draw(output='mpl')
```

[36]:



Teleportación cuántica

[38]:
```
from qiskit import ClassicalRegister

psi = QuantumRegister(1,name='ket{\psi}')
beta = QuantumRegister(1,name='ket{\beta_{xy}^B')
medidor = ClassicalRegister(2,'medicion')
qc = QuantumCircuit(psi,beta,medidor)
qc.h(0)
qc.cx(0,1)
qc.measure([0,1],[0,1])
#print(qc.draw(output='latex_source'))
qc.draw()
```

[38]:
```
      ket{\psi}:  H M

ket{eta_{xy}^B:  X M

     medicion: 2/
                           0  1
```

[39]:
```
m1 = QuantumRegister(1,name='m1')
m2 = QuantumRegister(1,name='m2')
beta = QuantumRegister(1,name='ket{\beta_{xy}^B}')
qc = QuantumCircuit(m1,m2,beta)
qc.cx(0,2)
qc.cz(1,2)

#print(qc.draw(output='latex_source'))
qc.draw()
```

[39]:
```
              m1:

              m2:

ket{eta_{xy}^B}:  X
```

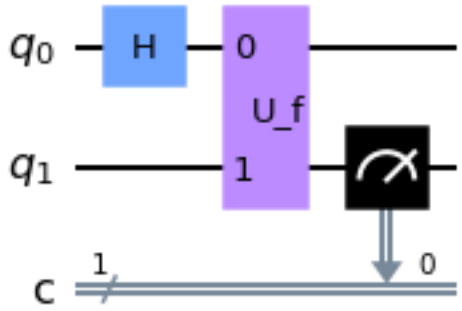
[ ]:
```

```

8.2 Capitulo 4

8.2.1 Algoritmo de Deutsch

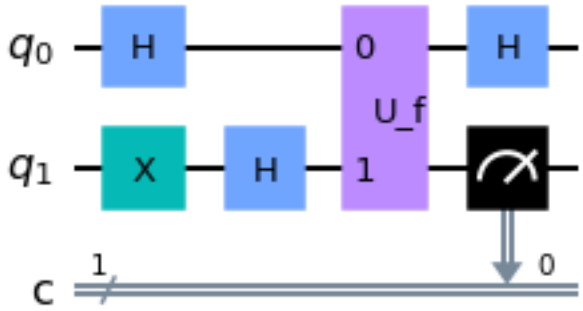
[3]:
```
#Paso 1
from qiskit import QuantumCircuit
from qiskit.circuit import Gate

U_f = Gate(name='U_f', num_qubits=2, params=[])

qc = QuantumCircuit(2,1)
qc.h(0)
qc.append(U_f,[0,1])
qc.measure(1,0)

#print(qc.draw(output='latex_source'))
qc.draw(output='mpl')
```

[3]:



[5]:
```
# Algoritmo para U_f
qc = QuantumCircuit(2,1)
qc.h(0)
qc.x(1)
qc.h(1)
qc.append(U_f,[0,1])
qc.h(0)
qc.measure(1,0)

#print(qc.draw(output='latex_source'))
qc.draw(output='mpl')
```

[5]:



Algoritmo

[6]:
```
#Algoritmo de Deutsch
def Deutsch_Algorithm(oracle):
    """

    Parameters
```

```
    ----------
    f : Circuito cuántico U_f, dos qubits de entrada y dos qubits de salida

    Returns
    -------
    qc circuito cuántico correspondiente al algoritmo de Deutsch

    """
    qc = QuantumCircuit(2,1)
    qc.h(0)
    qc.x(1)
    qc.h(1)
    qc.append(oracle,[0,1])
    qc.h(0)
    qc.measure(0,0)
    return qc
```

[8]:
```
import numpy as np
from qiskit.providers.aer import QasmSimulator
from qiskit import transpile
simulator = QasmSimulator()

n = 1

const_oracle = QuantumCircuit(n+1)

output = np.random.randint(2)
if output == 1:
    const_oracle.x(n)

const_oracle.draw()

qc = Deutsch_Algorithm(const_oracle)

compiled_circuit = transpile(qc, simulator)

# Execute the circuit on the qasm simulator
job = simulator.run(compiled_circuit, shots=1000)

# Grab results from the job
result = job.result()

# Returns counts
counts = result.get_counts(compiled_circuit)
print("\nTotal count for 00 and 11 are:",counts)

# Draw the circuit
qc.draw()
```

```
Total count for 00 and 11 are: {'0': 1000}
```

[8]:
```
q_0:  H 0           H M
```

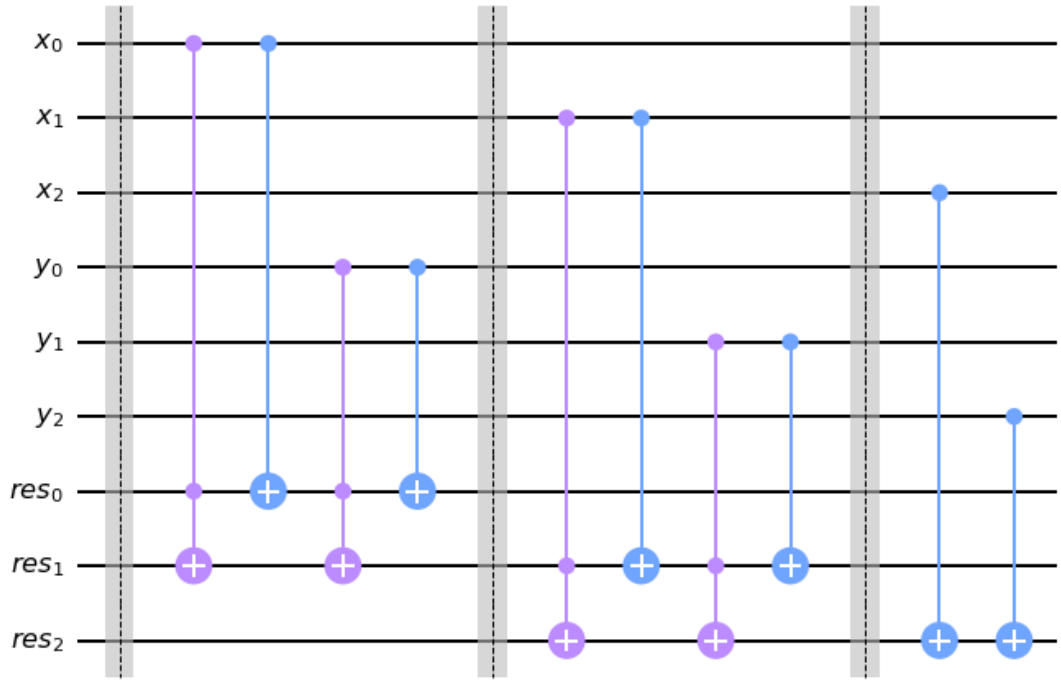
```
       circuit-7
q_1:  X  H 1

c: 1/
                                  0
```

8.2.2 Algoritmo suma modular

Suma modular para números de 3 bits

[11]:
```
from qiskit import QuantumRegister

x_0 = QuantumRegister(1, name='x_0')
x_1 = QuantumRegister(1, name='x_1')
x_2 = QuantumRegister(1, name='x_2')
y_0 = QuantumRegister(1, name='y_0')
y_1 = QuantumRegister(1, name='y_1')
y_2 = QuantumRegister(1, name='y_2')
res_0 = QuantumRegister(1, name='res_0')
res_1 = QuantumRegister(1, name='res_1')
res_2 = QuantumRegister(1, name='res_2')

qc = QuantumCircuit(x_0,x_1,x_2,y_0,y_1,y_2,res_0,res_1,res_2)
qc.barrier()
qc.ccx(0,6,7)
qc.cx(0,6)
qc.ccx(3,6,7)
qc.cx(3,6)
qc.barrier()
qc.ccx(1,7,8)
qc.cx(1,7)
qc.ccx(4,7,8)
qc.cx(4,7)
qc.barrier()
qc.cx(2,8)
qc.cx(5,8)
#print(qc.draw(output='latex_source'))
qc.draw(output='mpl')
```

[11]:



8.2.3 Algoritmo Deutsch-Jozsa

[24]:
```
#Pendiente
```

[ ]:
```
```

Algoritmo suma modular n bits

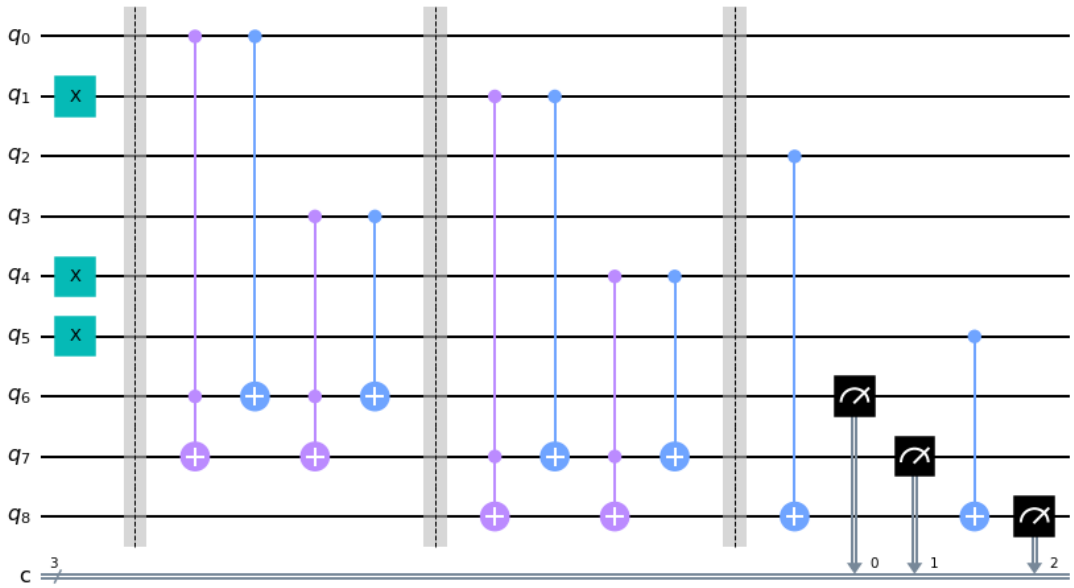
[12]:
```
def circuito_suma(n):
    qc = QuantumCircuit(3*n,n)
    for i in range(n-1):
        qc.barrier()
        qc.ccx(i,i+2*n,i+2*n+1)
        qc.cx(i,i+2*n)
        qc.ccx(i+n,i+2*n,i+2*n+1)
        qc.cx(i+n,i+2*n)
    qc.barrier()
    qc.cx(n-1,3*n-1)
    qc.cx(2*n-1,3*n-1)
    for i in range(n):
        qc.measure(2*n+i,i)
    return qc
```

[14]:
```
def suma_bits(n,st1,st2):
    qc = QuantumCircuit(3*n,n)
    i = n-1
    for c in st1:
```

```
        if c=='1':
            qc.x(i)
        i-=1
    i=2*n-1
    for c in st2:
        if c=='1':
            qc.x(i)
        i-=1
    result = qc.compose(circuito_suma(n))
    return result
```

[15]:
```
qc = suma_bits(3,'010','110')
qc.draw(output='mpl')
```

[15]:



[21]:
```
#Si queremos un programa que nos diga directamente el resultado:

def suma_modular(a,b):
    aux = max(a,b)
    m = min(a,b)
    a=aux
    b=m
    a = "{0:b}".format(a)
    b = "{0:b}".format(b)
    if (len(a)>len(b)):
        r = len(a)-len(b)
        st = '0'*r
        b=st+b
    qc = suma_bits(len(a),a,b)
    compiled_circuit = transpile(qc, simulator)
    job = simulator.run(compiled_circuit)
```

```
    result = job.result()
    counts = result.get_counts(compiled_circuit)
    print(qc.draw())
    print('El resultado de la suma en binario de ',a,'+',b, 'es: ',counts)
    return qc
```

[22]:
```
suma_modular(3,3)
suma_modular(3,4)
suma_modular(1,5)
```

```
q_0:  X

q_1:  X

q_2:  X

q_3:  X

q_4:  X  X M

q_5:  X  X  X  X M

c: 2/
                                          0       1
El resultado de la suma en binario de  11 + 11 es:  {'10': 1024}

q_0:

q_1:

q_2:  X

q_3:  X

q_4:  X

q_5:

q_6:  X  X M

q_7:  X  X  X  X M

q_8:  X  X  X  X M

c: 3/
                                                                0  1       2
El resultado de la suma en binario de  100 + 011 es:  {'111': 1024}

q_0:  X

q_1:

q_2:  X
```

```
q_3:  X

q_4:

q_5:

q_6:  X  X M

q_7:  X  X  X  X M

q_8:  X  X  X  X M

c: 3/
                                                                    0  1       2
El resultado de la suma en binario de  101 + 001 es:  {'110': 1024}
```

[22]:
```
<qiskit.circuit.quantumcircuit.QuantumCircuit at 0x7fb6ff96e160>
```

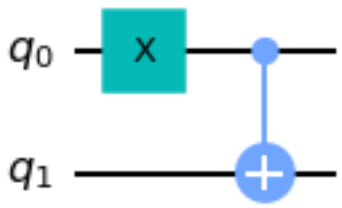
[ ]:

8.3 Operadores de mutación

8.3.1 Mutantes a mano

[3]:
```
from qiskit import QuantumCircuit

qc = QuantumCircuit(2)
qc.x(0)
qc.cx(0,1)
#print(qc.draw(output='latex_source'))
qc.draw(output='mpl')
```

[3]:



[6]:
```
from qiskit import QuantumCircuit, Aer, assemble

import numpy as np
from qiskit.visualization import plot_histogram, plot_bloch_multivector

svsim = Aer.get_backend('aer_simulator')
qc.save_statevector()
```

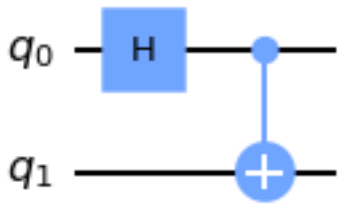
```
qobj = assemble(qc)
final_state = svsim.run(qobj).result().get_statevector()

# In Jupyter Notebooks we can display this nicely using Latex.
# If not using Jupyter Notebooks you may need to remove the
# array_to_latex function and use print(final_state) instead.
from qiskit.visualization import array_to_latex
array_to_latex(final_state, prefix="\\text{Statevector} = ")
```

[6]:

$$\text{Statevector} = \begin{bmatrix} 0 & 0 & 0 & 1 \end{bmatrix}$$

[7]:
```
# Aplicar operador de mutación que cambie la puerta x por la puerta h
qc = QuantumCircuit(2)
qc.h(0)
qc.cx(0,1)
#print(qc.draw(output='latex_source'))
qc.draw(output='mpl')
```

[7]:



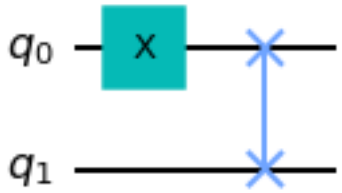
[8]:
```
svsim = Aer.get_backend('aer_simulator')
qc.save_statevector()
qobj = assemble(qc)
final_state = svsim.run(qobj).result().get_statevector()

# In Jupyter Notebooks we can display this nicely using Latex.
# If not using Jupyter Notebooks you may need to remove the
# array_to_latex function and use print(final_state) instead.
from qiskit.visualization import array_to_latex
array_to_latex(final_state, prefix="\\text{Statevector} = ")
```

[8]:

$$\text{Statevector} = \begin{bmatrix} \frac{1}{\sqrt{2}} & 0 & 0 & \frac{1}{\sqrt{2}} \end{bmatrix}$$

[9]:
```
# Aplicar operador de mutación que cambie la puerta cnot por la puerta swap
qc = QuantumCircuit(2)
qc.x(0)
qc.swap(0,1)
#print(qc.draw(output='latex_source'))
qc.draw(output='mpl')
```

[9]:



[10]:
```
svsim = Aer.get_backend('aer_simulator')
qc.save_statevector()
qobj = assemble(qc)
final_state = svsim.run(qobj).result().get_statevector()

# In Jupyter Notebooks we can display this nicely using Latex.
# If not using Jupyter Notebooks you may need to remove the
# array_to_latex function and use print(final_state) instead.
from qiskit.visualization import array_to_latex
array_to_latex(final_state, prefix="\\text{Statevector} = ")
```

[10]:

$$\text{Statevector} = \begin{bmatrix} 0 & 0 & 1 & 0 \end{bmatrix}$$

8.4 Operadores de mutación en qiskit

Importamos las puertas que usa qiskit

[12]:
```
from qiskit.circuit.library import HGate, SwapGate
from qiskit.circuit.library.standard_gates import (
    IGate,
    U1Gate,
    U2Gate,
    U3Gate,
    XGate,
    YGate,
    ZGate,
    HGate,
    SGate,
    SdgGate,
    TGate,
    TdgGate,
    RXGate,
    RYGate,
    RZGate,
    CXGate,
    CYGate,
    CZGate,
    CHGate,
    CRZGate,
```

```
    CU1Gate,
    CU3Gate,
    SwapGate,
    RZZGate,
    CCXGate,
    CSwapGate,
)
#Primer tipo de operador, cambiar puertas consecutivas de orden, cambiando tambien el␣
 ↪circuito

one_q_ops = [
        IGate,
        U1Gate,
        U2Gate,
        U3Gate,
        XGate,
        YGate,
        ZGate,
        HGate,
        SGate,
        SdgGate,
        TGate,
        TdgGate,
        RXGate,
        RYGate,
        RZGate,
    ]
one_param = [U1Gate, RXGate, RYGate, RZGate, RZZGate, CU1Gate, CRZGate]
two_param = [U2Gate]
three_param = [U3Gate, CU3Gate]
two_q_ops = [CXGate, CYGate, CZGate, CHGate, CRZGate, CU1Gate, CU3Gate, SwapGate,␣
 ↪RZZGate]
three_q_ops = [CCXGate, CSwapGate]
```

Funciones auxiliares

[14]:
```
from qiskit.circuit import Instruction
import random
import numpy as np
from qiskit import QuantumCircuit, transpile
from qiskit.providers.aer import QasmSimulator
from qiskit.visualization import plot_histogram




def change(ls,q1,q2):
    if q1 in ls:
        index = ls.index(q1)
        ls[index]=q2
    return ls
```

```
def replace(circ, origin, instruction: Instruction):
    circ._data = [(instruction, _inst[1], _inst[2]) if _inst[0] == origin  else _inst␣
 ↪for _inst in circ._data]
def replace_target_qubit(circ,origin,target_qubit,final_qubit):
    circ._data = [(_inst[0], change(_inst[1],target_qubit,final_qubit), _inst[2]) if␣
 ↪_inst[0] == origin  else _inst for _inst in circ._data]
def replace_target_clbit(circ,origin,target_clbit,final_clbit):
    circ._data = [(_inst[0], _inst[1], change(_inst[2],target_clbit,final_clbit)) if␣
 ↪_inst[0] == origin  else _inst for _inst in circ._data]
```

Operador 1: gate_mutant

[19]:
```
def gate_mutant(circ1,input_gate=None,output_gate=None):
    """



    Parameters
    ----------
    circ1 : QuantumCircuit
    input_gate : Instruction
    output_gate : Instruction

    Description
    -----------
    Intercambia input_gate por output_gate en circ1, si no se especifica alguna de las␣
 ↪dos, se toma de manera aleatoria.

    Returns
    -------
    circ : QuantumCircuit
        Mutant

    """
    circ = circ1.copy()
    if input_gate is None:
        n = len(circ.data)
        i=0
        num_clbits=-1
        while num_clbits!=0 and i<n:
            r = random.randint(0, n-1)
            inst = circ.data[r]
            input_gate=inst[0]
            num_clbits = input_gate.num_clbits
            i+=1
    if output_gate is None:
        """
        Si no le decimos lo contrario, cambia puertas que afectan a la misma cantidad␣
 ↪de bits
        """
        num_qubits=input_gate.num_qubits
        if num_qubits==1:
            GATE = one_q_ops[random.randint(0, len(one_q_ops)-1)]
```

```
        elif num_qubits==2:
            GATE = two_q_ops[random.randint(0, len(two_q_ops)-1)]
        elif num_qubits==3:
            GATE = three_q_ops[random.randint(0, len(three_q_ops)-1)]
        if GATE in one_param:
            p = random.uniform(0, 2*np.pi)
            output_gate = GATE(p)
        elif GATE in two_param:
            p1 = random.uniform(0, 2*np.pi)
            p2 = random.uniform(0, 2*np.pi)
            output_gate = GATE(p1,p2)
        elif GATE in three_param:
            p1=random.uniform(0, 2*np.pi)
            p2=random.uniform(0, 2*np.pi)
            p3=random.uniform(0, 2*np.pi)
            output_gate = GATE(p1,p2,p3)
        else:
            output_gate = GATE()
    replace(circ,input_gate,output_gate)
    print('Se reemplaza la puerta ',input_gate.name,' por la puerta ',output_gate.name)
    return circ
```

Operador 2: targetqubit_mutant

[21]:
```
def targetqubit_mutant(circ1,gate=None,target_qubit=None,final_qubit=None):
    """


    Parameters
    ----------
    circ1 : QuantumCircuit
    gate : Instruction
    target_qubit : Qubit
    final_qubit : Quibit

    Description
    -----------
    Intercambia, en gate, target_qubit por final_qubit. Los parámetros no␣
 ↪especificados se seleccionan de manera aleatoria

    Returns
    -------
    circ : QuantumCircuit
        Mutant

    """
    circ = circ1.copy()
    if gate is None:
        n = len(circ.qubits)
        if n<3:
            raise Exception('Impossible to change target qubits for any gate as there␣
 ↪are less than 3 qubits')
        else:
```

```
            i=0
            num_qubits=0
            while num_qubits<2 and i<n:
                r = random.randint(0, n-1)
                inst = circ.data[r]
                num_qubits = inst[0].num_qubits
                i+=1
                gate=inst[0]
    if gate.num_qubits<2:
        raise Exception('Could not find multi qubit gates, change the circuit or try␣
↪again')
    else:
        if target_qubit is None:
            target_qubit = inst[1][random.randint(0, len(inst[1])-1)]
        if final_qubit is None:
            final_qubit = circ.qubits[random.randint(0,len(circ.qubits)-1)]
        replace_target_qubit(circ, gate, target_qubit, final_qubit)
    print('Se han cambiado los qubits a los que afecta la puerta ',gate.name)
    return circ
```

Operador 3: targetclbit_mutant

[22]:
```
def targetclbit_mutant(circ1,gate=None,target_clbit=None,final_clbit=None):
    """


    Parameters
    ----------
    circ1 : QuantumCircuit
    gate : Instruction
    target_clbit : Clbit
    final_clbit : Clbit


    Description
    -----------
    Intercambia, en gate, target_clbit por final_clbit. Los parámetros no␣
↪especificados se seleccionan de manera aleatoria


    Returns
    -------
    circ : QuantumCircuit
        Mutant

    """
    circ = circ1.copy()
    if gate is None:
        n = len(circ.data)
        if len(circ.clbits)<2:
            raise Exception('Impossible to change target clbits as there are less than␣
↪2')
```

```
        else:
            i=0
            num_clbits=0
            while num_clbits==0 and i<n:
                r = random.randint(0, n-1)
                inst = circ.data[r]
                num_clbits = inst[0].num_clbits
                i+=1
                gate=inst[0]
    if gate.num_clbits==0:
        raise Exception('Could not find gates with classical bits or meaurements,␣
↪change the circuit or try again')
    else:
        if target_clbit is None:
            target_clbit = inst[2][random.randint(0, len(inst[2])-1)]
        if final_clbit is None:
            final_clbit = circ.clbits[random.randint(0,len(circ.clbits)-1)]
        replace_target_clbit(circ, gate, target_clbit, final_clbit)
    print('Se ha cambiado los bits clásicos afectados')
    return circ
```

Operador 4: measure_mutant

[23]:
```
def measure_mutant(circ1,target_qubit=None,final_qubit=None):
    """


    Parameters
    ----------
    circ1 : TQuantumCircuit
    target_qubit : Qubit
    final_qubit : Qubit

    Description
    -----------
    Intercambia, en una medición, target_qubit por final_qubit. Los parámetros no␣
↪especificados se seleccionan de manera aleatoria


    Returns
    -------
    circ : QuantumCircuit
        Mutant
    """
    circ=circ1.copy()
    n=len(circ.data)
    if len(circ.clbits)<2:
        raise Exception('Impossible to change target clbits as there are less than 2')
    if target_qubit is None:
        i=0
        found=False
        while i<n and not found:
            r = random.randint(0, n-1)
```

```
            inst = circ.data[r]
            if inst[0].name=='measure':
                found=True
            i+=1
            gate=inst[0]
        target_qubit=inst[1][0]
    else:
        found=False
        i=0
        while not found and i<n:
            inst = circ.data[i]
            if inst[0]=='measure' and target_qubit in inst[1]:
                found=True
            gate=inst[0]
            i+=1
    if gate.name != 'measure':
        raise Exception('Could not find measurement gate, change the circuit or try␣
↪again')
    if final_qubit is None:
        final_qubit = circ.qubits[random.randint(0,len(circ.qubits)-1)]
    print('Se cambia el qubit al que afecta la medición')
    replace_target_qubit(circ, gate, target_qubit, final_qubit)
    return circ
```

Operador de operadores

[24]:
```
def mutant(circ,gate=True,target_qubit=True,target_clbit=True,measure=True):
    """


    Parameters
    ----------
    circ : QuantumCircuit
    gate : bool
    target_qubit : bool
    target_clbit : bool
    measure : bool
        Indica si se quiere dar la opción de usar al operador measure


    Description
    -----------
    Garantiza que devuelve un mutante del circuito original de manera aleatoria␣
↪seleccionando unos de los cuatro operadores, si se indica que alguno de los tres es␣
↪falso, no se generará el mutante usando ese operador


    Returns
    -------
    result : QuantumCircuit
        Mutant
```

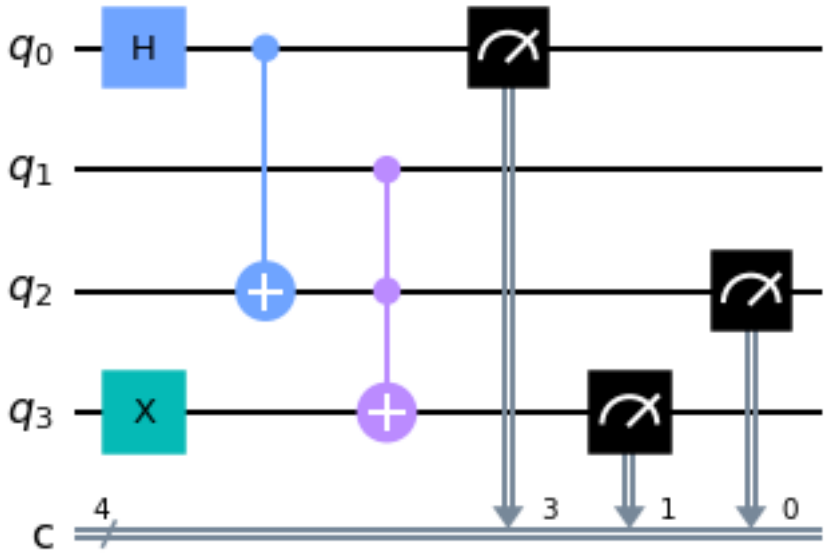
```
    """

    r = random.randint(1, 4)
    if r==1 and gate:
        result = gate_mutant(circ)
    elif r==2 and target_qubit:
        result = targetqubit_mutant(circ)
    elif r==3 and target_clbit:
        result = targetclbit_mutant(circ)
    elif measure:
        result = measure_mutant(circ)
    if result == circ:
        result = mutant(circ,gate,target_qubit,target_clbit,measure)
    return result
```

[25]:
```
# Ahora hacemos que sean métodos para QuantumCircuit
QuantumCircuit.mutant = mutant
QuantumCircuit.gate_mutant = gate_mutant
QuantumCircuit.targetqubit_mutant = targetqubit_mutant
QuantumCircuit.targetclbit_mutant = targetclbit_mutant
QuantumCircuit.measure_mutant = measure_mutant
```

8.5 Ejemplos de uso

[28]:
```
#Creamos un circuito para mutarlo
qc = QuantumCircuit(4,4)
qc.h(0)
qc.x(3)
qc.cx(0,2)
qc.ccx(1,2,3)
qc.measure([0,3,2],[3,1,0])
#print(qc.draw(output='latex_source'))
qc.draw(output='mpl')
```

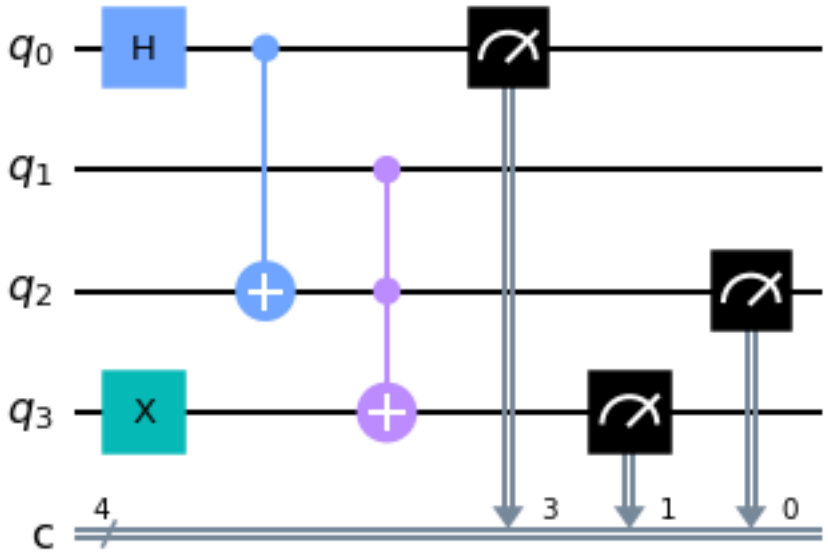
[28]:



[29]:
```
# Operador 1
qc.gate_mutant()#Esto nos devolverá el cambio de cualquier puerta por cualquier otra

#Cambiamos particularmente la puerta H por la puerta X
qc.gate_mutant(HGate(),XGate())
qc.draw(output='mpl')
```

```
Se reemplaza la puerta  ccx  por la puerta  ccx
Se reemplaza la puerta  h  por la puerta  x
```

[29]:



[30]:
```
# Por ejemplo, podemos crear una lista de mutantes dada una lista de puertas
puertas = [XGate(),SdgGate(),ZGate(),YGate()]
mutantes = [qc.gate_mutant(HGate(),puerta) for puerta in puertas]
mutantes
```

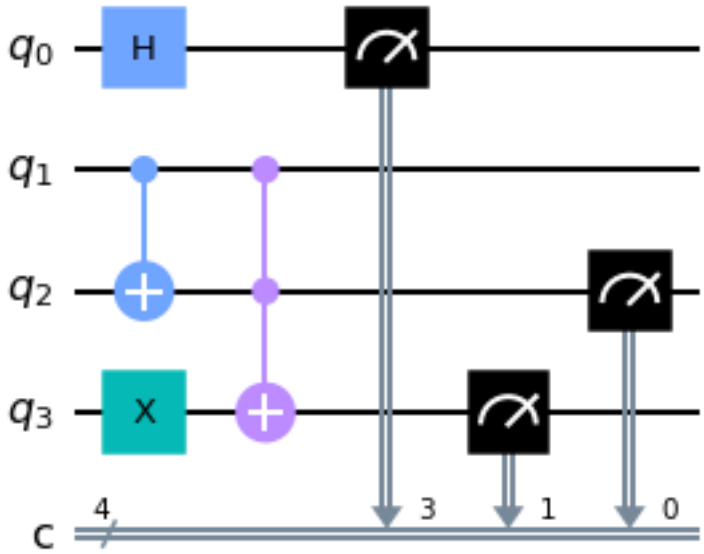
```
Se reemplaza la puerta  h  por la puerta  x
Se reemplaza la puerta  h  por la puerta  sdg
Se reemplaza la puerta  h  por la puerta  z
Se reemplaza la puerta  h  por la puerta  y
```

[30]:
```
[<qiskit.circuit.quantumcircuit.QuantumCircuit at 0x7f0b0c3c0a30>,
 <qiskit.circuit.quantumcircuit.QuantumCircuit at 0x7f0b0c42c8e0>,
 <qiskit.circuit.quantumcircuit.QuantumCircuit at 0x7f0b0c555940>,
 <qiskit.circuit.quantumcircuit.QuantumCircuit at 0x7f0b0c3c02e0>]
```

[31]:
```
qc.targetqubit_mutant().draw(output='mpl')
```

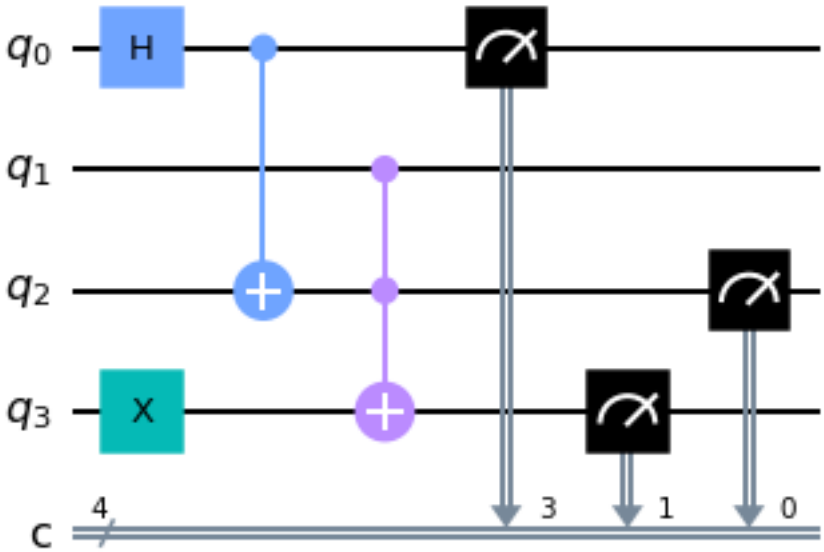
```
Se han cambiado los qubits a los que afecta la puerta  cx
```

[31]:



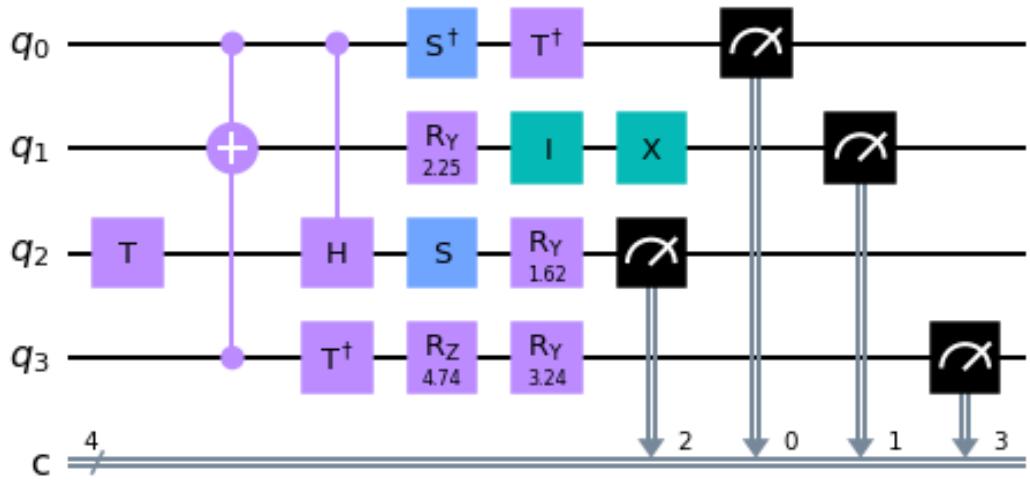
[32]:
```
qc.draw(output='mpl')
#El anterior mutante ha cambiado el qubit de control en la puerta cx
```

[32]:



[33]:
```
from qiskit.circuit.random import random_circuit

circ = random_circuit(4, 4, measure=True)
circ.draw(output='mpl')
```

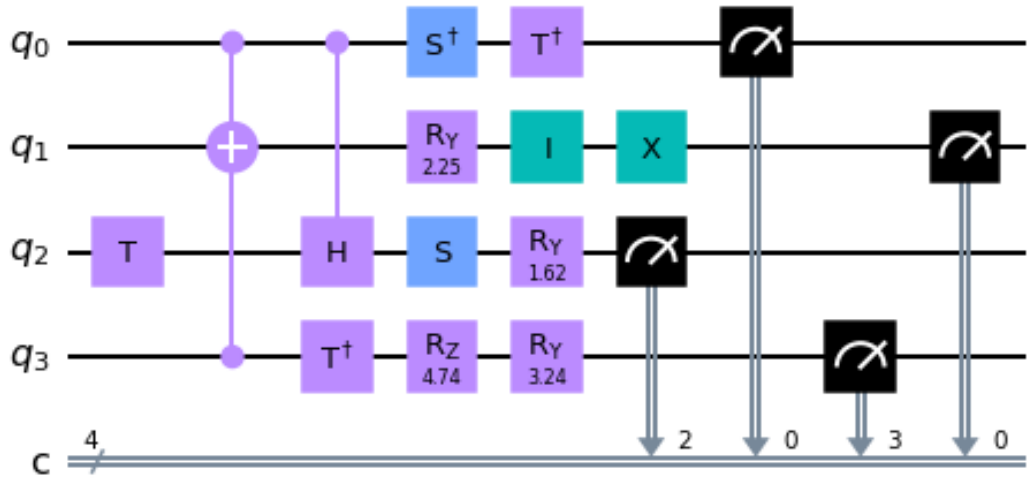
[33]:



[34]:
```
c1 = circ.mutant()
#print(c1.draw(output='latex_source'))
c1.draw(output='mpl')
```

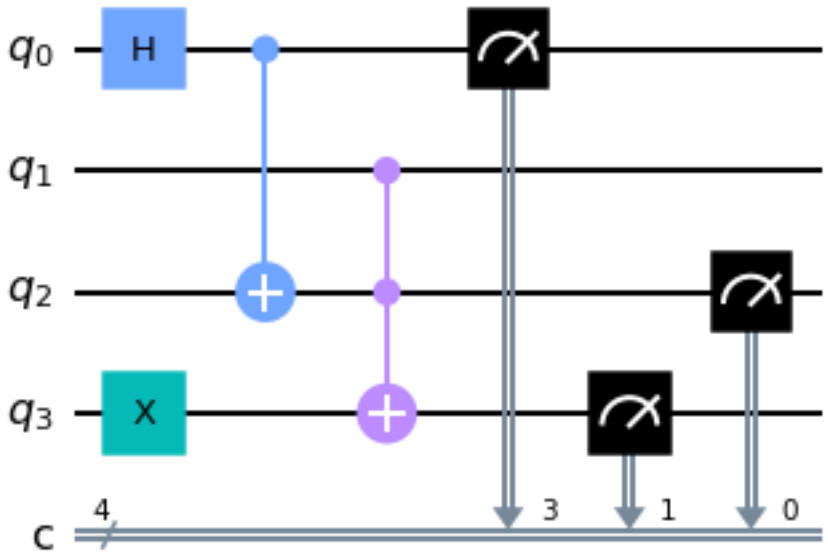
Se ha cambiado los bits clásicos afectados

[34]:



[35]:
```
#Creamos un circuito para mutarlo
qc = QuantumCircuit(4,4)
qc.h(0)
qc.x(3)
qc.cx(0,2)
qc.ccx(1,2,3)
qc.measure([0,3,2],[3,1,0])
#print(qc.draw(output='latex_source'))
qc.draw(output='mpl')
```

[35]:

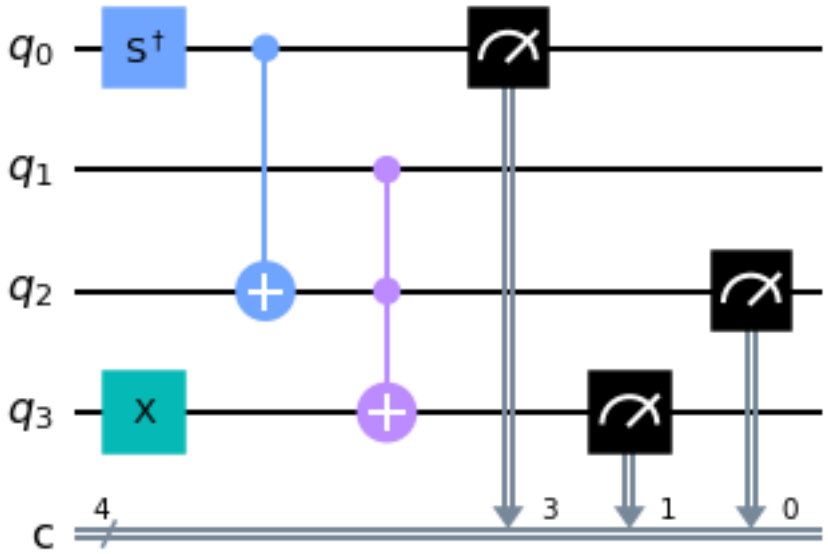


[36]:
```
#Ahora creamos tres mutantes del circuito
m1 = qc.gate_mutant(HGate(),SdgGate())
m2 = qc.targetqubit_mutant(CXGate(),qc.qubits[0],qc.qubits[1])
m3 = qc.measure_mutant(qc.qubits[0],qc.qubits[1])
```

```
Se reemplaza la puerta  h  por la puerta  sdg
Se han cambiado los qubits a los que afecta la puerta  cx
Se cambia el qubit al que afecta la medición
```

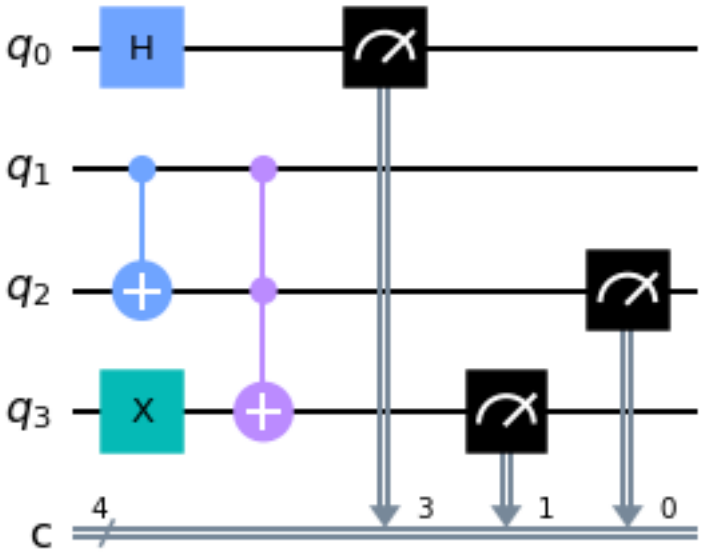
[37]:
```
#print(m1.draw(output='latex_source'))
m1.draw(output='mpl')
```

[37]:



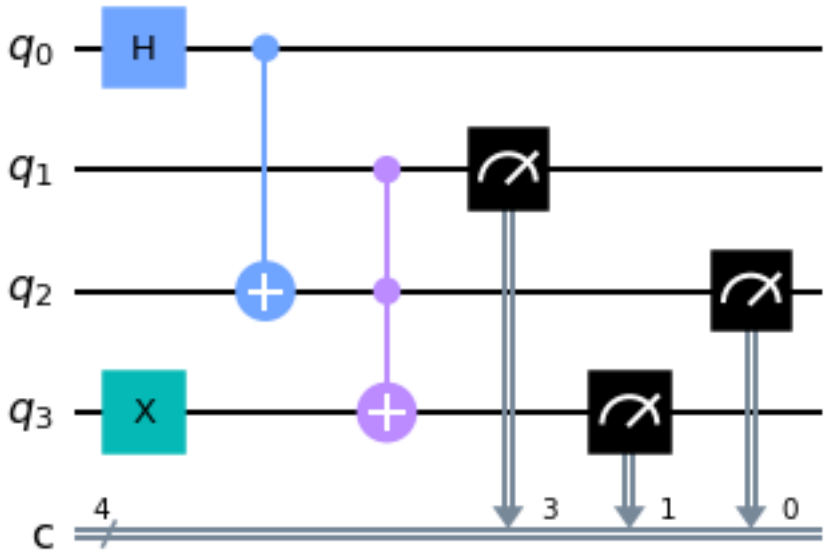
[39]:
```
#print(m2.draw(output='latex_source'))
m2.draw(output='mpl')
```

[39]:



[41]:
```
#print(m3.draw(output='latex_source'))
m3.draw(output='mpl')
```

[41]:



8.6 Ejemplo de metamorphic testing

[46]:
```
from qiskit import QuantumCircuit, transpile
from qiskit.providers.aer import QasmSimulator
from qiskit.visualization import plot_histogram

simulator = QasmSimulator()



def circuito_suma(n):
    qc = QuantumCircuit(3*n,n)
    for i in range(n-1):
        qc.barrier()
        qc.ccx(i,i+2*n,i+2*n+1)
        qc.cx(i,i+2*n)
        qc.ccx(i+n,i+2*n,i+2*n+1)
        qc.cx(i+n,i+2*n)
    qc.barrier()
    qc.cx(n-1,3*n-1)
    qc.cx(2*n-1,3*n-1)
    for i in range(n):
        qc.measure(2*n+i,i)
    return qc
```

```
def suma_bits(n,st1,st2):
    qc = QuantumCircuit(3*n,n)
    i = n-1
    for c in st1:
        if c=='1':
            qc.x(i)
        i-=1
    i=2*n-1
    for c in st2:
        if c=='1':
            qc.x(i)
        i-=1
    result = qc.compose(circuito_suma(n))
    return result
def suma_modular(a,b):
    aux = max(a,b)
    m = min(a,b)
    a=aux
    b=m
    a = "{0:b}".format(a)
    b = "{0:b}".format(b)
    if (len(a)>len(b)):
        r = len(a)-len(b)
        st = '0'*r
        b=st+b
    qc = suma_bits(len(a),a,b)
    compiled_circuit = transpile(qc, simulator)
    job = simulator.run(compiled_circuit)
    result = job.result()
    counts = result.get_counts(compiled_circuit)
    print(qc.draw())
    print('El resultado de la suma en binario de ',a,'+',b, 'es: ',counts)
    return qc
```

[47]:
```
#Ejemplo de la regla metamórfica
qc=suma_bits(5,'10101','10100')
compiled_circuit = transpile(qc, simulator)
job = simulator.run(compiled_circuit,shots=10)
result = job.result()
counts1 = result.get_counts(compiled_circuit)
print(counts1)

qc=suma_bits(5,'10100','10101')
compiled_circuit = transpile(qc, simulator)
job = simulator.run(compiled_circuit,shots=10)
result = job.result()
counts2 = result.get_counts(compiled_circuit)
print(counts2)
counts1 == counts2
```

```
{'01001': 10}
{'01001': 10}
```

[47]:
```
True
```

[48]:
```
import random
# Usaremos esta función para crear un conjunto test
def rand_key(p):

    # Variable to store the
    # string
    key1 = ""

    # Loop to find the string
    # of desired length
    for i in range(p):

        # randint function to generate
        # 0, 1 randomly and converting
        # the result into str
        temp = str(random.randint(0, 1))

        # Concatenation the random 0, 1
        # to the final result
        key1 += temp

    return(key1)

n = 5
test = [rand_key(n) for i in range(100)]
```

[49]:
```
import functools
import operator


l=[None]*(len(test)-1)
for i in range(len(test)-1):

    qc1 = suma_bits(n,test[i],test[i+1])
    qc2 = suma_bits(n,test[i+1],test[i])

    compiled_circuit = transpile(qc1, simulator)
    job = simulator.run(compiled_circuit,shots=10)
    result = job.result()
    counts1 = result.get_counts(compiled_circuit)


    compiled_circuit = transpile(qc2, simulator)
    job = simulator.run(compiled_circuit,shots=10)
    result = job.result()
    counts2 = result.get_counts(compiled_circuit)


    l[i]=counts1==counts2

if not (False in l):
    print('Test superado con éxito')
```

```
Test superado con éxito
```

[ ]: