Escuela Superior
de Ingeniería y Tecnología
Universidad de La Laguna

Trabajo de Fin de Grado

QuantumSolver: Librería para el desarrollo cuántico

QuantumSolver: Toolset for quantum development

José Daniel Escánez Expósito

La Laguna, 13 de junio de 2022

Dña. **Pino Caballero-Gil**, con N.I.F. 45.534.310-Z, Catedrática de Universidad adscrita al Departamento de Ingeniería Informática y de Sistemas de la Universidad de La Laguna, como tutora

D. **Francisco Martín-Fernández**, con N.I.F. 78.629.638-K, Miembro del Personal Técnico Superior de IBM Research (IBM Quantum), como cotutor

C E R T I F I C A N

Que la presente memoria titulada:

"*QuantumSolver: Librería para el desarrollo cuántico*"

ha sido realizada bajo su dirección por D. **José Daniel Escánez-Expósito**, con N.I.F. 79.159.491-T.

Y para que así conste, en cumplimiento de la legislación vigente y a los efectos oportunos firman la presente en La Laguna a 13 de junio de 2022

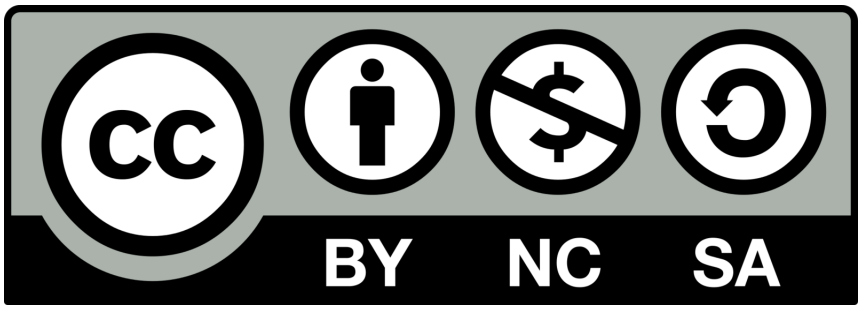
Agradecimientos

A mis tutores, Pino y Paco, por la dedicación y cercanía con la que me han ayudado a solventar las dificultades teóricas y prácticas surgidas.

A mis familiares y a mis amigos, por todo el incondicional apoyo e ilimitada paciencia que me han brindado durante el transcurso del Grado de Ingeniería Informática y en el desarrollo de este trabajo.

Y, especialmente, a mi madre, a mi hermana, a Ana y a Yaret, por ser pilares fundamentales en mi vida y motivarme en la consecución de todas mis metas y objetivos.

Licencia



Resumen

En este trabajo se expone el desarrollo de una librería cuántica llamada QuantumSolver, implementada bajo la licencia de software de código abierto MIT. Dicha herramienta incluye varios algoritmos cuánticos encapsulados en una sencilla y ampliable estructura de componentes. Los algoritmos recogidos por la librería son: La generación de números aleatorios, la resolución de los problemas de Deutsch-Jozsa y Bernstein-Vazirani, el algoritmo de Grover, el teletransporte cuántico, el protocolo de codificación superdensa, y el protocolo criptográfico cuántico BB84. Se describen aquí los principales detalles de la implementación del toolset, así como las interfaces desarrolladas y las conclusiones obtenidas de la investigación realizada sobre las funcionalidades conseguidas.

Palabras clave: Computación cuántica, Qiskit, Librería cuántica, Generación de números aleatorios, Algoritmo de Deutsch-Jozsa, Algoritmo de Bernstein-Vazirani, Algoritmo de Grover, Teletransporte Cuántico, Codificación superdensa, Criptografía cuántica, Protocolo BB84

Abstract

This document presents the development of a quantum library called QuantumSolver, implemented under the open-source MIT license. This tool includes several quantum algorithms encapsulated in a simple and scalable component structure. The algorithms collected by the library are: Random number generation, solving the Deutsch-Jozsa and Bernstein-Vazirani problems, Grover's algorithm, quantum teleportation, superdense coding protocol, and the quantum cryptographic protocol BB84. The main details of the implementation of the toolset are described here, as well as the interfaces developed and the conclusions obtained from the research carried out on the functionalities achieved.

Keywords: Quantum computing, Qiskit, Quantum library, Random numbers generation, Deutsch-Jozsa algorithm, Bernstein-Vazirani algorithm, Grover algorithm, Quantum teleportation, Superdense coding, Quantum cryptography, BB84 Protocol

Índice general

Índice de Figuras

Índice de Tablas

Capítulo 1

Introducción

1.1. Estado del arte

El interés en las tecnologías relacionadas con la computación cuántica ha crecido notablemente en los últimos años, cobrando el tema un gran auge hoy en día. Su utilidad para vulnerar los algoritmos criptográficos actuales ha generado la ya imperiosa necesidad de la creación de nuevos protocolos seguros para las comunicaciones. Por otra parte, dadas algunas de sus curiosas propiedades, es evidente que el fomento de este modelo de computación generará importantes avances tecnológicos para el conjunto de la sociedad.

En 2019, Google anunció haber alcanzado la supremacía cuántica [1]. Esto verificó experimentalmente el hecho de que ciertas tareas computacionales pueden ejecutarse exponencialmente más rápido en un procesador cuántico que en uno clásico. El procesador cuántico *Sycamore* tardó aproximadamente 200 segundos en completar una tarea que, según la referencia del momento, para una supercomputadora clásica de última generación requería aproximadamente 10.000 años.

Recientemente, el 1 de junio de 2022, fue publicado un artículo [2] en la prestigiosa revista científica *Nature* que defendía una mejora cuántica superior, lograda utilizando un procesador fotónico. En dicho trabajo se afirma que por término medio, los mejores algoritmos y superordenadores disponibles tardarían más de 9.000 años en producir, utilizando métodos exactos, una sola muestra de la distribución programada, mientras que Borealis sólo requirió 36 microsegundos. Parece que este podría ser un hito crítico en el camino hacia un ordenador cuántico práctico, pues valida características tecnológicas clave de la fotónica como plataforma para ese objetivo.

Importantes multinacionales tecnológicas, como Google [3], IBM [4] y Amazon [5], invierten sus mejores esfuerzos en investigar y desarrollar el modelo de computación cuántica, así como sus aplicaciones e implementaciones. Si bien existe un gran interés y expectación en los avances que está logrando la computación cuántica, es cierto que su complejidad, debido a la diferencia entre el marco teórico cuántico respecto al clásico, supone un obstáculo para gran parte del sector de la programación. Es por ello que existe un gran interés en fomentar las tecnologías cuánticas, haciéndolas más accesibles para cualquier persona cercana a la informática y para el público en general.

1.2. Objetivos

El objetivo del presente proyecto es el desarrollo de un *toolset open-source* de algoritmos cuánticos, con repositorio público en GitHub, persiguiendo la abstracción y el

encapsulamiento sencillos de *software* cuántico con diferentes funcionalidades. A raíz de esto, fomentar el uso de tecnologías cuánticas es también un principal objetivo de la propuesta presentada en este trabajo.

La librería debe contener una cantidad suficiente de algoritmos y protocolos cuánticos, agrupando algunos de los más simples, para conseguir la accesibilidad en la propuesta; varios procesos que despierten curiosidad con fines didácticos, persiguiendo el aprendizaje en este tipo de tecnologías; y determinados algoritmos pertenecientes a los considerados más famosos, logrando un llamativo trabajo también en el ámbito científico y divulgativo. Entre las librerías existentes que implementan total o parcialmente algoritmos y protocolos tratados en este trabajo destacan [6] y [7], si bien ninguna de ellas contempla las mismas funcionalidades, herramientas y algoritmos de la presente propuesta. Todos los algoritmos que componen *QuantumSolver* han sido parametrizados para estudiar su comportamiento durante la ejecución con diferentes argumentos. Además, en todos los casos se comprueba la entrada, notificando al usuario si se ha cometido un error.

El *toolset* objeto de este trabajo no solo contempla la implementación de esos algoritmos, sino que también los encapsula de manera sencilla para lograr una arquitectura fácilmente extensible, que permita la fácil adición de nuevos procedimientos. Otra funcionalidad de la librería es la ejecución de esos algoritmos en *hardware* real cuántico, así como en simuladores. Este requisito ha sido abarcado proponiendo una solución cómoda para el usuario, en la que no se le obligue a realizar registros en servicios de terceros cuando intente tener un acercamiento sencillo al software.

QuantumSolver está dirigido tanto a un usuario totalmente ajeno a la informática que, por ejemplo, desee obtener un número aleatorio gracias a la computación cuántica; como a un programador experimentado que, por ejemplo, aspire a contribuir en la implementación de esta librería. Debido a que la propuesta pretende satisfacer a una amplia variedad de posible público de los programas ejecutables disponibles, es un objetivo fundamental el desarrollo de la posibilidad de ejecución del software por medio de dos interfaces diferentes: Interfaz por Línea de Comandos (*Command Line Interface*, CLI) e Interfaz Web. La implementación de esta última está más orientada al público general, permitiendo que cualquier usuario pueda ejecutar algoritmos en *hardware* cuántico real, sin tener ningún tipo de experiencia previa con la programación informática. En ambas interfaces, se pueden visualizar los circuitos generados por los algoritmos cuánticos ejecutados.

1.3. Fases

1.3.1. Estudio bibliográfico

Durante los primeros meses del transcurso del proyecto (noviembre, diciembre y enero), se realizó la revisión bibliográfica de contenidos introductorios en el marco teórico de la computación cuántica [8]. Posteriormente, tras tomar decisiones acerca de las tecnologías a utilizar, también se revisaron contenidos formativos de *Qiskit*, el SDK de código abierto que IBM ofrece para trabajar con ordenadores cuánticos a nivel de pulsos, circuitos y módulos de aplicación [9].

1.3.2. Planificación

En esta etapa, que transcurrió durante el mes de enero, se fijaron los requisitos principales del proyecto. Se eligieron algunos de los algoritmos candidatos a estar presentes en el mismo, realizando el correspondiente diseño de clases para la librería y tomando consideraciones acerca de las tecnologías a utilizar (como el posible uso de *Qiskit* o la elección de Python3 como lenguaje en el que desarrollar la librería).

Para organizar estas primeras ideas, y con la intención de que el número de las mismas fuera en aumento según la evolución del proyecto, se decidió utilizar la metodología Kanban, haciendo uso de un *GitHub Project Board* [10].

1.3.3. Desarrollo

La implementación se llevó a cabo a medida que finalizaba la parte de investigación en los algoritmos candidatos y tecnologías necesarias para su desarrollo. Se creó un repositorio público en GitHub [11], con licencia MIT, para alojar el código fuente del proyecto. En este, se se ha creado una rama por requisito y se ha realizado un *Pull Request* con cada una de ellas [12]. Se realizó, por tanto, un *merge* de cada rama de requisito con la principal, al terminar la implementación del mismo. Se ha creado una batería de pruebas unitarias que comprueban el correcto funcionamiento de algunas de las entidades básicas de la librería [13].

1.3.4. Documentación

La manera correcta de descargar, instalar y ejecutar *QuantumSolver* se encuentra recogida en el fichero *README.md* del repositorio [14]. Este fue redactado durante la última fase del proyecto. Durante el desarrollo, se realizaron comentarios en todo el código de la librería utilizando el formato de *Doxygen*, para la generación de documentación automática. La documentación generada se encuentra disponible en la web pública de las *GitHub Pages* del proyecto [15]. Esta brinda ayuda en la visualización de las relaciones existentes entre las diferentes entidades desarrolladas, así como sus datos, funcionalidades y responsabilidades.

1.4. Estructura de la memoria

Este documento comienza describiendo algunos de los avances más importantes en el campo de la computación cuántica, así como el generalizado creciente interés en la misma. Continuando la introducción, se describen los objetivos y fases del desarrollo de este Trabajo Fin de Grado. En el segundo capítulo se realiza la explicación de los diferentes componentes de software del *toolset* cuántico desarrollado. También se indica el programa principal implementado y las interfaces con las que cuenta. La descripción de los algoritmos cuánticos incluidos se halla en el tercer capítulo. Para cada uno de ellos se expone la explicación del mismo, así como las fórmulas, representaciones gráficas y fragmentos de código relevantes. El siguiente capítulo recoge el análisis de la ejecución y resultados de los algoritmos anteriores. Se sintetiza, en el quinto capítulo, la implementación realizada del protocolo criptográfico cuántico BB84, detallando sus entidades, fundamento, ejecución y resultados. Los capítulos sexto y séptimo realizan un recorrido por las conclusiones y líneas futuras; tanto en inglés como en español,

respectivamente. El octavo capítulo recoge una estimación del presupuesto desglosado del total desarrollo de la librería. Finalmente, en apéndices se incluyen tres artículos producto de este trabajo, que han sido enviados a tres congresos. Los dos primeros, uno nacional y uno internacional, han sido ya aceptados y serán presentados en las próximas semanas. El tercero está actualmente en fase de evaluación.

Capítulo 2

Detalles de la implementación

2.1. Componentes

QuantumSolver es una librería cuántica desarrollada en Python3, gracias a *Qiskit*, que cuenta con dos componentes principales: *QExecute* y *QAlgorithmManager*.

2.1.1. QExecute

QExecute es el motor de ejecución de *QuantumSolver*. Se encarga de la autenticación contra los servicios de IBM, que ofrecen acceso a su *hardware* (tanto *hardware* cuántico real como simuladores) por medio de un *API token* de *"IBM Quantum Experience"* [16] [17]. Además cuenta con un modo invitado para no generar la inevitable necesidad al usuario de obtener el *token* teniendo que crear una cuenta en *IBM Quantum*. En este modo solo se permite ejecutar haciendo uso del simulador local *'aer_simulator'*, por lo que no se podrá utilizar el *hardware* cuántico real proporcionado por IBM. *QExecute* cuenta con métodos para la visualización del listado de los *backends* disponibles y la selección del deseado para realizar la ejecución. Además, es el componente encargado de realizar la propia ejecución de los circuitos cuánticos.

2.1.2. QAlgorithmManager

QAlgorithmManager es el gestor de algoritmos cuánticos de *QuantumSolver*. Se encarga de agrupar y listar todos los algoritmos disponibles, además de seleccionar el que se desee ejecutar. También permite gestionar los argumentos de los diferentes algoritmos y del intercambio de información entre ellos y el programa principal.

2.1.3. QAlgorithm

QAlgorithm es la entidad que se corresponde con un algoritmo cuántico cualquiera. Se trata de una clase abstracta que puede servir como plantilla para añadir de manera intuitiva un nuevo algoritmo a la librería. Cualquier entidad válida derivada de esta representa un algoritmo que *QuantumSolver* puede ejecutar. Estas entidades, siguiendo la plantilla de *QAlgorithm*, contienen información relevante sobre el algoritmo en cuestión: nombre, descripción, parámetros, maneras en que se debe analizar y tratar el resultado de la ejecución del circuito, y en que se deben interpretar y comprobar los parámetros que sean introducidos como una lista de cadenas de texto. El método principal de la entidad es la generación parametrizada del circuito cuántico correspondiente al algoritmo.

```python
## An abstract class of Quantum Algorithm, it can be used as template
class QAlgorithm(ABC):
  def __init__(self):
    self.name = 'QAlgorithm'
    self.description = 'QAlgorithm Description'
    self.parameters = [
      {
        'type': '',
        'description': '',
        'constraint': ''
      }
    ]
    ## How to parse the result of the circuit execution
    self.parse_result = lambda counts: counts
    ## How to parse the input parameters
    self.parse_parameters = lambda parameters: []

  ## An abstract method to check the parameters
  @abstractmethod
  def check_parameters(self, parameters):
    pass

  ## An abstract method to generate the circuit
  @abstractmethod
  def circuit(self):
    pass
```

En el código se puede apreciar la sencillez de la plantilla de *QAlgorithm*. Definiendo unos pocos atributos y funciones, cualquier usuario es capaz de añadir un nuevo algoritmo cuántico a la librería, realizando una contribución a la misma que amplíe sus funcionalidades. En la Fig. 2.1 se puede observar el diagrama de herencia sobre la clase abstracta QAlgorithm.

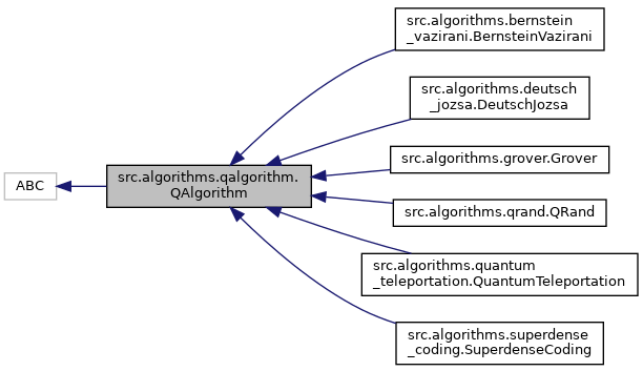


Figura 2.1: Gráfico de herencia de la clase abstracta QAlgorithm

2.2. Producto final

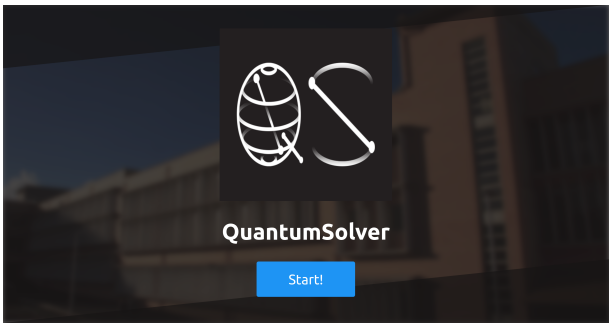
2.2.1. Programa principal

En la pantalla de inicio, el programa principal de *QuantumSolver* ofrece las alternativas de ejecutar el modo invitado, o bien autenticarse usando un *API token* de IBM. En cualquier caso, se despliega un menú que contiene las opciones de visualización y selección de los *backends* y algoritmos disponibles. Una vez elegido un algoritmo, se solicita la introducción de sus parámetros. Cuando *backend*, algoritmo y parámetros han sido establecidos, se despliegan dos opciones. Por una parte, se permite ejecutar el algoritmo una única vez y obtener el resultado, además de una representación gráfica del circuito. Por otra parte, es posible ejecutar el algoritmo varias veces para observar su comportamiento representado en un histograma generado. Se ha denominado "modo experimental" a esta última opción.

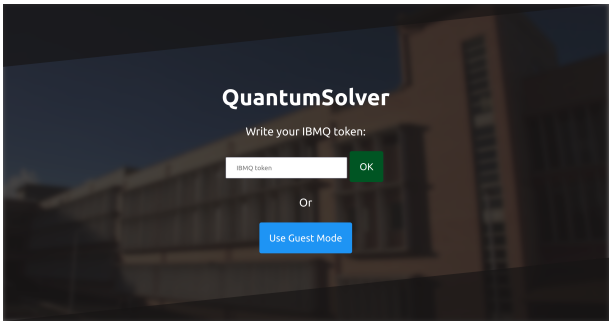
2.2.2. Interfaces

Para *QuantumSolver* se ha desarrollado una versión web que cuenta con un *backend* en Python3, utilizando el *framework* Flask, y un *frontend* usando TypeScript, React, HTML5 y CSS. De las dos interfaces ofrecidas para ejecutar *QuantumSolver*, la interfaz web es más intuitiva para el público general que la basada en línea de comandos, reuniendo ambas las mismas funcionalidades.

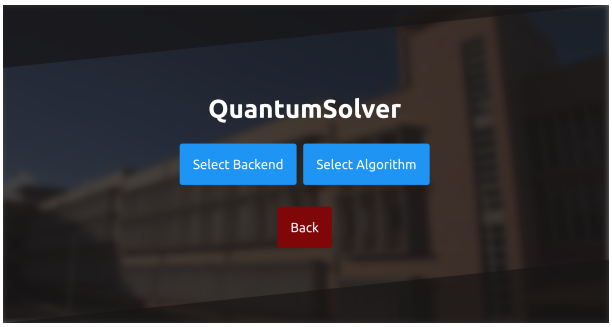
A continuación, en Fig. 2.2, Fig. 2.3, Fig. 2.4 Fig. 2.5, Fig. 2.6 y Fig. 2.7 se exponen diversas capturas de pantalla que facilitan la comparativa entre ambas interfaces.



(a) Pantalla de inicio en la Interfaz Web



(b) Menú de autenticación en la Interfaz Web



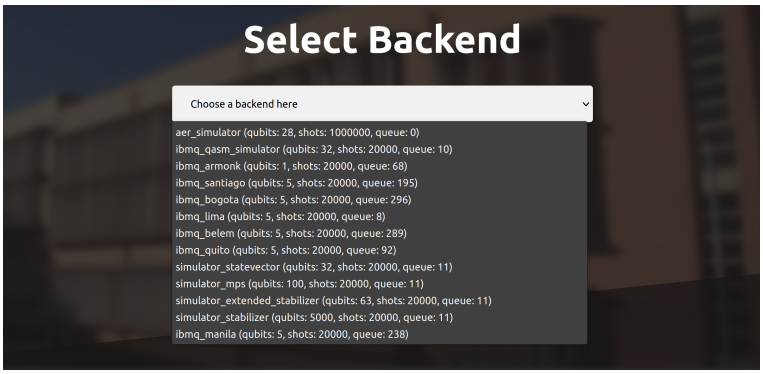
(c) Menú principal en la Interfaz Web

```
QuantumSolver
=============

A little quantum toolset developed using Qiskit
WARNING: The toolset uses your personal IBM Quantum Experience
token to access to the IBM hardware.
You can access to your API token or generate another one here:
https://quantum-computing.ibm.com/account

You can also use the Guest Mode which only allows you to run
quantum circuits in a local simulator ("aer_simulator").

[&] Write your IBM Quantum Experience token (Or Press Enter for Guest Mode):
✔ Loading Guest Mode

QuantumSolver (Guest Mode)
==========================

[1] See available Backends
[2] See available Algorithms
[3] Select Backend
[4] Select Algorithm
[0] Exit
```

(d) Pantalla de inicio, menú de autenticación y menú principal en la CLI

Figura 2.2: Comparativa inicial entre ambas interfaces

```
[&] Select an option: 1

Avaliable backends:
[1]     Name: aer_simulator
        Number of qubits: 28
        Maximum shots: 1000000
        Jobs in queue: 0
        Is a simulator
        ✅ Is operational

[2]     Name: ibmq_qasm_simulator
        Number of qubits: 32
        Maximum shots: 20000
        Jobs in queue: 22
        Is a simulator
        ✅ Is operational

[3]     Name: ibmq_armonk
        Number of qubits: 1
```

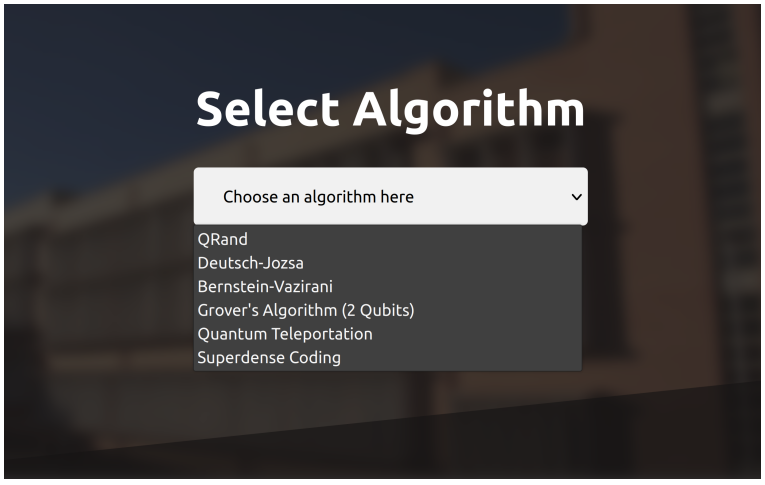


(a) Selección de backend en la CLI (b) Selección de backend en la Interfaz Web

Figura 2.3: Comparativa de selección de backend en ambas interfaces

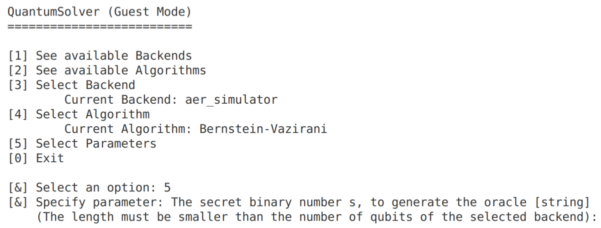
```
[2] See available Algorithms
[3] Select Backend
[4] Select Algorithm
[0] Exit

[&] Select an option: 2

Avaliable algorithms:
[1]     Name: QRand
        Description: Gives a random number between 0 and 2 ^ n_qubits - 1
        Parameters:
            A positive number of qubits to use (int)

[2]     Name: Deutsch-Jozsa
        Description: Given a hidden Boolean function f: f({x_0,x_1,x_2,...}) → 0 or 1, where x_n is 0 or 1; determine whether the given function is balanced or constant. A co
nstant function returns all 0's or all 1's for any input, while a balanced function returns 0's for exactly half of all inputs and 1's for the other half.
        Parameters:
            The oracle type: "constant" or "balanced" (string)

            A positive number of qubits to use (int)

[3]     Name: Bernstein-Vazirani
        Description: Using an oracle: f(x) = (s * x) mod 2. Obtain s (a secret number)
        Parameters:
            The secret binary number s, to generate the oracle (string)

[4]     Name: Grover's Algorithm (2 Qubits)
        Description: Performs the search in an unordered sequence of data
        Parameters:
            Two bits to create the mark state (string)

[5]     Name: Quantum Teleportation
        Description: Transmit one qubit using two classical bits
        Parameters:
            Probability of measure 0 (float)

[6]     Name: Superdense Coding
        Description: Transmit two classical bits using one qubit of communication
        Parameters:
            The message: two classical bits to transmit (string)
```

(a) Selección de algoritmo en la CLI

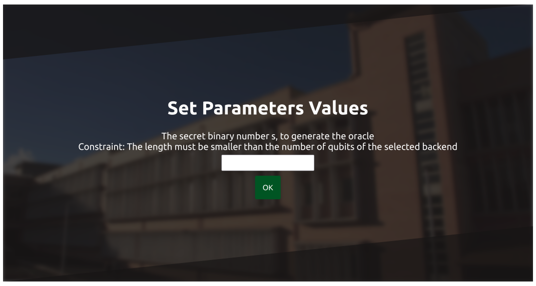


(b) Selección de algoritmo en la Interfaz Web

Figura 2.4: Comparativa de selección de algoritmo en ambas interfaces

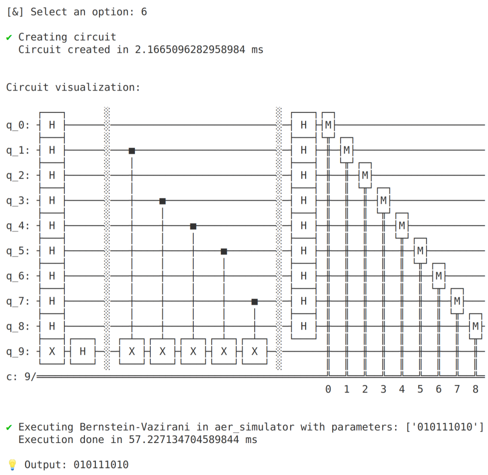


(a) Selección de parámetros en la CLI

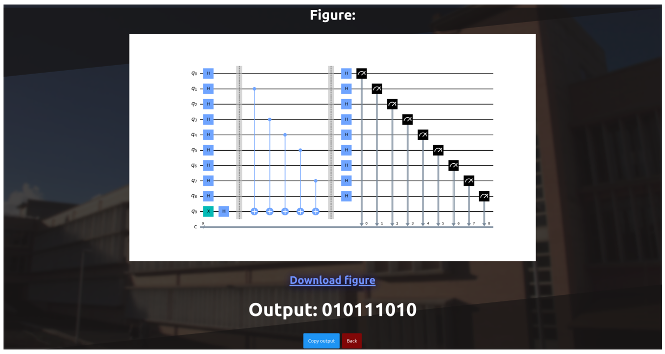


(b) Selección de parámetros en la Interfaz Web

Figura 2.5: Comparativa de selección de parámetros en ambas interfaces

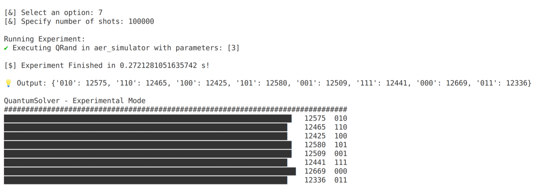


(a) Ejecución simple en la CLI

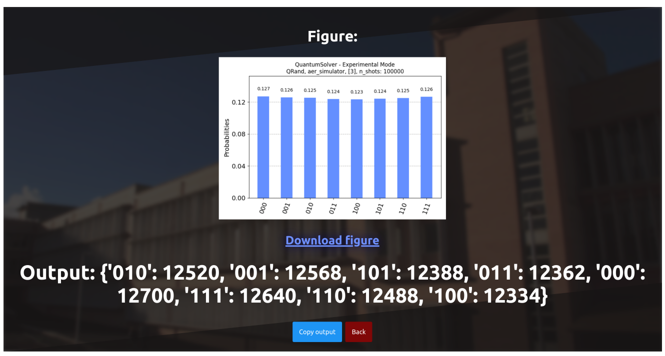


(b) Ejecución simple en la Interfaz Web

Figura 2.6: Comparativa de ejecución simple en ambas interfaces



(a) Ejecución del modo experimental en la CLI



(b) Ejecución del modo experimental en la Interfaz Web

Figura 2.7: Comparativa de ejecución del modo experimental en ambas interfaces

Capítulo 3

Algoritmos cuánticos implementados

3.1. Generación de números aleatorios

El algoritmo cuántico *QRand* implementado en *QuantumSolver* recibe como parámetro un número natural n y permite generar un circuito de cuya ejecución resulta un número aleatorio entre 0 y $2^n - 1$. Su funcionamiento, como se muestra en la Fig. 3.1, se basa en la inicialización de n cúbits, por defecto a $|0\rangle$; la aplicación de una puerta lógica Hadamard [8] [18] a cada uno, para generar un estado de superposición en el que se tenga la misma probabilidad de medir 0 que de medir 1 (véase Ec. (3.1)); y finalmente la medición del resultado, haciendo colapsar cada cúbit en un estado aleatorio e interpretándose como un número binario.

$$|00\ldots0\rangle \xrightarrow{H^{\otimes n}} \frac{1}{\sqrt{2^n}} \sum_{x\in\{0,1\}^n} |x\rangle \tag{3.1}$$

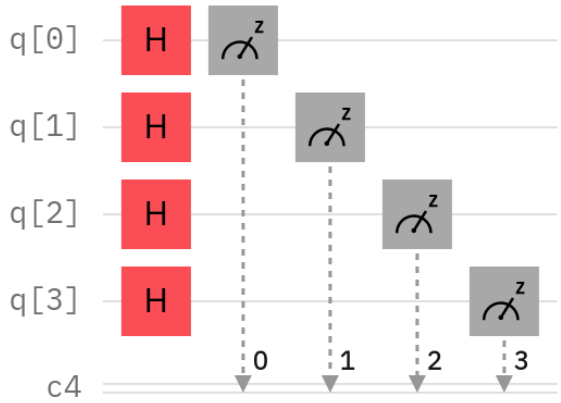


Figura 3.1: Implementación de la generación de números aleatorios para 4 cúbits

```
## Create the QRand circuit
def circuit(self, n=1):
  circuit = QuantumCircuit(n, n)
  n_range = list(range(n))

```

```
    circuit.h(n_range)
    circuit.measure(n_range, n_range)
    return circuit
```

3.2. Algoritmo de Deutsch-Jozsa

El algoritmo de Deutsch-Jozsa [19] es un claro ejemplo de la ventaja existente entre la ejecución de ciertos algoritmos cuánticos y la del mejor algoritmo clásico para resolver el mismo problema.

Dada una función oculta f (véase Ec. (3.2)), que ante una cadena binaria arbitraria devuelve 0 o 1, se debe determinar si f se trata de una función constante o balanceada (se garantiza que es de alguno de estos dos tipos). Una función constante es aquella que ante cualquier entrada, devuelve siempre 0 o siempre 1. Por el contrario, una función balanceada devuelve para exactamente la mitad de las cadenas 0, y para la otra mitad 1.

$$f(\{x_0, x_1, x_2, ..., x_{n-1}\}) \in \{0, 1\}, x_i \in \{0, 1\} \tag{3.2}$$

El número de posibles cadenas binarias de entrada de longitud n es 2^n, por lo que clásicamente se podría evaluar la función hasta encontrar dos valores distintos o hasta llegar a $2^{n-1}+1$ entradas (en el peor caso). Terminado el proceso, si en las evaluaciones obtenidas se ha encontrado un solo valor distinto a los demás, se determinará f balanceada; y si en todas las evaluaciones ha resultado el mismo valor, se determina f constante. En general, es improbable llegar al caso límite de evaluar las $2^{n-1}+1$ entradas, para que en esa última evaluación se demuestre f balanceada. La probabilidad de que la función sea constante crece según el número de entradas k que mantienen el mismo valor, siguiendo la Ec. (3.3). Para obtener un $100\,\%$ de confianza, se deben realizar $2^{n-1}+1$ evaluaciones.

$$p_{\text{constante}}(k) = 1 - \frac{1}{2^{k-1}} \qquad \text{para } 1 < k \leq 2^{n-1} \tag{3.3}$$

Utilizando la computación cuántica, es posible resolver el problema con un $100\,\%$ de certeza tras una única llamada a f. En este caso, la función se implementa gracias a un oráculo cuántico U_f que sigue la expresión de la Ec. (3.4). El símbolo $\oplus$ hace referencia a la suma módulo 2.

$$U_f(|x\rangle\,|y\rangle) = |x\rangle\,|y \oplus f(x)\rangle \tag{3.4}$$

La implementación realizada del algoritmo cuántico de Deutsch-Jozsa [20], recibe dos parámetros: el tipo de oráculo deseado (constante o balanceado) y el número n de cúbits que se utilizan como tamaño de la entrada de f. Una vez introducidos estos parámetros, se genera un circuito de $n+1$ cúbits. De ellos, n conforman el primer registro para codificar la entrada (formada por n cúbits con valor $|0\rangle$); y el restante establece el segundo registro, para la salida del oráculo cuántico (inicializado con el valor $|1\rangle$, obtenido al aplicar una puerta cuántica X [18] a un cúbit que por defecto tiene el valor $|0\rangle$). Además, como se muestra en la Fig. 3.2, se deben aplicar puertas lógicas Hadamard a los $n+1$ cúbits antes y después del oráculo; excepto a la salida del mismo, que no lo precisa dado que no será medida.

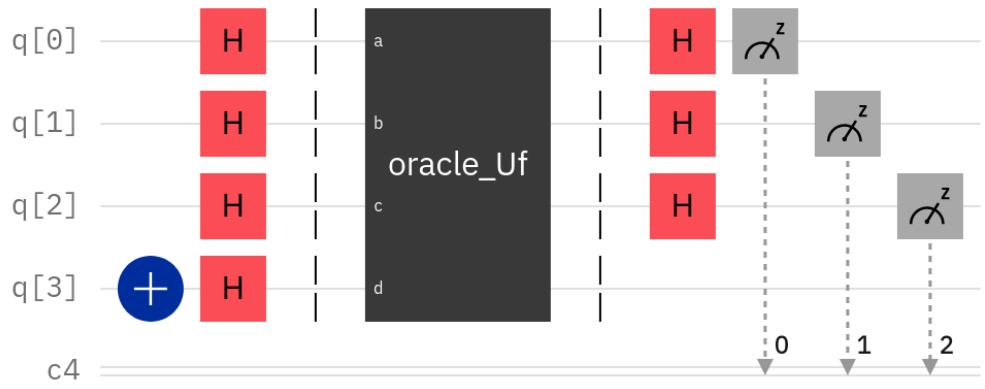


Figura 3.2: Implementación del algoritmo Deutsch-Jozsa para 3 cúbits

```
## Create the Deutsch-Jozsa circuit
def circuit(self, oracle_type='constant', n=1):
  circuit = QuantumCircuit(n + 1, n)
  # Initial setup: Input in state |+>, output in state |->
  circuit.x(n)
  for qubit in range(n + 1):
    circuit.h(qubit)

  # Create and append oracle
  circuit.append(self.create_oracle(oracle_type, n), range(n+1))

  # Finally, perform the H-gates again and measure
  for qubit in range(n):
    circuit.h(qubit)

  circuit.barrier()
  for i in range(n):
    circuit.measure(i, i)

  return circuit
```

En lo que al oráculo U_f se refiere, deberá su implementación al tipo de función f a implementar. Si se opta por f constante, la implementación es bastante sencilla, se ha decidido elegir al azar una de estas dos funciones:

- $\forall x, f(x) = 0$, no aplicar ninguna puerta al registro 2.
- $\forall x, f(x) = 1$, aplicar una puerta X (comparable con el *NOT* clásico) al registro 2.

```
if oracle_type == 'constant':
  if np.random.choice([False, True]):
    oracle_qc.x(n)
```

Para la implementación de una función f balanceada, se deben utilizar n puertas *CNOT* con control en cada uno de los cúbits del primer registro y con objetivo en aquel cúbit perteneciente al segundo registro. La puerta *CNOT* es el "equivalente" al *XOR* clásico. Se puede observar su comportamiento en la Tabla (3.1).

Tabla 3.1: Comportamiento de la puerta lógica CNOT sobre cúbits en estado básico

CNOT			
Antes		Después	
Control	Objetivo	Control	Objetivo
$\|0\rangle$	$\|0\rangle$	$\|0\rangle$	$\|0\rangle$
$\|0\rangle$	$\|1\rangle$	$\|0\rangle$	$\|1\rangle$
$\|1\rangle$	$\|0\rangle$	$\|1\rangle$	$\|0\rangle$
$\|1\rangle$	$\|1\rangle$	$\|1\rangle$	$\|0\rangle$

Se pueden aplicar puertas X al azar antes y después de la aplicación de las puertas *CNOT*, para que el comportamiento de este oráculo de función balanceada no sea predecible.

```
if oracle_type == 'balanced':
  # Generate a random number that indicates which CNOTs to wrap in X-gates
  b = np.random.randint(1, 2 ** n)
  # Format 'b' as a binary string of length 'n', padded with zeros
  b_str = format(b, '0' + str(n) + 'b')

  # Each digit in the binary string corresponds to a qubit,
  # if it is 1 apply an X-gate to that qubit
  for i, current_char in enumerate(b_str):
    if current_char == '1':
      oracle_qc.x(i)

  # Do the controlled-NOT gates for each qubit,
  # using the output qubit as the target
  for qubit in range(n):
    oracle_qc.cx(qubit, n)

  # Place the final X-gates
  for i, current_char in enumerate(b_str):
    if current_char == '1':
      oracle_qc.x(i)
```

La Fig. 3.3 muestra un ejemplo de implementación del algoritmo Deutsch-Jozsa para 3 cúbits (con oráculo transparente).

La inicialización de los cúbits se muestra en la Ec. (3.5), para establecer los valores indicados a los cúbits; y en la Ec. (3.6), tras aplicar las puertas Hadamard por primera vez.

$$|\psi_0\rangle = |0\rangle^{\otimes n}|1\rangle \tag{3.5}$$

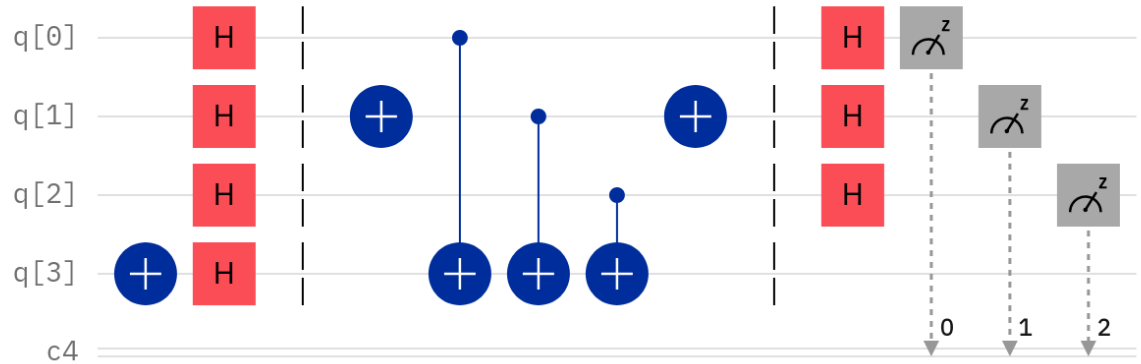


Figura 3.3: Ejemplo de implementación del algoritmo Deutsch-Jozsa para 3 cúbits

$$|\psi_1\rangle = \frac{1}{\sqrt{2^{n+1}}} \sum_{x=0}^{2^n-1} |x\rangle \left(|0\rangle - |1\rangle\right) \tag{3.6}$$

La aplicación del oráculo U_f sobre los cúbits inicializados se ilustra en la Ec. (3.7). Se llega a la conclusión ilustrada, al considerar que $\forall x, f(x) \in \{0, 1\}$.

$$\begin{aligned} |\psi_2\rangle &= \frac{1}{\sqrt{2^{n+1}}} \sum_{x=0}^{2^n-1} |x\rangle(|f(x)\rangle - |1 \oplus f(x)\rangle) \\ &= \frac{1}{\sqrt{2^{n+1}}} \sum_{x=0}^{2^n-1} (-1)^{f(x)} |x\rangle(|0\rangle - |1\rangle) \end{aligned} \tag{3.7}$$

A continuación, el segundo registro de la salida del oráculo y por tanto, su correspondiente cúbit puede ser ignorado. Se aplican n puertas Hadamard a cada cúbit correspondiente con el primer registro de salida del oráculo, y el comportamiento queda reflejado en la Ec. (3.8). En esta, $x \cdot y = x_0 y_0 \oplus x_1 y_1 \oplus \ldots \oplus x_{n-1} y_{n-1}$ resultando la suma en módulo 2 del producto bit a bit.

$$\begin{aligned} |\psi_3\rangle &= \frac{1}{2^n} \sum_{x=0}^{2^n-1} (-1)^{f(x)} \left[\sum_{y=0}^{2^n-1} (-1)^{x \cdot y} |y\rangle \right] \\ &= \frac{1}{2^n} \sum_{y=0}^{2^n-1} \left[\sum_{x=0}^{2^n-1} (-1)^{f(x)} (-1)^{x \cdot y} \right] |y\rangle \end{aligned} \tag{3.8}$$

Al medir el primer registro, se evaluará en una cadena de tamaño n formada por todos los bits a 0, si f es constante; o en otro caso, si es balanceado. La probabilidad de medir la función constante, es decir, de realizar la medición de todos los bits a 0, viene determinada por la Ec. (3.9). Esta se toma el valor 1 con f constante y 0 con f balanceada.

$$p_{medir}(|0\rangle^{\otimes n}) = |\frac{1}{2^n} \sum_{x=0}^{2^n-1} (-1)^{f(x)}|^2 \tag{3.9}$$

Cuando f es constante, el oráculo no tiene ningún efecto (ignorando los efectos sobre la fase global) en los cúbits de entrada, por lo que los estados cuánticos antes y después de consultar el oráculo son los mismos (véase Ec. (3.10)). Como la puerta H es su propia

inversa, aplicándolas antes y después del oráculo se obtiene el estado cuántico inicial $|00\ldots0\rangle$ en el primer registro.

$$H^{\otimes n}\begin{bmatrix}1\\0\\0\\\vdots\\0\end{bmatrix} = \tfrac{1}{\sqrt{2^n}}\begin{bmatrix}1\\1\\1\\\vdots\\1\end{bmatrix} \xrightarrow{\text{tras aplicar } U_f} H^{\otimes n}\tfrac{1}{\sqrt{2^n}}\begin{bmatrix}1\\1\\1\\\vdots\\1\end{bmatrix} = \begin{bmatrix}1\\0\\0\\\vdots\\0\end{bmatrix} \tag{3.10}$$

Sin embargo, cuando f es balanceada, el retroceso de fase [21] añade una fase negativa a exactamente la mitad de estos estados (véase Ec. (3.11)). El estado cuántico después de consultar al oráculo es ortogonal al estado cuántico antes de consultar el oráculo. De esta manera, al aplicar las últimas puertas H, se obtiene un estado cuántico que es ortogonal a $|00\ldots0\rangle$. Esto significa que nunca se podrá medir el estado con todos los bits a cero.

$$U_f\tfrac{1}{\sqrt{2^n}}\begin{bmatrix}1\\1\\1\\\vdots\\1\end{bmatrix} = \tfrac{1}{\sqrt{2^n}}\begin{bmatrix}-1\\1\\-1\\\vdots\\1\end{bmatrix} \tag{3.11}$$

3.3. Algoritmo de Bernstein-Vazirani

El algoritmo cuántico de Bernstein-Vazirani [22], se trata de un caso particular del algoritmo Deutsch-Jozsa. La implementación incluida en *QuantumSolver* recibe como parámetro una clave formada por una cadena binaria de tamaño n, que se utiliza para codificar un oráculo que brinda información acerca de dicha clave. Concretamente, ante una cadena candidata, la información que devuelve el algoritmo es la confirmación de si el número de coincidencias de 1 entre las cadenas clave y candidata es par o impar.

La Ec. (3.12) refleja que el oráculo realiza el producto binario entre las parejas de bits de la clave a adivinar y la cadena candidata, y luego le aplica la suma módulo 2 a los n bits resultantes.

$$f_s(x) = (s * x)(mod\ 2) \tag{3.12}$$

Clásicamente, se podría resolver este problema con n consultas al oráculo (Ec. (3.13)).

$$\begin{cases} f_s(100\ldots0) = s_0 \\ f_s(010\ldots0) = s_1 \\ f_s(001\ldots0) = s_2 \\ \ldots \\ f_s(000\ldots1) = s_{n-1} \end{cases} \tag{3.13}$$

En el caso cuántico se necesita solo una consulta al oráculo, que devolverá correctamente la clave con un 100 % de probabilidad (sin contar con posibles errores de ruido generados por el *hardware*).

La implementación realizada [23] es muy similar a la del algoritmo Deutsch-Jozsa, la única diferencia reside en el oráculo. Internamente, este debe implementarse aplicando

puertas *CNOT* con control en aquellos cúbits que se correspondan con los bits de la clave que estén a 1, y objetivo en la salida del oráculo.

```
def create_oracle(self, circuit, n):
  for i, char in enumerate(reversed(secret_number)):
    if char == '1':
      circuit.cx(i, n)
  return circuit
```

La Fig. 3.4 muestra un ejemplo de implementación del oráculo Bernstein-Vazirani con clave oculta “011”.

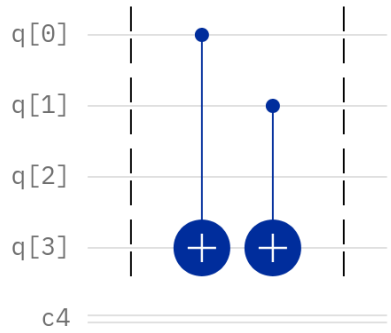


Figura 3.4: Ejemplo de implementación del oráculo Bernstein-Vazirani con clave oculta “011”

La caracterización del oráculo, aplicado sobre la cadena x candidata, se muestra en la Ec. (3.14).

$$|x\rangle \xrightarrow{f_s} (-1)^{s\cdot x}|x\rangle \tag{3.14}$$

La transformación realizada de los n cúbits que codifican la entrada $|00\ldots0\rangle$ al aplicarles las puertas Hadamard de la inicialización se muestra en la Ec. (3.15).

$$|00\ldots0\rangle \xrightarrow{H^{\otimes n}} \frac{1}{\sqrt{2^n}} \sum_{x\in\{0,1\}^n} |x\rangle \tag{3.15}$$

La aplicación del oráculo cuántico se muestra en la Ec. (3.16).

$$\frac{1}{\sqrt{2^n}} \sum_{x\in\{0,1\}^n} |x\rangle \xrightarrow{f_a} \frac{1}{\sqrt{2^n}} \sum_{x\in\{0,1\}^n} (-1)^{a\cdot x}|x\rangle \tag{3.16}$$

El paso final, aplicando a cada cúbit anterior una puerta Hadamard, se muestra en la Ec. (3.17).

$$\frac{1}{\sqrt{2^n}} \sum_{x\in\{0,1\}^n} (-1)^{a\cdot x}|x\rangle \xrightarrow{H^{\otimes n}} |a\rangle \tag{3.17}$$

Se observa que, al aplicar las operaciones descritas, el resultado es la clave codificada en el oráculo.

3.4. Algoritmo de Grover

El algoritmo de Grover [24] demuestra la superior capacidad de velocidad en la búsqueda de bases de datos de un ordenador cuántico sobre un ordenador clásico. Puede acelerar cuadráticamente la resolución de un problema de búsqueda no estructurada, aunque también sirve como técnica general o subrutina para obtener mejoras cuadráticas en el tiempo de ejecución de una variedad de algoritmos. Este procedimiento se denomina la amplificación de la amplitud.

En una lista no ordenada de N elementos, clásicamente se deben consultar uno a uno hasta encontrar el buscado. Para encontrar ese elemento marcado, se tiene que comprobar una media de $N/2$ elementos, N en el peor caso. Con la técnica cuántica de la amplificación de la amplitud, se logra encontrar el elemento en $\sqrt{N}$ pasos. Esto resulta en un aumento cuadrático de la velocidad, lo que supone un ahorro de tiempo considerable para encontrar elementos marcados en listas largas. Además, el algoritmo no utiliza la estructura interna de lista, lo que lo hace genérico y proporciona inmediatamente una aceleración cuántica cuadrática para un alto número de problemas clásicos.

Para implementar la base de datos [25] se puede recurrir a un oráculo U_ω descrito por la Ec. (3.18) y en notación matricial por la Ec. (3.19). Se debe tener en cuenta que $f(x) = 0$ si x no es un elemento buscado $(x \neq w)$ y $f(x) = 1$ si x es un elemento buscado $(x = w)$.

$$U_\omega|x\rangle = (-1)^{f(x)}|x\rangle \tag{3.18}$$

$$U_\omega = \begin{bmatrix} (-1)^{f(0)} & 0 & \cdots & 0 \\ 0 & (-1)^{f(1)} & \cdots & 0 \\ \vdots & 0 & \ddots & \vdots \\ 0 & 0 & \cdots & (-1)^{f(2^n-1)} \end{bmatrix} \tag{3.19}$$

Véase un ejemplo para 2 cúbits, marcando el elemento “10” en la Ec. (3.20).

$$U_\omega = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \leftarrow \omega = 10 \tag{3.20}$$

Una vez aclarado el tipo de dato para la representación de la base de datos, se procede a explicar la técnica de la amplificación de la amplitud.

1. Se establece el vector $|s\rangle = H^{\otimes n}|0\rangle^n$, dando lugar a una superposición uniforme. Es por ello que en la implementación realizada se han inicializado los n cúbits y se les ha aplicado una puerta Hadamard a cada uno.
   En la Fig. 3.5, la gráfica de la izquierda corresponde al plano bidimensional formado por los vectores perpendiculares $|w\rangle$ y $|s'\rangle$, que permite expresar el estado inicial como $|s'\rangle = \sin\theta\,|w\rangle + \cos\theta\,|s'\rangle$, donde $\theta = \arcsin\langle s|w\rangle = \arcsin\frac{1}{\sqrt{N}}$. A la derecha se muestra el gráfico de barras de las amplitudes del estado $|s\rangle$.

2. Se aplica el oráculo de reflexión U_f al estado $|s\rangle$. La amplitud de $|w\rangle$ queda negativa al reflectar $|s\rangle$ sobre $|s'\rangle$ (véase Fig. 3.6). En la implementación, se consigue crear un oráculo U_f aplicando una puerta CZ entre los dos cúbits. Además, en el cúbit

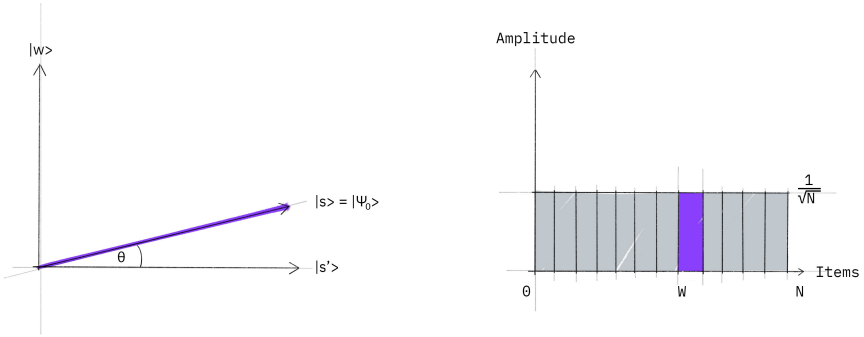


Figura 3.5: Paso 1 del algoritmo de Grover

contrario a aquellos cúbits que están establecidos a 0 en el elemento buscado, se debe aplicar una puerta S antes y después de la CZ. Por ejemplo, para el elemento "10" consúltese la Fig. 3.7.

```
## Create the oracle
def get_oracle(self, n, mark_state):
  oracle = QuantumCircuit(n, n)

  for i, char in enumerate(mark_state):
    if char == '0':
      oracle.s(i)

  oracle.cz(0, 1)

  for i, char in enumerate(mark_state):
    if char == '0':
      oracle.s(i)

  return oracle
```

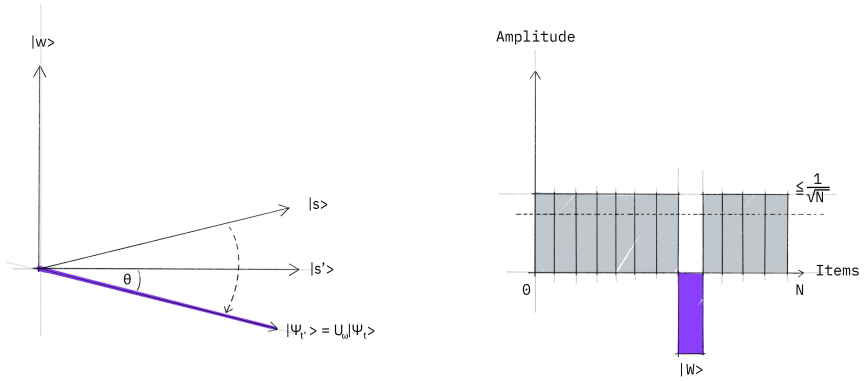


Figura 3.6: Paso 2 del algoritmo de Grover

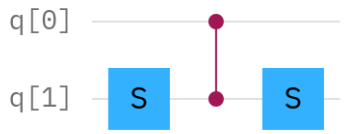


Figura 3.7: Implementación del oráculo o reflector U_f en el algoritmo de Grover de 2 cúbits para el elemento “10”

3. Se aplica una reflexión adicional $U_s = 2|s\rangle\langle s| - 1$ para completar la transformación (véase Fig. 3.8). Para implementar esta reflexión, se debe aplicar una puerta Hadamard y una Z a ambos cúbits. A continuación una puerta CZ entre ellos y finalizar con una Hadamard en ambos (véase Fig. 3.9).

```
## Create the diffusor
def get_diffusor(self, n, n_range):
  diffusor = QuantumCircuit(n, n)
  diffusor.h(n_range)
  diffusor.z(n_range)
  diffusor.cz(0, 1)
  diffusor.h(n_range)
  return diffusor
```

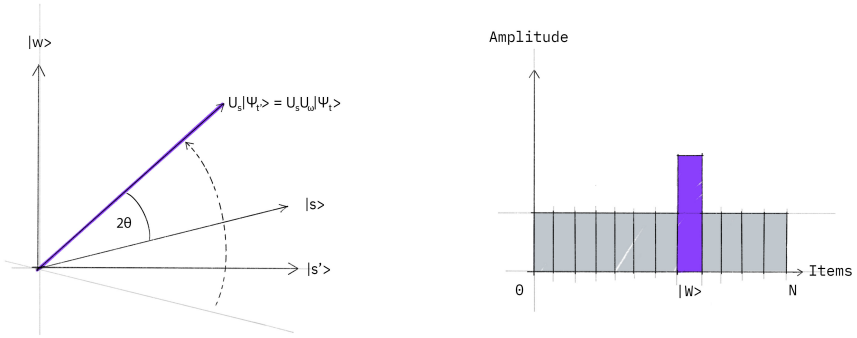


Figura 3.8: Paso 3 del algoritmo de Grover

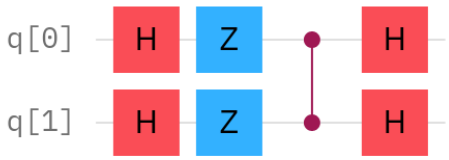


Figura 3.9: Implementación del reflector U_s en el algoritmo de Grover de 2 cúbits

La transformación realizada $U_s U_f$ (pasos 2 y 3) consigue rotar $|s\rangle$ más cerca de $|w\rangle$. Se puede denotar la aplicación de t iteraciones sobre estos pasos como $|\psi_t\rangle = (U_s U_f)^t |s\rangle$. Con aproximadamente $t = \sqrt{N}$ iteraciones es suficiente como para finalizar el procedimiento.

La Fig. 3.10 muestra un circuito en alto nivel del algoritmo de Grover de n cúbits.

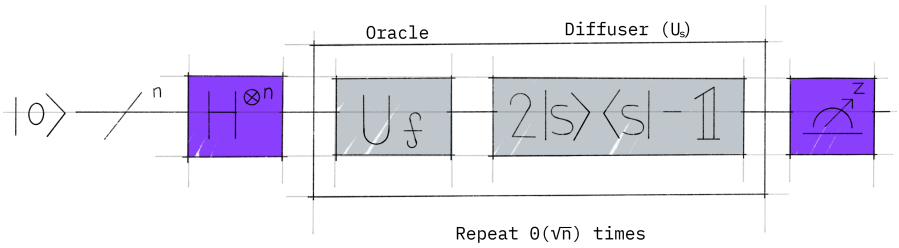


Figura 3.10: Circuito en alto nivel del algoritmo de Grover de n cúbits

Para la implementación realizada se ha calculado el valor de t (número de iteraciones). Con $N = 4$, $\theta = \arcsin \frac{1}{\sqrt{4}} = \frac{\pi}{6}$. Después de t pasos se tiene que $(U_s U_\omega)^t |s\rangle = \sin\theta_t |\omega\rangle + \cos\theta_t |s'\rangle$, donde $\theta_t = (2t+1)\theta$. Se quiere obtener $\theta_t = \frac{\pi}{2}$ teniendo $\theta = \frac{\pi}{6}$, por lo que $t = 1$. Esto significa que solo hace falta un única iteración para encontrar el elemento buscado. Un ejemplo de circuito generado por la implementación parametrizada, con el elemento buscado "10", se puede observar en la Fig. 3.11.

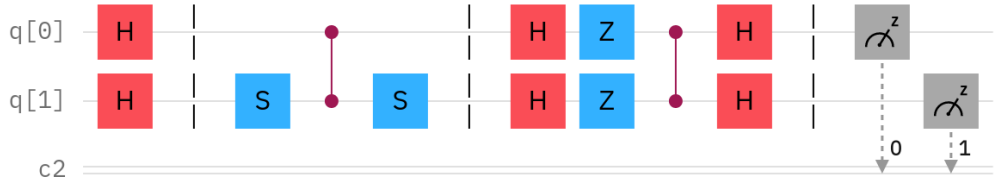


Figura 3.11: Circuito del algoritmo de Grover de 2 cúbits con elemento buscado "10"

3.5. Teletransporte cuántico

El teletransporte cuántico utiliza dos bits clásicos para transmitir un cúbit (véase Fig. 3.12). Según el teorema de no clonación [26], es imposible clonar un cúbit. Solo es posible clonar estados clásicos, no superposiciones. Al transmitir un cúbit de un emisor a un receptor, el emisor lo pierde; es por ello que se denomina teletransporte cuántico.

Se procede a describir la implementación realizada [27], que simula la comunicación cuántica haciendo uso de circuitos cuánticos.

Primeramente, emisor y receptor comparten un par entrelazado de cúbits. Entonces, el emisor realiza una serie de operaciones en su cúbit y envía los resultados al receptor por medio de dos bits clásicos. Tras recibir esta información, el receptor es capaz de realizar operaciones para obtener el cúbit del emisor.

El par entralazado se puede conseguir aplicando una puerta Hadamard a un cúbit y posteriormente hacerlo control de una *CNOT* que tiene como objetivo otro cúbit inicializado a $|0\rangle$. A continuación, el emisor procede a aplicar otra *CNOT* con control en el cúbit que

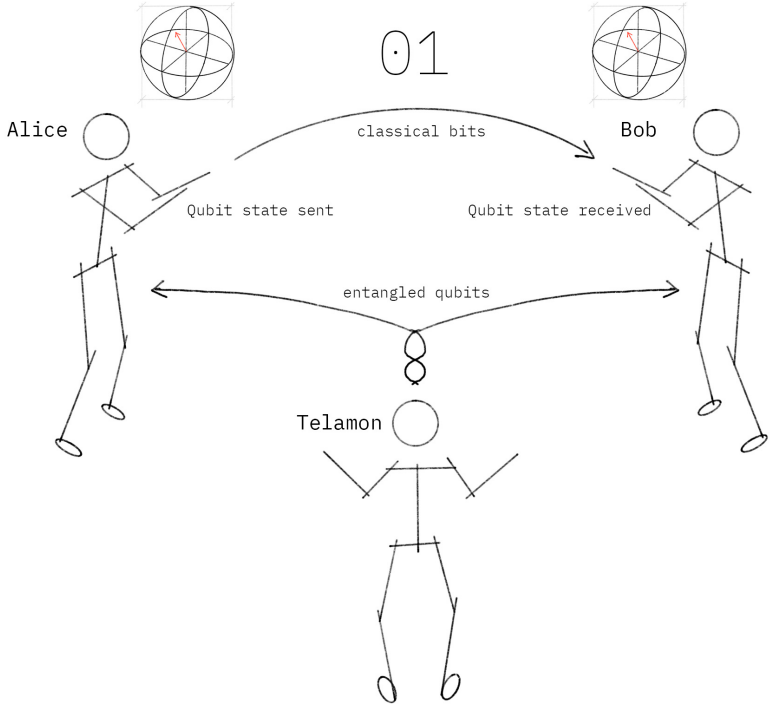


Figura 3.12: Ilustración del teletransporte cuántico a alto nivel

desea enviar y objetivo en aquel que posee del par entralazado. Tras esto, procede aplicar una última puerta Hadamard al cúbit al que aplicó el control de la *CNOT* y a medir ambos cúbits, enviando los resultados al receptor. Este seguirá las siguientes instrucciones y le aplicará las operaciones correspondientes a su cúbit del par entrelazado, obteniendo el cúbit que el emisor quiso enviar:

- $00 \rightarrow$ No hacer nada.
- $01 \rightarrow$ Aplicar una puerta X.
- $10 \rightarrow$ Aplicar una puerta Z.
- $11 \rightarrow$ Aplicar una puerta Z y después una puerta X.

Para el desarrollo del circuito se han seguido las consideraciones que compatibilizan su implementación con su ejecución en hardware real de IBM [28]. Además, para brindar una mayor interacción con el usuario, se ha parametrizado (como el resto de los algoritmos implementados) logrando la elección de la probabilidad de medir 0 en el cúbit a transmitir. Para ello, se genera el vector de estado cuántico $(\sqrt{p_{medir}(0)}, \sqrt{1 - p_{medir}(0)})$, y se asocia inicialmente al cúbit teletransportado.

```
## Create a statevector of a qubit that has the specified probability to measure zero
def get_statevector(self, measure_zero_prob):
  a1 = sqrt(measure_zero_prob)
  a2 = sqrt(1 - measure_zero_prob)
  return Statevector([complex(a1, 0), complex(a2, 0)])
```

Este proceso se realiza en el módulo *initialize* presente en la Fig. 3.13, donde se ilustra el circuito completo implementado.

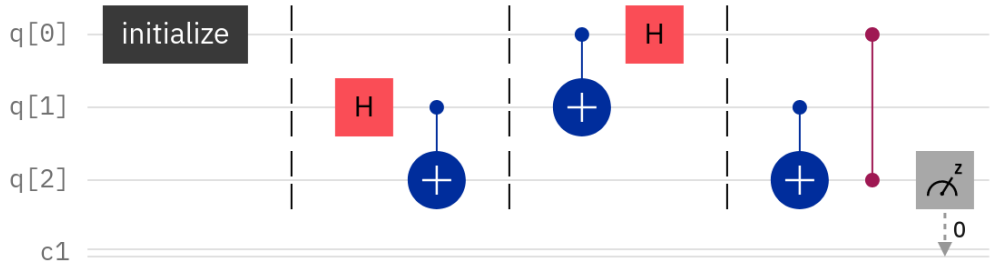


Figura 3.13: Circuito implementado del teletransporte cuántico

El correcto funcionamiento de esta característica se puede comprobar en el modo experimental, al ejecutar el circuito un número elevado de veces, obteniendo aproximadamente el porcentaje de mediciones en 0 solicitado por el usuario.

3.6. Protocolo de codificación superdensa

De manera similar al teletransporte cuántico, el protocolo de codificación superdensa [29] es capaz de transmitir dos bits clásicos por medio de un único cúbit de comunicación (véase Fig. 3.14). Los dos algoritmos son parecidos dado que, en cierta medida, el emisor y el receptor "intercambian sus equipos".

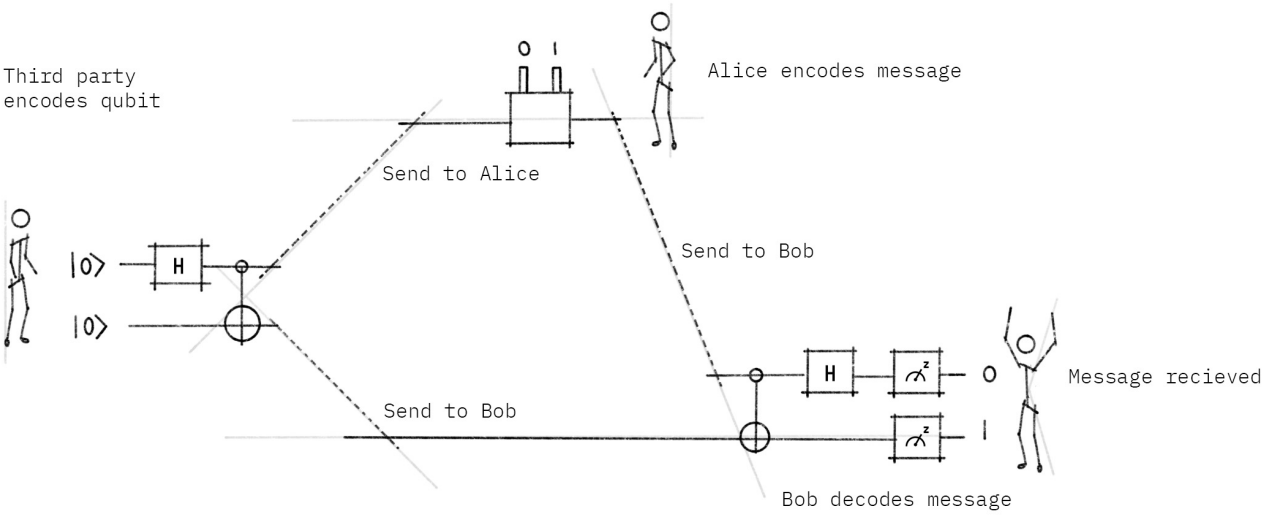


Figura 3.14: Ilustración del protocolo de codificación superdensa a alto nivel

La implementación sigue los siguientes pasos:

1. Emisor (A) y receptor (B) comparten un par entrelazado de cúbits. Para generarlo, primeramente se inicializan ambos a $|0\rangle$ (véase Ec. (3.21)).

$$|00\rangle = |0\rangle_A \otimes |0\rangle_B \tag{3.21}$$

Se aplica una puerta Hadamard al cúbit del emisor, dejándolo en estado de superposición (véase Ec. (3.22)).

$$|+0\rangle = \frac{1}{\sqrt{2}}(|00\rangle + |10\rangle) \tag{3.22}$$

Por último se aplica una *CNOT* con control en el cúbit del emisor y objetivo en el del receptor (véase Ec. (3.23)).

$$\text{CNOT}\frac{1}{\sqrt{2}}(|00\rangle + |10\rangle) = \frac{1}{\sqrt{2}}(|00\rangle + |11\rangle) \tag{3.23}$$

```
## Create the simplest (and maximal) example of quantum entanglement
def create_bell_pair(self):
  bell_state_circuit = QuantumCircuit(2)
  bell_state_circuit.h(1)
  bell_state_circuit.cx(1, 0)
  return bell_state_circuit
```

2. Cada cúbit es enviado a su correspondiente dueño. La meta final del emisor es enviar dos bits clásicos por medio de su cúbit. Para ello se aplican las reglas ilustradas en la Tabla (3.2). Dependiendo del mensaje a codificar, aplica a su cúbit las puertas indicadas.

Tabla 3.2: Normas de codificación en la codificación superdensa

Mensaje a enviar	Puerta aplicada	Estado resultante
00	I	$\|00\rangle + \|11\rangle$
01	X	$\|10\rangle + \|01\rangle$
10	Z	$\|00\rangle - \|11\rangle$
11	ZX	$-\|10\rangle + \|01\rangle$

```
## Encode a message
def encode_message(self, circuit, qubit, msg):
  if msg[0] == '1':
    circuit.z(qubit)
  if msg[1] == '1':
    circuit.x(qubit)
  return circuit
```

3. El receptor recibe el cúbit del emisor y le practica unas operaciones de restauración para “invertir” el proceso inicial del entrelazamiento. Estas operaciones son una *CNOT* con control en el cúbit recibido por el emisor y objetivo en el propio del receptor, y posteriormente una Hadamard al del emisor. En la Tabla (3.3) se ilustran los casos posibles. Al medir los cúbits obtendrá el mensaje del emisor.

Tabla 3.3: Casos de decodificación en la codificación superdensa

Mensaje recibido	Tras aplicar la *CNOT*	Tras aplicar la Hadamard
$\|00\rangle + \|11\rangle$	$\|00\rangle + \|10\rangle$	$\|00\rangle$
$\|10\rangle + \|01\rangle$	$\|11\rangle + \|01\rangle$	$\|01\rangle$
$\|00\rangle - \|11\rangle$	$\|00\rangle - \|10\rangle$	$\|10\rangle$
$-\|10\rangle + \|01\rangle$	$-\|11\rangle + \|01\rangle$	$\|11\rangle$

```
## Decode a message
def decode_message(self, circuit):
  circuit.cx(1, 0)
  circuit.h(1)
  return circuit
```

Se ha implementado un algoritmo parametrizado que permite realizar el protocolo de codificación superdensa para cualquier mensaje binario de longitud dos. Se puede observar un ejemplo de circuito generado para el mensaje “11” en la Fig. 3.15.

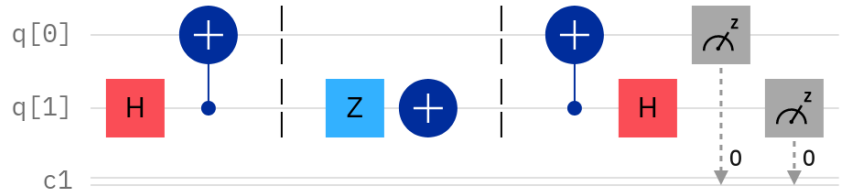


Figura 3.15: Circuito del protocolo de codificación superdensa para el mensaje “11”

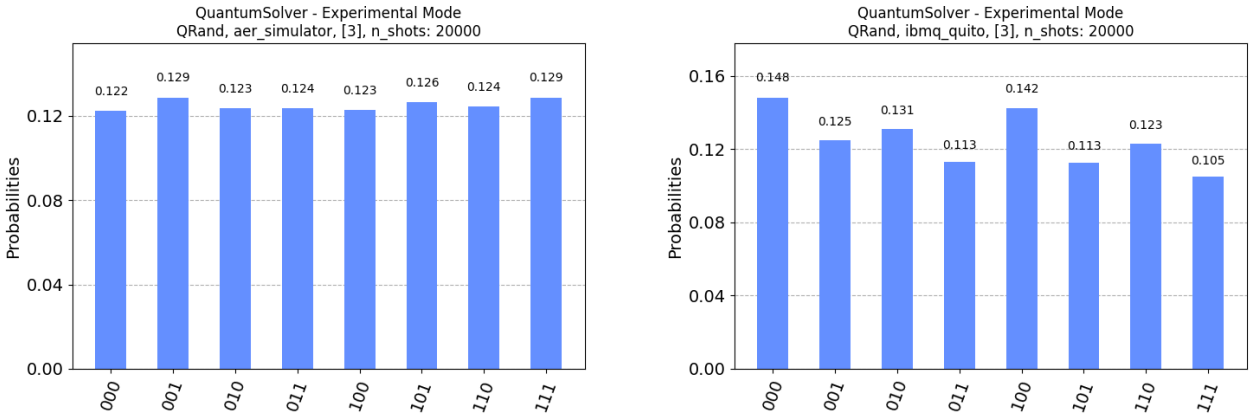
Capítulo 4

Ejecución y resultados

Con la finalidad de corroborar el correcto funcionamiento de todos los algoritmos desarrollados, se ha comparado los resultados obtenidos con los expuestos por IBM. Para todos ellos, se realizan dos ejecuciones del modo experimental, una en el simulador local "aer_simulator" y otra en el procesador cuántico de IBM basado en circuitos "ibmq_quito", que cuenta con cinco cúbits en su arquitectura superconductora. Ambos se ejecutan 20.000 veces con los mismos parámetros y se comprueban los resultados obtenidos, comparando los histogramas generados.

4.1. Generación de números aleatorios

En la generación de números aleatorios, el resultado esperado es que el número de repeticiones de cada elemento se distribuya uniformemente. En este caso, se ha utilizado el parámetro de tres cúbits, por lo que se deben generar números del 0 al 7 de manera aleatoria. Podemos comprobar que en las dos ejecuciones realizadas (Fig. 4.1a y Fig. 4.1b) los resultados son similares; aunque evidentemente, en el caso cuántico real existen imprecisiones que dan lugar a los errores propios de estos sistemas físicos. La mitigación de estos errores es todo un campo de investigación en activo.



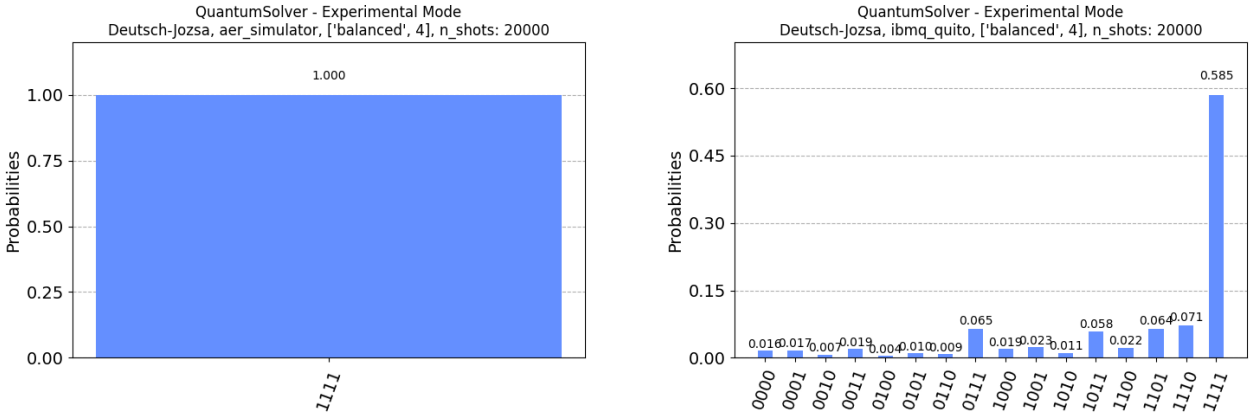
(a) Utilizando el backend "aer_simulator"

(b) Utilizando el backend "ibmq_quito"

Figura 4.1: Histograma tras 20.000 ejecuciones de QRand

4.2. Algoritmo de Deutsch-Jozsa

El algoritmo de Deutsch-Jozsa devuelve una cadena de n ceros si el oráculo implementado codifica una función constante. Se debe interpretar la obtención de cualquier otro caso como una función balanceada. En la simulación (Fig. 4.2a), el $100\,\%$ de las ejecuciones determinan correctamente el tipo de función. En el *hardware* cuántico real (Fig. 4.2b), el $58{,}5\,\%$ de los resultados coinciden con la simulación exenta de ruido. Cabe añadir que, considerando que se determina la función como constante solo cuando se obtienen n ceros, la probabilidad de acierto obtenida es del $99{,}984\,\%$.



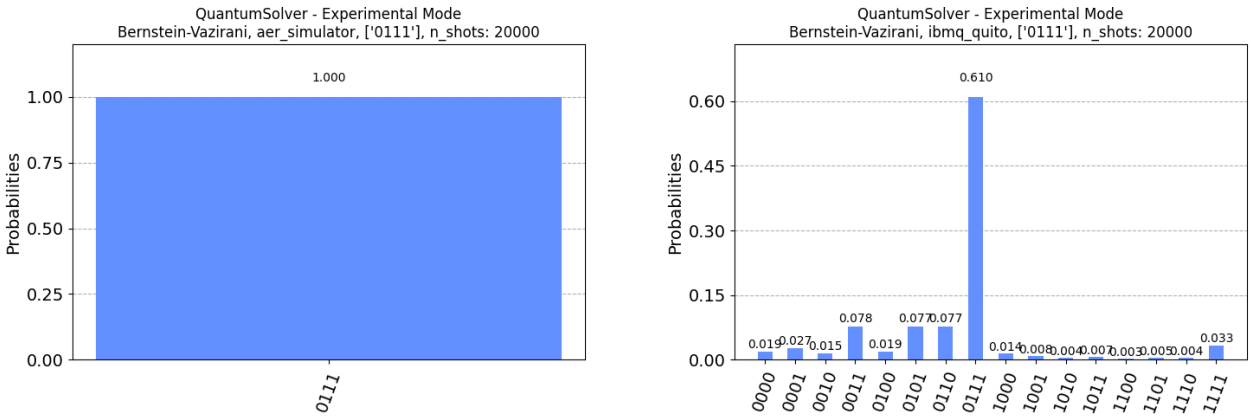
(a) Utilizando el backend “aer_simulator”

(b) Utilizando el backend “ibmq_quito”

Figura 4.2: Histograma tras 20.000 ejecuciones de Deutsch-Jozsa

4.3. Algoritmo de Bernstein-Vazirani

En las ejecuciones del algoritmo de Bernstein-Vazirani, el simulador (Fig. 4.3a) acierta la clave oculta en el oráculo en todas ellas. Sin embargo, en el *hardware* de IBM (Fig. 4.3b), solo acierta en un $61\,\%$ de los intentos.



(a) Utilizando el backend “aer_simulator”

(b) Utilizando el backend “ibmq_quito”

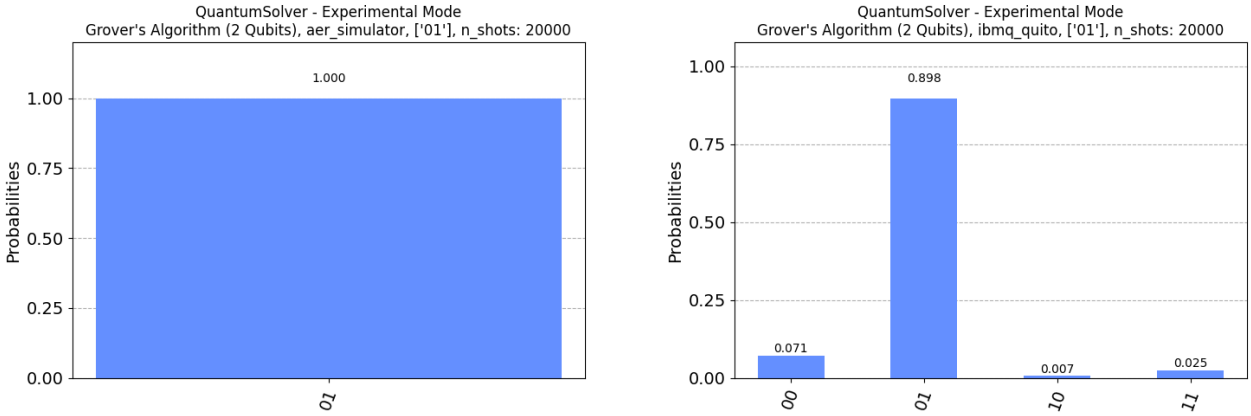
Figura 4.3: Histograma tras 20.000 ejecuciones del algoritmo Bernstein-Vazirani

Como ocurre en la mayoría de algoritmos similares, son más probables los casos en los que hay un único error en el valor de uno de los bits que conforman la clave. En este

caso, en el que la clave es 0111, ha sido más probable también que el oráculo determine las candidatas a clave: 0110, 0101, 0011 y 1111.

4.4. Algoritmo de Grover

El algoritmo de Grover con parámetro que define el estado marcado “01” debe de encontrar ese elemento y devolverlo. Nuevamente, en la simulación (Fig. 4.4) se obtiene un éxito del $100\,\%$; mientras que en el *hardware* cuántico real se ha acertado en un $89{,}8\,\%$ de los casos. Los casos restantes han devuelto estados no marcados.

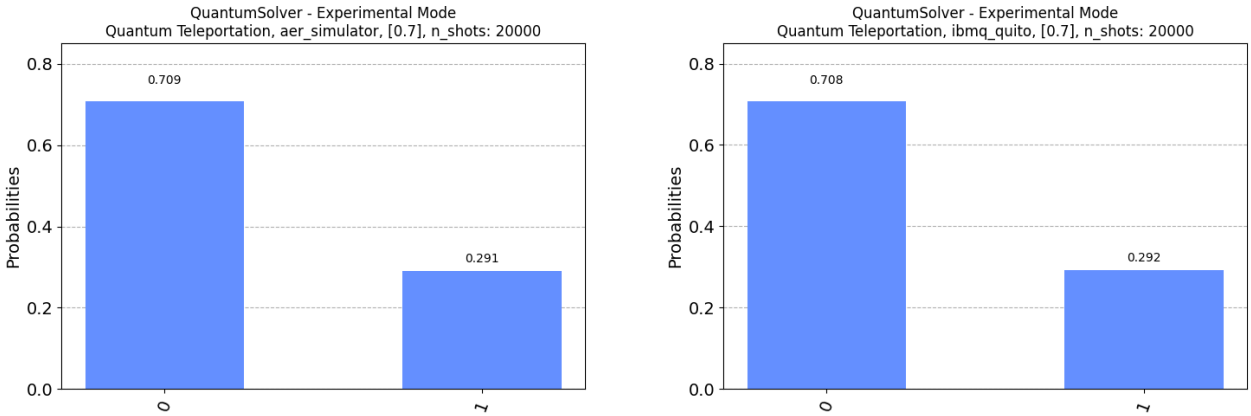


(a) Utilizando el backend “aer_simulator” (b) Utilizando el backend “ibmq_quito”

Figura 4.4: Histograma tras 20.000 ejecuciones del algoritmo de Grover

4.5. Teletransporte cuántico

En esta ocasión, podemos observar resultados prácticamente idénticos en ambos casos de ejecución (Fig. 4.5a y Fig. 4.5b). Esto es debido a las consideraciones vistas en su implementación, para adecuarlo al *hardware* de IBM (véase Sección 3.5).



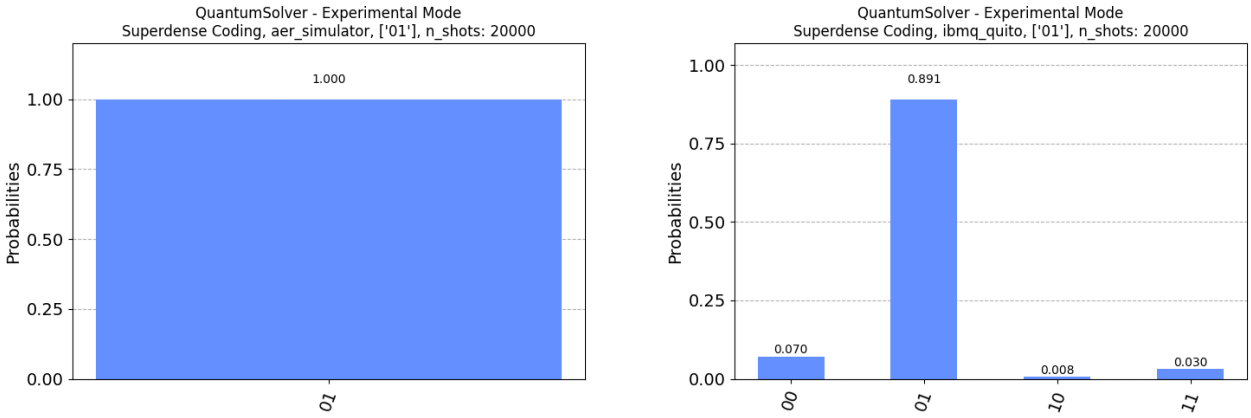
(a) Utilizando el backend “aer_simulator” (b) Utilizando el backend “ibmq_quito”

Figura 4.5: Histograma tras 20.000 ejecuciones del algoritmo de teletransporte cuántico

Se ha realizado el teletransporte, de manera exitosa, de un cúbit con un $70\,\%$ de probabilidad de medirse en 0 y un $30\,\%$ de determinarse en 1. Al realizar el experimento un número considerablemente alto de veces, en este caso 20.000, se puede observar como la probabilidad de obtener estos valores se ha codificado correctamente.

4.6. Protocolo de codificación superdensa

Por último, utilizando el protocolo de codificación superdensa se trata de transmitir el valor "01", utilizando un único cúbit de comunicación. De nuevo, el simulador obtiene éxito en todas las repeticiones de la ejecución mientras que las ejecuciones practicadas sobre el *hardware* de IBM obtienen un $89{,}1\,\%$ de acierto (Fig. 4.6), logrando unos resultados muy similares a los que obtuvo en la ejecución del algoritmo de Grover. Esto es por lo comentado en los resultados del algoritmo de de Bernstein-Vazirani: ganan mayor probabilidad, aquellos valores que distan un solo bit del deseado. Si los bits a transmitir son 01, serán más probables también 00 y 11.



(a) Utilizando el backend "aer_simulator"

(b) Utilizando el backend "ibmq_quito"

Figura 4.6: Histograma tras 20.000 ejecuciones del protocolo de codificación superdensa

Capítulo 5

Implementación del protocolo BB84

El protocolo criptográfico BB84 [30] para la distribución de claves cuánticas ha sido implementado gracias a los componentes de la librería *QuantumSolver*.

5.1. Entidades

Se ha implementado una entidad principal (*Participant*) y sus entidades derivadas (*Sender* y *Receiver*). Una instancia de la clase *Sender* es capaz de comunicarse con otra de la clase *Receiver*, de forma que la comunicación es secreta gracias al protocolo BB84, que permite la generación de una libreta de un solo uso que únicamente comparten emisor y receptor. La simulación del canal cuántico se describe mediante circuitos cuánticos de Qiskit [31]. La única diferencia entre las entidades *Sender* y *Receiver* es que la primera tiene un método para enviar un mensaje (inicializando un circuito cuántico) y la segunda para recibirlo (añadiendo la fase de medición al circuito). La clase base *Participant* contiene los métodos para la generación y muestra de valores, ejes, claves y libretas de un solo uso, entre otros.

5.2. Fundamento

La entidad emisora escoge al azar valores para codificar cada uno de los cúbits a transmitir y ejes en los que codificarlos, resultando las posibilidades indicadas en la Tabla 5.1.

Tabla 5.1: Posibilidades valor - eje y circuitos asociados

Valor	Eje	Circuito
$\|0\rangle$	Z	$q:$ ——
$\|0\rangle$	X	$q:$ —H——
$\|1\rangle$	Z	$q:$ —X——
$\|1\rangle$	X	$q:$ —X—H——

A continuación, la entidad receptora recibe esos circuitos (cada uno representando un cúbit) y, de manera aleatoria, elige ejes en los que medirlos. Aproximadamente el 50 % de los cúbits serán medidos correctamente, es decir, en el mismo eje que el emisor. El restante 50 % deberá ser descartado dado que al medirlos en el eje equivocado, se tendrá exactamente un 50 % de probabilidad de emitir el valor codificado correcto, lo que implica la pérdida de la correspondiente información. Para saber cuáles son los valores a descartar, ambas entidades hacen públicos los ejes en los que se midieron los cúbits, dado que no hay riesgo al realizar tal acción. Así desechan aquellos valores en los que los ejes no coinciden.

Los valores resultantes tras los descartes podrían ser considerados la clave generada, pero antes hay que verificar su seguridad. En el caso en el que un atacante intermedio haya interceptado los cúbits, se da una incoherencia entre la clave enviada por el emisor y la recibida por el receptor medida con el eje correcto, en aproximadamente el 50 % de ellos [32]. En la Tabla 5.2 se ilustran los 4 casos posibles de mediciones de cúbits, todos con un 25 % de probabilidad de ocurrir.

Tabla 5.2: Casos posibles de las mediciones de un cúbit en la fase de verificación de BB84

Medición de emisor y receptor legítimos	Medición del atacante intermedio	Conclusión sobre ese cúbit
Distinto eje	Eje del emisor	Es desechado en la fase de descarte de los valores, aunque el receptor intermedio lo haya medido con un 100 % de probabilidad de obtener valor correcto
Distinto eje	Eje del receptor	Es desechado en la fase de descarte de los valores, aunque el receptor intermedio lo haya medido con un 50 % de probabilidad de obtener valor correcto (total incertidumbre del valor)
Mismo eje	Mismo eje que ambos	Ha sido interceptado por el atacante sin que se aborte el protocolo, con un 100 % de probabilidad de que el valor final sea correcto
Mismo eje	Eje contrario a ambos	Hay un 50 % de probabilidad de abortar el protocolo, en caso de que en el emisor colapse con el valor contrario al emitido por el emisor

En el último caso de la Tabla 5.2 se observa que, con una probabilidad del 50 %, se aborta el protocolo por detectar la comunicación comprometida. Teniendo en cuenta que cada uno de los cuatro casos tiene una probabilidad del 25 %, cuando se envía un único cúbit el protocolo es abortado con una probabilidad $p_{detectar}$ del 12.5 %. Cuando se envían n cúbits, dado que al detectar alguna incoherencia el protocolo se aborta, la probabilidad de declarar la comunicación insegura se deduce de la de no detectar ningún ataque, a partir de la Ec. (5.1).

$$1 - (1 - p_{detectar})^n = 1 - \left(1 - \frac{1}{4} * \frac{1}{2}\right)^n = 1 - \left(\frac{7}{8}\right)^n \tag{5.1}$$

Este valor también se puede obtener aplicando el Teorema de Bayes con las probabilidades de no detectar ataque sobre cúbit no desechado ($\frac{3}{4}$), y sobre cúbit desechado

(1).

Para verificar el resultado del protocolo, el receptor publica la mitad de la clave obtenida. De esta manera, el emisor puede compararla con la correspondiente mitad de su clave y, si ambas coinciden, la ejecución del protocolo se considera segura, pudiendo utilizar la mitad restante de la clave generada como clave secreta compartida. En caso contrario, se deduce que alguna entidad intermedia ha lanzado un ataque de escucha secreta o *eavesdropping*, interceptando los cúbits enviados y midiéndolos antes de remitirlos al receptor legítimo.

```
## Generate a key for Alice and Bob
def __generate_key(self, backend, original_bits_size, n_bits, verbose):
  # Encoder Alice
  alice = Sender('Alice', original_bits_size)
  alice.set_values()
  alice.set_axes()
  message = alice.encode_quantum_message()

  # Interceptor Eve
  eve = Receiver('Eve', original_bits_size)
  eve.set_axes()
  message = eve.decode_quantum_message(message, self.measure_density, backend)

  # Decoder Bob
  bob = Receiver('Bob', original_bits_size)
  bob.set_axes()
  message = bob.decode_quantum_message(message, 1, backend)

  # Alice - Bob Remove Garbage
  alice_axes = alice.axes # Alice share her axes
  bob_axes = bob.axes # Bob share his axes

  # Delete the difference
  alice.remove_garbage(bob_axes)
  bob.remove_garbage(alice_axes)

  # Bob share some values of the key to check
  SHARED_SIZE = round(0.5 * len(bob.key))
  shared_key = bob.key[:SHARED_SIZE]

  # Alice check the shared key
  if alice.check_key(shared_key):
    shared_size = len(shared_key)
    alice.confirm_key(shared_size)
    bob.confirm_key(shared_size)

  return alice, bob
```

5.3. Ejecución y resultados

La librería *QuantumSolver* permite la ejecución de una implementación realizada del protocolo BB84. Al ejecutar el programa, se despliega un menú que facilita la visualización y selección del *backend* disponible. Tras elegirlo, se dan dos opciones al usuario. Por una parte, puede correr el algoritmo una única vez y visualizar la traza entre los diferentes participantes en la comunicación, para lo que es necesario especificar una cadena de caracteres como mensaje y un valor entre 0 y 1 como densidad de intercepción (la probabilidad con la que el receptor intermedio medirá cada cúbit).

En las Fig. 5.1 y Fig. 5.2 se puede apreciar una misma traza en la que se genera una libreta de un solo uso (al no existir intercepción en la comunicación) suficiente para enviar el mensaje *“secret”*.

```
BB84 SIMULATOR (Guest Mode)
===========================

[1] See available Backends
[2] Select Backend
      Current Backend: aer_simulator
[3] Run Algorithm
[4] Experimental mode
[0] Exit

[&] Select an option: 3
[&] Message (string): secret
[&] Interception Density (float between 0 and 1): 0
[i] Running BB84 simulation in aer_simulator with message "secret" and density "0.0"
Alice Values:
[1, 0, 1, 1, 1, 0, 0, 1, 1, 1, 0, 1, 1, 1, 0, 1, 1, 0, 0, 1, 0, 1, 0, 0, 1, 1, 1, 1, 0, 1, 0, 1, 1, 0, 0, 1, 1, 0, 1, 0, 1, 0, 1, 0, 1, 1, 0, 1, 0, 1, 1, 1, 1, 1, 0, 0,
 0, 0, 1, 0, 1, 0, 0, 0, 1, 0, 1, 0, 0, 0, 1, 1, 1, 0, 0, 1, 1, 0, 1, 0, 0, 1, 1, 1, 1, 1, 0, 1, 1, 0, 1, 1, 1, 1, 0, 1, 0, 1, 1, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0, 1, 0, 0, 1, 0, 1,
 0, 0, 0, 0, 1, 0, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 1, 1, 0, 1, 1, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 0, 0, 1, 0, 0, 1, 1, 1, 1, 1, 1, 0, 1, 0, 1, 0, 0, 0, 0, 0,
 1, 0, 1, 0, 0, 1]

Alice Axes:
[1, 0, 1, 1, 1, 0, 1, 0, 1, 1, 1, 0, 0, 0, 0, 0, 1, 0, 0, 1, 1, 0, 1, 1, 0, 0, 0, 0, 0, 1, 0, 0, 1, 1, 1, 1, 0, 1, 0, 1, 1, 0, 0, 1, 0, 0, 1, 0, 0, 1, 1, 0, 1, 0, 1, 0, 0, 0, 0, 0,
 1, 0, 1, 1, 0, 0, 1, 0, 1, 0, 1, 0, 0, 0, 1, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 1, 0, 0, 1, 0, 0, 0, 0, 0, 0, 1, 1, 0, 1, 0, 1, 1, 1, 1, 0, 0, 1, 1, 0, 1, 1, 1, 1, 1, 0, 1, 1, 0, 0,
 0, 0, 0, 1, 0, 0, 0, 1, 0, 1, 0, 0, 1, 1, 1, 0, 1, 0, 1, 1, 1, 1, 1, 1, 0, 0, 1, 0, 0, 0, 0, 0, 0, 1, 0, 0, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 1, 0, 0, 0, 1,
 1, 1, 0, 0, 0, 1]

Eve Values:
[-1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -
1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1,
 -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -
1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1,
 -1, -1, -1, -1, -1, -1]

Eve Axes:
[1, 0, 1, 1, 0, 1, 0, 1, 0, 1, 0, 1, 1, 0, 1, 0, 0, 0, 1, 0, 1, 0, 0, 0, 0, 0, 1, 1, 1, 0, 0, 1, 1, 1, 1, 0, 0, 1, 0, 1, 1, 1, 0, 1, 1,
 1, 1, 1, 1, 0, 0, 1, 0, 1, 1, 1, 1, 1, 0, 0, 1, 0, 0, 1, 0, 1, 0, 1, 0, 0, 1, 1, 1, 1, 1, 0, 0, 1, 0, 1, 1, 1, 1, 0, 1, 0, 1, 1, 0, 0, 1, 0, 0, 0, 0, 1, 1,
 1, 1, 0, 1, 1, 1, 1, 1, 1, 0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 1, 0, 0, 1, 1, 1, 0, 0, 1, 1, 1, 0, 0, 1, 0, 0, 1, 0, 1, 0, 1, 0, 1, 1, 1, 1, 0, 0, 0, 0, 0, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 1, 1,
 1, 0, 1, 0, 0, 1]
```

Figura 5.1: Inicio del protocolo BB84

```
Bob Values:
[1, 1, 0, 1, 0, 1, 1, 1, 1, 0, 1, 1, 1, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 1, 1, 0, 1, 1, 1, 0, 1, 0, 1, 1, 0, 1, 0, 1, 1, 1, 1, 1, 0, 1, 0, 1, 0, 1, 0, 0, 0, 0, 1, 1, 1, 1, 0, 0,
 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0, 1, 1, 1, 0, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 0, 1, 0, 0, 1, 1, 1, 0, 1, 1, 1, 0, 1, 0, 1, 1, 0, 0, 1, 0, 0, 0, 0, 1,
 0, 0, 0, 0, 1, 0, 0, 1, 1, 1, 1, 1, 0, 0, 1, 1, 0, 0, 0, 1, 0, 1, 1, 1, 1, 0, 1, 1, 0, 0, 1, 0, 0, 0, 0, 0, 1, 0, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0,
 0, 0, 1, 1, 0, 0, 1]

Bob Axes:
[1, 1, 0, 1, 0, 1, 0, 1, 1, 0, 0, 1, 0, 1, 0, 1, 0, 0, 1, 1, 1, 1, 0, 0, 0, 1, 1, 1, 0, 0, 1, 1, 0, 1, 0, 1, 0, 0, 0, 0, 1, 0, 1, 0, 1, 0, 0, 1, 0, 1, 0, 0, 0, 1, 1, 0, 1, 0, 0,
 1, 1, 1, 1, 0, 1, 1, 0, 1, 0, 0, 0, 1, 0, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 1, 1, 1, 0, 1, 1, 0, 0, 1, 1, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1,
 0, 0, 0, 1, 0, 1, 1, 0, 0, 1, 1, 1, 0, 1, 1, 0, 1, 1, 1, 0, 1, 0, 1, 1, 0, 0, 0, 1, 1, 0, 0, 0, 1, 1, 1, 1, 1, 1, 0, 1, 0, 1, 0, 1, 0, 0, 1, 0, 0, 1, 0, 1, 1, 1, 1, 0, 0,
 0, 0, 1, 0, 0, 1]

Alice Key:
[1, 1, 1, 1, 0, 0, 0, 1, 0, 1, 1, 0, 0, 1, 0, 0, 1, 1, 0, 1, 0, 1, 0, 1, 0, 1, 1, 0, 0, 0, 1, 0, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 0, 1, 1, 0, 0, 1, 0, 1, 0, 0, 0, 0, 1, 1, 1,
 1, 1, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 1, 1, 1, 0, 1, 0, 1, 1, 1, 0, 1, 1, 0, 0, 0]

Bob Key:
[1, 1, 1, 1, 0, 0, 0, 1, 0, 1, 1, 0, 0, 1, 0, 0, 1, 1, 0, 1, 0, 1, 0, 1, 0, 1, 1, 0, 0, 0, 1, 0, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 0, 1, 1, 0, 0, 1, 0, 1, 0, 0, 0, 0, 1, 1, 1,
 1, 1, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 1, 1, 1, 0, 1, 0, 1, 1, 1, 0, 1, 1, 0, 0, 0]

Shared Bob Key:
[1, 1, 1, 1, 0, 0, 0, 1, 0, 1, 1, 0, 0, 1, 0, 0, 1, 1, 0, 1, 0, 1, 0, 1, 0, 1, 1, 0, 0, 0, 1, 0, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 0, 1]

Final Keys

Alice Key:
[1, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 0, 1, 1, 1, 1, 1, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 1, 1, 1, 0, 1, 0, 1, 1, 1, 0, 1, 1, 0, 0, 0]

Bob Key:
[1, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 0, 1, 1, 1, 1, 1, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 1, 1, 1, 0, 1, 0, 1, 1, 1, 0, 1, 1, 0, 0, 0]

Secure Communication!
```

Figura 5.2: Verificación de la clave generada en el protocolo BB84

En la Fig. 5.3 se determina la comunicación segura, por lo que los participantes de la comunicación comparten la libreta de un solo uso. El emisor transmite su mensaje codificado con la total confianza de que solo el receptor legítimo será capaz de decodificarlo.

```
Secure Communication!

Alice OTP:
[36, 16, 62, 8, 7, 23, 24, 0]

Bob OTP:
[36, 16, 62, 8, 7, 23, 24, 0]

Initial Message:
secret
Encoded Message:
Wu]zbc
💡 Decoded Message:
secret

✅ The initial message and the decoded message are identical
✔ Running BB84 simulation in aer_simulator with message "secret" and density "0.0"
  BB84 simulation runned in 129.6684741973877 ms
```

Figura 5.3: Codificación y decodificación del mensaje en el protocolo BB84

En la Fig. 5.4 no se llega a generar la libreta de un solo uso, dado que se encuentran las discrepancias entre los inicios de las claves generadas y se detecta la intercepción ocurrida.

```
Alice Key:
[1, 1, 0, 1, 0, 1, 0, 1, 0, 0, 0, 0, 1, 1, 1, 1, 0, 1, 1, 0, 0, 1, 0, 0, 0, 1, 1, 0, 1, 1, 0, 1, 0, 0, 1, 0, 0, 1, 0, 1, 0, 0, 1, 1, 0, 0, 1, 1, 1, 1, 1, 1, 0, 1, 1, 0, 1, 1,
 0, 1, 1, 0, 0, 0, 0, 1, 0, 1, 0, 0, 1, 0, 1, 0, 1, 0, 1, 0, 0, 1, 1, 1, 0, 0, 0, 0, 1, 1, 1, 0, 1, 0, 0, 0, 1, 0, 1, 0, 1, 1, 0, 0, 1, 1, 0, 1, 0, 1, 1, 1, 1, 0, 1, 0, 1, 0, 1,
 0, 1, 1, 0, 0, 1, 0, 1, 0, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 0, 0, 0, 0]

Bob Key:
[1, 0, 0, 1, 0, 0, 0, 0, 0, 1, 1, 0, 0, 1, 1, 1, 0, 0, 1, 1, 0, 1, 0, 0, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 1, 1, 0, 0, 1, 0, 1, 1, 0, 1, 1, 0, 1, 1,
 0, 1, 0, 1, 1, 1, 0, 0, 0, 1, 0, 0, 1, 0, 0, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 0, 0, 0, 1, 1, 1, 0, 0, 1, 0, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 1, 0, 0, 0, 1, 1, 1, 1, 0, 1, 0, 1, 1, 1,
 0, 1, 1, 1, 1, 0, 0, 1, 1, 1, 1, 1, 0, 1, 1, 1, 0, 1, 0, 0, 0, 1, 0, 1, 0, 0]

Shared Bob Key:
[1, 0, 0, 1, 0, 0, 0, 0, 0, 1, 1, 0, 0, 1, 1, 1, 0, 0, 1, 1, 0, 1, 0, 0, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 1, 1, 0, 0, 1, 0, 1, 1, 0, 1, 1, 0, 1, 1,
 0, 1, 0, 1, 1, 1, 0, 0, 0, 1, 0, 0]

Unsecure Communication! Eve has been detected intercepting messages

❌ Message not send
✔ Running BB84 simulation in aer_simulator with message "intercept" and density "1.0"
  BB84 simulation runned in 351.470947265625 ms
```

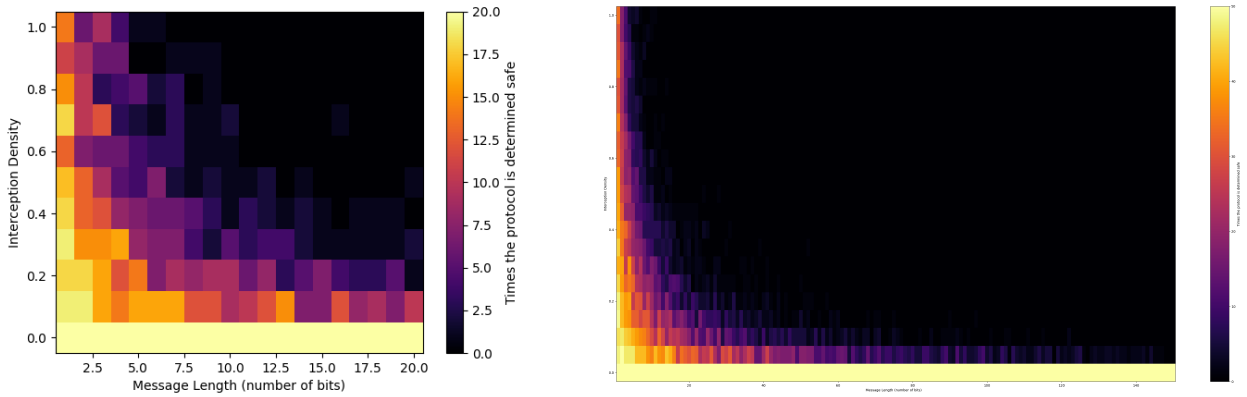
Figura 5.4: Intercepción detectada en el protocolo BB84

Por otra parte, el usuario puede ejecutarlo varias veces y mostrar un mapa de calor, que representa con colores más claros las condiciones en que ha habido más ocasiones en las que la comunicación ha sido considerada segura, y en colores más oscuros las comunicaciones en las que se han detectado intercepciones. Para este modo experimental, se deben especificar diferentes condiciones: la longitud máxima del mensaje en número de bits, que definirá el eje x en el mapa de calor, este tomará valores enteros positivos hasta ese máximo; el valor del *step* de la densidad de intercepción, que definirá el eje y del mapa de calor, este tomará $\frac{1}{step}$ valores entre 0 y 1; y el número de repeticiones para cada instancia generada del problema.

La Fig. 5.5 se muestran dos ejemplos de mapas de calor generados, correspondientes a los parámetros de la Tabla 5.3. Ilustran dos casos del modo experimental, uno de ellos más sencillo y rápido de computar; y otro más detallado y lento de procesar.

Tabla 5.3: Parámetros relevantes de los mapas de calor

Figura	Longitud máxima del mensaje	*Step* de densidad de intercepción	Repeticiones de cada instancia	Tiempo de ejecución (aprox.)
5.5a	20 bits	0.1	20	5 minutos
5.5b	150 bits	0.05	50	960 minutos



(a) Primer ejemplo (b) Segundo ejemplo

Figura 5.5: Mapas de calor de BB84 generados

En las Fig. 5.6 y Fig. 5.7 se muestra la interfaz de ejecución de este modo expermiental, tras la selección del *backend* deseado y el establecimiento de los parámetros exigidos.

```
BB84 SIMULATOR (Guest Mode)
===========================

[1] See available Backends
[2] Select Backend
        Current Backend: aer_simulator
[3] Run Algorithm
[4] Experimental mode
[0] Exit

[&] Select an option: 4
[&] Specify maximum message length (number of bits): 10
[&] Specify density step: 0.2
[&] Specify number of repetitions for each instance: 10

Running BB84 Simulator Experiment (in aer_simulator):
|                                  |  457/600 [76%] in 10s (45.7/s, eta: 3s)
```

Figura 5.6: Barra de carga del modo experimental del protocolo BB84

```
Running BB84 Simulator Experiment (in aer_simulator):
|                                  | 600/600 [100%] in 15.4s (38.88/s)

[$] Experiment Finished in 15.503013372421265 s!

💡 Output:

x: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

y: [0.0, 0.2, 0.4, 0.6000000000000001, 0.8, 1.0]

Image:
[[10. 10. 10. 10. 10. 10. 10. 10. 10. 10.]
 [10.  9.  6.  9.  7.  5.  3.  6.  5.  2.]
 [ 8.  8.  6.  4.  3.  3.  1.  2.  1.  2.]
 [ 8.  6.  1.  4.  7.  2.  1.  1.  0.  1.]
 [ 9.  2.  2.  0.  1.  0.  0.  0.  0.  0.]
 [ 7.  2.  2.  1.  1.  0.  0.  0.  0.  0.]]
```

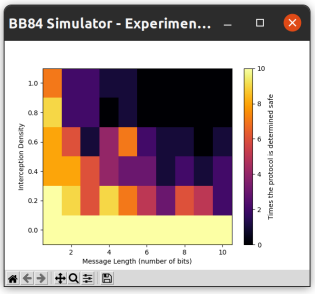


Figura 5.7: Resultados del modo experimental del protocolo BB84

Capítulo 6

Conclusiones y líneas futuras

La librería *QuantumSolver* para desarrollo cuántico permite, de manera sumamente accesible, la ejecución de algoritmos en *hardware* cuántico real y simuladores proporcionados por IBM. Los resultados obtenidos en la ejecución de todos los algoritmos implementados en la librería cumplen correctamente con las expectativas. Merece destacar que las dos interfaces desarrolladas juegan un papel fundamental para facilitar el uso del software dependiendo del ámbito de uso. La interfaz CLI aporta claridad para su ejecución y contribución, mientras que la interfaz web brinda facilidad en la ejecución para el público general. La comodidad en el uso de la interfaz gráfica radica, en gran parte, en su diseño minimalista, que permite la ejecución de las funcionalidades generales de manera intuitiva. Además, cuenta con el modo invitado, en el que se suprime la presencia de la dificultad del registro en los servicios de IBM Quantum.

Destacan tres posibles líneas de acción futuras: acceso, adición y visualización.

Para facilitar aún más el acceso al software, se podría realizar un lanzamiento en una web pública de la interfaz correspondiente. Del mismo modo, se podría publicar el módulo de Python en PyPi, consiguiendo una mejora en el proceso de descarga e instalación.

Por otra parte, la simplicidad en el proceso de adición de nuevos algoritmos a la librería se debe a la sencilla arquitectura de entidades usada, respaldada por una batería de pruebas unitarias. Precisamente, un importante objetivo de este proyecto ha sido facilitar la posible continuación del desarrollo mediante la ampliación del número de algoritmos disponibles. De hecho, gracias a los componentes de ejecución y algoritmia ofrecidos en *QuantumSolver*, se pueden implementar simulaciones más complejas, como la realizada del protocolo de criptografía cuántica BB84. Algunas líneas futuras relacionadas con la implementación de protocolos cuánticos criptográficos son el desarrollo, utilizando las herramientas disponibles en el *toolset*, de los protocolos B92 y E91.

Por último, para mejorar la visualización de resultados incrementando su posible impacto, se podría crear un entorno web, o mejorar el ya existente, realizando el desarrollo de simuladores de experimentos *ad hoc* basándose en los algoritmos que componen la librería. Un ejemplo de esto sería la creación de una página, dentro del entorno web, que simulara el lanzamiento de una moneda. Para ello deberá de llamar al algoritmo *QRand* de *QuantumSolver* con un único cúbit, y asignar cada uno de los estados devueltos a los estados de la moneda. Se propone también la creación de animaciones visuales durante el proceso de ejecución como, en este caso, la moneda girando hasta que se defina el resultado. Implementaciones similares podrían ser desarrolladas para el lanzamiento de un dado, la elección de una carta, la elección de colores, la elección de un elemento dentro de una lista de nombres ofrecida por el usuario, etc. Para otros algoritmos presentes en la librería, como el de Bernstein-Vazirani, se propone la implementación de un divertido

pasatiempo al estilo del popular juego de palabras *Wordle*. De la misma manera, se pueden obtener multitud de procesos visuales y llamativos para el público general, que acerquen la computación cuántica al conjunto de la sociedad, despertando el interés en estas tecnologías.

Capítulo 7

Conclusions and future works

The *QuantumSolver* library for quantum development allows, in a very accessible way, the execution of algorithms in real quantum hardware and simulators provided by IBM. The results obtained in the execution of the algorithms implemented in the library comply correctly with the expectations. It is worth noting that the two interfaces developed play a fundamental role in the ease of use of the software depending on the scope of use of the software. The CLI interface achieves clarity of execution and contribution, and the web interface provides ease of execution for the general public. The ease of use of the graphical interface is largely due to its minimalist design, which allows the execution of the general functionalities in an intuitive way. In addition, it has a guest mode, which eliminates the difficulty of registering with IBM Quantum services.

Three possible future lines of action stand out: access, addition and visualization.

To further enable easy access to the software, a public web release of the corresponding interface could be made. Similarly, the Python module could be published on PyPi, thus improving the download and installation process.

On the other hand, the simplicity in the process of adding new algorithms to the library is due to its simple entity architecture, supported by a battery of unit tests. Precisely, an important goal of this project has been to facilitate its possible continuation by expanding the number of available algorithms. In fact, thanks to the execution and algorithm components offered in *QuantumSolver*, more complex simulations can be implemented, such as the one carried out on the BB84 quantum cryptography protocol. Some future lines related to the implementation of quantum cryptographic protocols are the development, using the tools available in the toolset, of B92 and E91 protocols.

Lastly, to improve the visualization of results and their possible impact, a web environment could be created, or the existing one could be improved, by developing experiment simulators *ad hoc* based on the algorithms that make up the library. An example of this would be the creation of a page, within the web environment, that simulates a coin toss. To do this, the *QRand* algorithm of *QuantumSolver* must be called with a single qubit, and each of the returned states must be assigned to the states of the coin. It is proposed to create visual animations during the execution process such as, in this case, the coin spinning until the result is defined. Similar implementations could be developed for the roll of a die, the choice of a card, the choice of colors, the choice of an element within a list of names offered by the user, etc. For other algorithms present in the library, such as the Bernstein-Vazirani algorithm, one could propose the implementation of an amusing pastime in the style of the popular word game *Wordle*. In the same way, a multitude of visual and eye-catching processes can be obtained for the general public, bringing quantum computing closer to society as a whole, awakening interest in these technologies.

Capítulo 8

Presupuesto

A continuación, se realiza una estimación de los costes del proyecto. Se estiman las horas utilizadas en la realización del mismo, así como el coste de componentes físicos utilizados.

8.1. Costes de personal

Tabla 8.1: Costes de personal

Tarea	Tiempo	Coste	Coste total
Investigación realizada sobre el marco teórico cuántico y el uso de las tecnologías necesarias	50 h	8€ / h	400€
Planificación del proyecto	10 h	8€ / h	80€
Programación de la librería	180 h	10€ / h	1800€
Generación de documentación interna y externa	10 h	9€ / h	90€
Obtención de resultados y redacción de la memoria	50 h	15€ / h	750€
Total	300 h		3120€

8.2. Costes de componentes

El equipo utilizado durante la realización del proyecto está valorado en, aproximadamente, 1000€ y tiene las siguientes características:

- Procesador: Intel(R) Core(TM) i7-7700 CPU @3.60GHz, 4 procesadores principales, 8 procesadores lógicos
- Tipo de sistema: PC basado en x64
- Memoria física instalada (RAM): 8,00 GB
- Factor de forma de la Memoria RAM: DIMM
- Nombre del SO: Ubuntu 20.04.2 LTS

8.3. Coste total

Tabla 8.2: Coste total

Presupuesto	Coste
Personal	3120€
Componentes	1000€
Total	4120€

Apéndice A

Artículos enviados a conferencias

A.1. VII Jornadas Nacionales de Investigación en Ciberseguridad JNIC (aceptado)

QuantumSolver: Librería para el desarrollo cuántico.
José Daniel Escánez-Expósito, Pino Caballero-Gil, Francisco Martín-Fernández.
Actas de las VII Jornadas Nacionales de Investigación en Ciberseguridad JNIC.
Bilbao. 27-29 de junio de 2022.

A.2. 20th International Conference on Security and Management SAM (aceptado)

QuantumSolver: A quantum tool-set for developers.
Jose Daniel Escanez-Exposito, Pino Caballero-Gil, Francisco Martin-Fernandez.
Proceedings of the 20th International Conference on Security and Management SAM (within the World Congress in Computer Science, Computer Engineering, and Applied Computing CSCE).
Las Vegas, USA. July 25-28, 2022.
Springer Nature.
Indexada en Computing Research and Education (CORE), con ranking C.
Indexada en CS Conference Rankings (0.83).
Indexada en GII-GRIN en Class WiP.

A.3. XVII Reunión Española sobre Criptología y Seguridad de la Información RECSI (enviado)

Evolución de la librería QuantumSolver para el desarrollo cuántico.
José Daniel Escánez-Expósito, Pino Caballero-Gil, Francisco Martín-Fernández.
XVII Reunión Española sobre Criptología y Seguridad de la Información RECSI.
Santander. 19-21 Octubre 2022.

Bibliografía

[1] E. Gibney, “Hello quantum world! Google publishes landmark quantum supremacy claim”, Nature, vol. 574, no. 7779, pp. 461-462, 2019 [Online]. Available: https://www.nature.com/articles/d41586-019-03213-z. [Accessed: 17-Apr-2022].

[2] L. Madsen et al., "Quantum computational advantage with a programmable photonic processor", Nature, vol. 606, no. 7912, pp. 75-81, 2022. Available: https://www.nature.com/articles/s41586-022-04725-x. [Accessed 17 April 2022].

[3] Google LLC, “Google Quantum AI”. [Online]. Available: https://quantumai.google/. [Accessed: 17-Apr-2022].

[4] IBM Research, “Quantum Computing - IBM Research”. [Online]. Available: https://research.ibm.com/quantum-computing. [Accessed: 17-Apr-2022].

[5] Amazon Inc., “Amazon Braket”. [Online]. Available: https://aws.amazon.com/es/braket/. [Accessed: 17-Apr-2022].

[6] A. G. Tudorache, V. I. Manta and S. Caraiman, “Implementation of the Bernstein-Vazirani Quantum Algorithm Using the Qiskit Framework”, Bulletin of the Polytechnic Institute of Iași. Electrical Engineering, Power Engineering, Electronics Section, 67(2), 31-40, 2021.

[7] A. Warke, B. K. Behera and P. K. Panigrahi, “Experimental realization of three quantum key distribution protocols”, Quantum Information Processing, 19(11), 1-15, 2020.

[8] IBM, “Learn Quantum Computation using Qiskit” [Online]. Available: https://qiskit.org/textbook/preface.html. [Accessed: 17-Apr-2022].

[9] IBM, “Qiskit”. [Online]. Available: https://qiskit.org/. [Accessed: 17-Apr-2022].

[10] J. D. Escánez, “QuantumSolver - Project Board”. [Online]. Available: https://github.com/alu0101238944/quantum-solver/projects/1. [Accessed: 17-Apr-2022].

[11] J. D. Escánez, “QuantumSolver”. [Online]. Available: https://github.com/alu0101238944/quantum-solver/. [Accessed: 17-Apr-2022].

[12] J. D. Escánez, “QuantumSolver - Pull Requests”. [Online]. Available: https://github.com/alu0101238944/quantum-solver/pulls?q=is%3Apr+is%3Aclosed. [Accessed: 17-Apr-2022].

[13] J. D. Escánez, “QuantumSolver - Test”. [Online]. Available: https://github.com/alu0101238944/quantum-solver/tree/main/test. [Accessed: 17-Apr-2022].

[14] J. D. Escánez, “QuantumSolver - Readme”. [Online]. Available: https://github.com/alu0101238944/quantum-solver#readme. [Accessed: 17-Apr-2022].

[15] J. D. Escánez, “QuantumSolver - Documentation”. [Online]. Available: https://alu0101238944.github.io/quantum-solver/. [Accessed: 17-Apr-2022].

[16] IBM, “IBM Quantum”, May-2016. [Online]. Available: https://quantum-computing.ibm.com/. [Accessed: 17-Apr-2022].

[17] IBM, “User account - IBM Quantum”, May-2016. [Online]. Available: https://quantum-computing.ibm.com/composer/docs/iqx/manage/account/#account-overview. [Accessed: 17-Apr-2022].

[18] IBM, “Single qubit Gates” [Online]. Available: https://qiskit.org/textbook/ch-states/single-qubit-gates.html. [Accessed: 17-Apr-2022].

[19] D.Deutsch and R. Jozsa, “Rapid solution of problems by quantum computation”, Proceedings of the Royal Society of London. Series A: Mathematical and Physical Sciences, vol. 439, no. 1907, pp. 553-558, 1992. doi:10.1098/rspa.1992.0167.

[20] IBM, “Deutsch-Jozsa Algorithm” [Online]. Available: https://qiskit.org/textbook/ch-algorithms/deutsch-jozsa.html. [Accessed: 17-Apr-2022].

[21] IBM, “Phase Kickback” [Online]. Available: https://qiskit.org/textbook/ch-gates/phase-kickback.html. [Accessed: 17-Apr-2022].

[22] E. Bernstein and U. Vazirani, “Quantum Complexity Theory”, SIAM Journal on Computing, vol. 26, no. 5, pp. 1411-1473, 1997 [Online]. Available: https://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.655.1186&rep=rep1&type=pdf. [Accessed: 17-Apr-2022].

[23] IBM, “Bernstein-Vazirani Algorithm” [Online]. Available: https://qiskit.org/textbook/ch-algorithms/bernstein-vazirani.html. [Accessed: 17-Apr-2022].

[24] L. K. Grover, “A fast quantum mechanical algorithm for database search”, Proceedings of the twenty-eighth annual ACM symposium on Theory of computing - STOC ’96, 1996.

[25] IBM, “Grover’s Algorithm” [Online]. Available: https://qiskit.org/textbook/ch-algorithms/grover.html. [Accessed: 17-Apr-2022].

[26] W. Wootters and W. Zurek, “A single quantum cannot be cloned”, Nature, vol. 299, no. 5886, pp. 802-803, 1982.

[27] IBM, “Quantum Teleportation” [Online]. Available: https://qiskit.org/textbook/ch-algorithms/teleportation.html. [Accessed: 17-Apr-2022].

[28] IBM, “Quantum Teleportation - 5. Teleportation on a Real Quantum Computer” [Online]. Available: https://qiskit.org/textbook/ch-algorithms/teleportation.html#5.1-IBM-hardware-and-Deferred-Measurement-. [Accessed: 17-Apr-2022].

[29] IBM, “Superdense Coding” [Online]. Available: https://qiskit.org/textbook/ch-algorithms/superdense-coding.html. [Accessed: 17-Apr-2022].

[30] C. H. Bennett and G. Brassard, “Quantum cryptography: Public key distribution and coin tossing,” arXiv, 1984, doi: 10.48550/ARXIV.2003.06557. [Online]. Available: https://arxiv.org/abs/2003.06557 [Accessed: 17-Apr-2022].

[31] IBM, “Quantum Key Distribution” [Online]. Available: https://qiskit.org/textbook/ch-algorithms/quantum-key-distribution.html. [Accessed: 17-Apr-2022].

[32] W. Dür, S. Heusler, “What we can learn about quantum physics from a single qubit”, 06-Dec-2013. [Online]. Available: https://arxiv.org/pdf/1312.1463.pdf. [Accessed: 17-Apr-2022].