UNIVERSIDAD POLITÉCNICA DE CARTAGENA

Escuela Técnica Superior de Ingeniería de Telecomunicación

Eficiencia de Algoritmos Cuánticos para Solución de Problemas de Enrutamiento

TRABAJO FIN DE GRADO

GRADO EN SISTEMAS DE TELECOMUNICACIÓN

Autor: Pablo Antonio Martínez Vicente

Director: Javier Cerrillo Moreno

Codirector: Antonio Pérez Garrido

Cartagena, 15 de septiembre de 2023

Resumen

Este trabajo propone un algoritmo para hayar la ruta más eficiente en una red de nodos de telecomunicaciones. Este algoritmo habitualmente conocido como problema del vendedor ambulante, se puede resolver con un algoritmo cuántico de forma más eficiente. En un trabajo de grado anterior, se propone una modificación del algoritmo de Grover de forma heurísitica y además utilizar una codificación novedosa con la intención de hacer más eficiente el algoritmo. El trabajo ha consistido en desarrollar esta idea e implementar el algoritmo para luego comprobar su funcionamiento en un número concreto de nodos, para este caso cinco.

Para obtener dicho fin, se comenzó con la optimización de una parte del algoritmo de Grover, conocido como oráculo. Para ello primero se ha desarrollado una implementación en un ordenador clásico en código python, luego se ha ido transformando parte a parte a código qiskit, el cuál es una herramienta proporcionada por IBM para el manejo de computación cuántica a nivel de circuitos. El algoritmo de Grover requiere transformar los costes en fases cuánticas, que van de 0 a 2π, para eso hemos necesitado también desarrollar un programa que transforme un conjunto de costes a una distribución de fases.

Tras tener implementado todo el algoritmo se puso en marcha y se testeó en diversas ocasiones para ver la capacidad de resolver un problema real, se hizo tanto en un ordenador cuántico simulado como en un ordenador cuántico real. Para implementar estas simulaciones ha sido necesario modificar la implementación convencional para que fuese capaz de poder trabajar con esta nueva codificación. Una vez implementado, se observó que en el 75 % de los casos la ruta de mayor probabilidad era también la de menor costo cuando las pruebas se hacían en un ordenador cuántico simulado. Cuando estas implementaciones se hicieron en un ordenador cuántico real, los resultados diferían de los vistos en un ordenador simulado, fallando a la hora de encontrar la ruta de menor costo. La conclusión es que el sistema opera adecuadamente en un entorno ideal. Sin embargo, al enfrentarse a las limitaciones de los prototipos utilizados en este estudio, el algoritmo no muestra eficacia.

Índice

1. Introducción

Para desarrollar una implementación que resuelva el problema propuesto, fue esencial estudiar y consolidar diversos conceptos. Esto incluye tanto la comprensión profunda del problema en sí, como los fundamentos de la computación cuántica. Además, también paso a exponer la codificación que se empleó cuyo objetivo era el de optimizar la eficiencia de la implementación.

1.1. Problema del vendedor ambulante

Consideremos un caso donde tenemos cinco nodos de telecomunicaciones por los cuales queremos transmitir un paquete de información, asegurándonos de que este pase una única vez por cada uno de ellos y, finalmente, regrese al nodo inicial de transmisión. Este tipo de problema es análogo al problema del vendedor ambulante (TSP por sus siglas en inglés).

Centrándonos ya en el TSP, supongamos que un vendedor tiene que visitar cuatro ciudades: A, B, C y D. Su objetivo es encontrar la ruta más corta que le permita visitar cada ciudad una sola vez y regresar al punto de partida, a este tipo de rutas se les llaman ciclos de Hamilton. Se le proporciona una tabla con las distancias entre cada par de ciudades a la que más adelante se referenciará como matriz de costes:

	A	**B**	**C**	**D**
A	-	10	15	20
B	10	-	35	25
C	15	35	-	30
D	20	25	30	-

Cuadro 1: Distancias entre ciudades para el TSP de la matriz de costes.

A simple vista, podemos ver que un posible recorrido sería A - B - C - D - A, que tiene un costo total de $10 + 35 + 30 + 20 = 95$. Pero este no es el recorrido más corto. La ruta más corta es A - B - D - C - A con un costo de $10 + 25 + 30 + 15 = 80$. Los TSP se pueden presentar en un formato gráfico, donde las ubicaciones se representan mediante nodos en el gráfico y las posibles rutas entre ellos mediante ejes, ver 1.

El TSP se vuelve mucho más complicado a medida que agregamos más ciudades, ya que el número de posibles caminos crece de forma factorial, como se puede ver en la siguiente expresión:

$$\frac{(N-1)!}{2} \tag{1}$$

donde N es el número de ciudades del problema. Aunque este es un ejemplo simple con solo cuatro ciudades, hay que imaginar lo que sería intentar encontrar la ruta más corta entre 10, 20 o 50 ciudades, la cantidad de posibles rutas serían $1,8 \times 10^5$, 6×10^{16} y 3×10^{62} respectivamente. Este tipo de problemas que aumentan de manera tan exagerada el número de posibles soluciones entre las que hay que buscar forman parte de un tipo de problemas que se llaman NP-complejo. NP es el acrónimo de *non-deterministic polynomial time*. Los problemas de esta clase tienen una característica distintiva: aunque encontrar una solución puede no ser eficiente, verificar si una solución dada es correcta sí se puede hacer en tiempo polinómico. Tomemos como ejemplo un grafo y la pregunta de si posee un ciclo hamiltoniano. Si se nos proporciona un ciclo específico, podemos determinar de manera eficiente (en tiempo polinómico) si es o no hamiltoniano. Así, este problema pertenece a NP. Sin embargo, la tarea de identificar tal ciclo en un grafo dado, si es que existe, es algo que no sabemos cómo lograr eficientemente. Es importante aclarar que todo problema en NP es también NP-hard, pero no necesariamente al revés. Los problemas NP-hard son aquellos que son, al menos, tan difíciles como los problemas más desafiantes de NP[3].

Además de su importancia teórica en informática y matemáticas, TSP tiene muchas aplicaciones prácticas y se utilizan en diferentes campos. Por ejemplo, en la planificación de la producción y la logística del transporte, donde puede ayudar a minimizar los costos de transporte y maximizar la eficiencia de las rutas. En este contexto, *vendedor* podría ser una entidad como un camión de reparto, y *ciudades* podría representar ubicaciones de entrega o recogida.

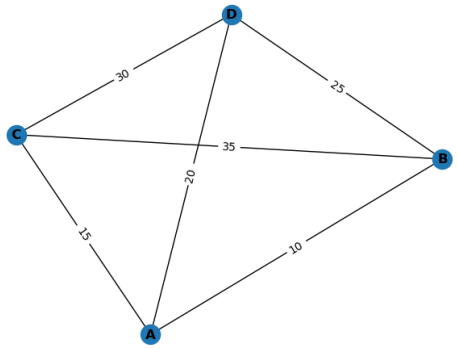

Figura 1: Imagen que representa un ejemplo de cuatro nodos (1,2,3,4) interconectados por una serie de caminos los cuales cada uno de ellos tienen costes determinados.

A pesar de la dificultad que supone encontrar el ciclo de Hamilton óptimo, se pueden utilizar una variedad de enfoques heurísticos, métodos exactos (como el método Branch & Bound[6]) y algoritmos basados en inteligencia artificial y computación evolutiva para encontrar una solución razonable en el tiempo de cálculo real [7]. Sin embargo, teniendo en cuenta la naturaleza NP-difícil de este problema, emerge la hipótesis de que podría haber una solución cuántica que superase en eficiencia a las técnicas clásicas previamente mencionadas. Es en este escenario donde se encuentra el principal impulso para el desarrollo del algoritmo cuántico propuesto en este trabajo.

1.2. Conceptos Clave de la Computación Cuántica

La computación cuántica ofrece una innovadora metodología para el procesamiento de información, divergiendo de los principios tradicionales que han fundamentado la informática desde sus comienzos. En contraste con las máquinas clásicas, que emplean bits como unidades fundamentales y operan en estados discretos de 0 o 1, la computación cuántica introduce una nueva dimensión en la representación y manipulación de la información gracias al empleo de qubits, los cuales pueden representar ambos estados simultáneamente gracias al fenómeno de la superposición cuántica. Esta capacidad permite a los ordenadores cuánticos explorar múltiples soluciones al mismo tiempo, otorgándoles un potencial significativo en ciertas tareas computacionales, como la factorización de números grandes o la búsqueda en grandes bases de datos. Se trata de una tecnología que ha experimentado grandes avances en las últimas décadas, desde propuestas teóricas hasta los primeros prototipos de ordenadores cuánticos. Por lo tanto es una tecnología prometedora para problemas de computación complejos como es este caso.

1.2.1. Ket

Un *ket* es una manera de describir un estado cuántico. Por lo tanto, es una forma de representar un estado en un espacio de Hilbert el cual puede ser expresado como combinación lineal de una base [8]. La notación usada es $|\psi\rangle$ donde ψ es el nombre del estado. Por ejemplo, los estados básicos de un qubit se representan típicamente como $|0\rangle$ y $|1\rangle$.

1.2.2. Qubit

Un qubit, o bit cuántico, representa la unidad fundamental de información en la computación cuántica, al igual que el bit lo es para la computación clásica. No obstante, mientras que un bit clásico puede estar en un estado de 0 o 1, un qubit puede encontrarse en una superposición coherente de estos estados [8]. Esta idea puede representarse matemáticamente como:

$$|\psi\rangle = \alpha\,|0\rangle + \beta\,|1\rangle\,, \tag{2}$$

donde $|0\rangle$ y $|1\rangle$ son los estados base del qubit, y α y β son coeficientes complejos, también conocidos como amplitudes de probabilidad, los cuales están sujetos a la condición

$$|\alpha|^2 + |\beta|^2 = 1. \tag{3}$$

La representación gráfica de un qubit se realiza a menudo utilizando la esfera de Bloch, ver 2. La esfera de Bloch, de radio unidad, permite visualizar cualquier estado $|\psi\rangle$ utilizando los ángulos θ y ϕ. θ es el ángulo entre el eje z y el vector del estado en la esfera de Bloch el cual varía entre 0 y π. φ por su parte es el ángulo en el plano xy, medido desde el eje x hacia el eje y. Varía entre 0 y 2π [8].

Usando estos dos ángulos, se puede describir cualquier estado cuántico de un qubit de la siguiente manera:

$$|\psi\rangle = \cos\left(\frac{\theta}{2}\right)|0\rangle + e^{i\varphi}\sin\left(\frac{\theta}{2}\right)|1\rangle\,. \tag{4}$$

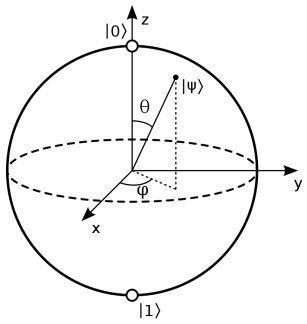

Figura 2: Representación de la esfera de Bloch.Fuente: www.wikipedia.com

Además, es crucial entender que los sistemas cuánticos pueden consistir en múltiples qubits. Un sistema de dos qubits, por ejemplo, tiene una representación de estado:

$$|\psi\rangle = \alpha|00\rangle + \beta|01\rangle + \gamma|10\rangle + \delta|11\rangle, \tag{5}$$

Aquí, $|00\rangle$, $|01\rangle$, $|10\rangle$ y $|11\rangle$ son estados base de dos qubits, y los coeficientes son números complejos que cumplen con la normalización.

La capacidad de manipular y entrelazar múltiples qubits es esencial para muchos de los algoritmos cuánticos avanzados. Sin embargo, el manejo de qubits, ya sean individuales o en conjunto, presenta retos significativos, como la decoherencia [10] y el ruido cuántico [8].

1.2.3. Circuito Cuántico

Un circuito cuántico es una rutina de cálculo que es capaz de realizar de forma efectiva varios calculos clásicos de manera simultánea [9]. Cualquier programa cuántico puede describirse mediante una serie de circuitos cuánticos y cálculos clásicos no simultáneos.

Los circuitos cuánticos típicamente se estructuran en tres fases: inicialización, aplicación de puertas lógicas y medición.

Inicialización: Antes de proceder con el cálculo cuántico, es crucial definir un estado cuántico adecuado. Esta etapa se logra empleando operaciones de inicialización y reinicio, comúnmente a través de puertas cuánticas como Hadamard, las cuales se detallarán posteriormente.

Puertas Cuánticas: En esta fase, se implementa una serie de puertas lógicas que constituyen el circuito, y que permiten efectuar las operaciones cuánticas deseadas.

Medición: Finalmente, una computadora clásica procesa las mediciones de los qubits para obtener resultados clásicos (0 o 1) que se almacenan en bits convencionales.

1.2.4. Puertas Lógicas Cuánticas

Las puertas lógicas cuánticas se encargan de efectuar operaciones sobre los qubits. Es esencial que estas puertas sean reversibles, lo cual deriva de la reversibilidad de la mecánica cuántica. En términos prácticos, esto implica que si aplicamos una puerta lógica cuántica a un qubit, modificando su estado, siempre existe otra puerta ya sea ella misma u otra distinta que devuelve el qubit al estado original. Estas operaciones cuánticas se representan matemáticamente mediante matrices $2^n \times 2^n$ donde n es el número de qubits implicados en dicha puerta lógica.

Las puertas lógicas más comunes y conocidas son las siguientes:

1. Puerta Hadamard (H): Esta es fundamental en la computación cuántica, ya que introduce el concepto de superposición. Al aplicar la puerta Hadamard a un qubit en estado base $|0\rangle$, se crea una superposición equilibrada de los estados $|0\rangle$ y $|1\rangle$. Lo que esto significa es que, tras la aplicación de esta puerta, hay una probabilidad igual de medir el qubit en estado $|0\rangle$ o $|1\rangle$.

2. Puerta S (o puerta Phase): Esta puerta agrega una fase al estado $|1\rangle$ de un qubit. Específicamente, introduce una fase de $\varphi = \frac{\pi}{2}$ al estado $|1\rangle$, sin afectar al estado $|0\rangle$.

3. Puerta Pauli-X: Funciona como una puerta NOT en la lógica clásica, pero para qubits. Invierte el estado de un qubit; transforma un $|0\rangle$ en $|1\rangle$ y viceversa.

En resumen:

Nombre	**Símbolo**	**Estado inicial**	**Estado final**	**Matriz unitaria**
Hadamard	H	$\|0\rangle$, $\|1\rangle$	$\frac{1}{\sqrt{2}}(\|0\rangle + \|1\rangle)$, $\frac{1}{\sqrt{2}}(\|0\rangle - \|1\rangle)$	$\frac{1}{\sqrt{2}}\begin{pmatrix}1 & 1\\ 1 & -1\end{pmatrix}$
X (NOT)	X	$\|0\rangle$, $\|1\rangle$	$\|1\rangle$, $\|0\rangle$	$\begin{pmatrix}0 & 1\\ 1 & 0\end{pmatrix}$
Phase	S	$\|1\rangle$	$i\|1\rangle$	$\begin{pmatrix}1 & 0\\ 0 & i\end{pmatrix}$

Cuadro 2: Resumen de las puertas cuánticas Hadamard, X y S.

Existen puertas lógicas cuánticas que actúan sobre más de un qubit. Especial mención a las puertas *CNOT* y *CPHASE*. Estas operan con un qubit de control y un qubit objetivo. Si el qubit de control se encuentra en el estado $|1\rangle$, entonces las puertas *CNOT* 0 *CPHASE* aplicarán al qubit objetivo. En cambio, si el qubit de control está en $|0\rangle$, no se aplicará nada.

1.3. Algoritmo de Grover

Explorar todas las posibles rutas o decisiones en este problema puede compararse con una situación sencilla: imagina tener una caja llena de esferas, donde cada una representa una posible solución. De todas estas esferas, solo una es la correcta. No obstante, identificarla de inmediato es una tarea complicada, ya que todas parecen similares a simple vista.

En este contexto, el algoritmo de Grover[4] actúa como una herramienta sofisticada diseñada para destacar la esfera correcta de entre todas las demás. En lugar de revisar cada esfera individualmente, lo que sería un proceso tedioso y lento, este algoritmo tiene la capacidad de guiarnos de manera más directa hacia la solución correcta.

1.3.1. Introducción al algoritmo

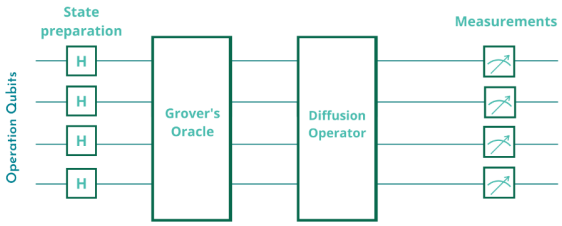

Figura 3: Representación de los tres bloques que conforman el algoritmo de Grover

Retomando el sencillo ejemplo, la esfera que buscamos tiene una característica distintiva, como puede ser un color diferente. A esta la denominaremos como la ganadora, por lo tanto la llamaremos $\boldsymbol{w}$ de *winner*. En un ordenador clásico, se requeriría en promedio $N/2$ iteraciones para encontrar la esfera ganadora. Esto ocurre porque, al buscar una por una, en algunos casos podríamos encontrarlo de inmediato, mientras que en otros podría requerir la revisión de todos los elementos. En contraste, con el algoritmo de Grover, se puede hallar la esfera ganadora en tan solo $\sqrt{N}$ iteraciones, lo que representa una mejora cuadrática. Aunque en computación cuántica existen algoritmos que ofrecen mejoras incluso exponenciales para ciertas tareas, para este tipo de problema, el algoritmo de Grover es el más adecuado.

El algoritmo de Grover se desglosa en tres componentes clave: el estado de preparación, el oráculo y el difusor, ver 3.

El estado de preparación genera un estado de superposición. Este efecto es posible gracias al empleo de las puertas de Hadamard. Tras el empleo de las puertas Hadamard, tendríamos todos los posibles estados con la misma probabilidad de medición. En nuestro ejemplo, sería tener la misma posibilidad de coger una esfera u otra. A este estado de superposición inicial lo llamaremos $|s\rangle$, donde $|x\rangle$ es cada uno de los estados que tenemos (cada esfera del conjunto) y N el número total de los posibles estados:

$$|s\rangle = \frac{1}{\sqrt{N}} \sum_{x=0}^{N-1} |x\rangle \,. \tag{6}$$

1.3.2. Oráculo y amplificador de Grover

Comenzamos con la explicación del oráculo. Es un componente clave para identificar y marcar la solución correcta en un conjunto de posibles soluciones. Volviendo al ejemplo práctico, el oráculo es

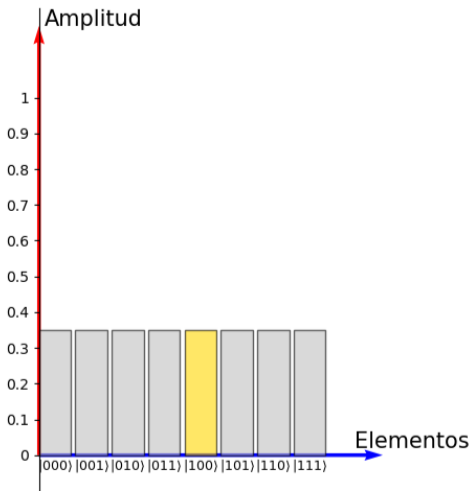

Figura 4: Gráfica que representa las amplitudes de probabilidad. Los elementos corresponden con todos los posibles estados que tenemos que en nuestro ejemplo que son 8. Podemos ver la representación de todos los estados.

la parte del algoritmo que es capaz de visualizar la característica propia del estado ganador (la esfera con el color distinto) y además es capaz de remarcarla.

Para entender mejor cómo funciona, podemos visualizarlo desde una perspectiva geométrica y desde la amplitud de probabilidad de los estados, para ello, vamos a pensar que nuestro conjunto de esferas son 8, por lo tanto se pueden numerar con 3 qubits, donde la esfera ganadora, es la que tiene el estado $|100\rangle$.

Es importante resaltar, que al principio todos los posibles estados tienen la misma amplitud de probabilidad, ver figura 4, como las amplitudes de probabilidad su módulo cuadrado deben sumar 1, es evidente que sus valores cuando generamos una superposición se saca de la siguiente fórmula:

$$\frac{1}{\sqrt{N}}, \tag{7}$$

donde N es el número de estados superpuestos.

Desde la perspectiva vectorial, hay dos vectores esenciales. El primero, denotado como $|w\rangle$, simboliza el objeto que estamos buscando. El segundo vector, $|s\rangle$, representa el estado de superposición inicial, creado utilizando las puertas de Hadamard, ver figura 5. En las etapas iniciales, cuando el objeto que queremos encontrar aún se encuentra en superposición y no se distingue de los demás, lo representamos con $|\psi\rangle$.

Sin embargo, a pesar de que los vectores $|w\rangle$ y $|s\rangle$ definen un espacio bidimensional, no son perpendiculares entre sí. Esto se debe a que el estado ganador, representado por el vector $|w\rangle$, todavía es parte de las posibilidades dentro del espacio de superposición $|s\rangle$, por lo tanto no pueden ser ortogonales.

Es por ello que se introduce un vector auxiliar, denotado como $|s'\rangle$ y perpendicular al vector del estado que estamos buscando, $|w\rangle$, ver 5. La relación entre estos vectores y el estado de superposición $|s\rangle$ se define como:

$$|s\rangle = \sin(\theta)\,|w\rangle + \cos(\theta)\,|s'\rangle\,, \quad \text{con} \quad \theta = \arcsin(\langle s|w\rangle) = \arcsin\left(\frac{1}{\sqrt{N}}\right). \tag{8}$$

La intervención del oráculo en este proceso consiste en aplicar una reflexión al estado objetivo. Esta

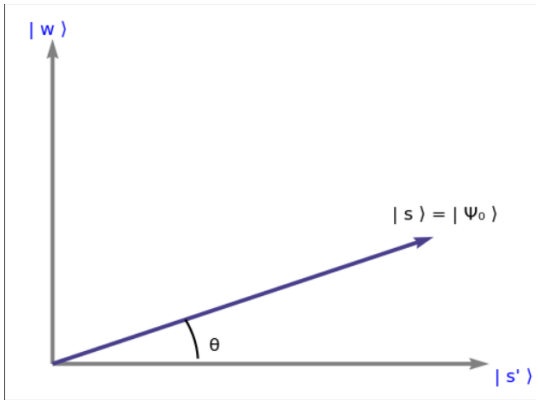

Figura 5: Espacio de dos dimensiones creado por los estados $|w\rangle$ y $|s'\rangle$.

acción se traduce en una reflexión de $|s\rangle$ respecto al vector $|s'\rangle$. El resultado de aplicar esta reflexión es obtener $|\psi_{t'}\rangle$, que viene de aplicar este oráculo al $|\psi_t\rangle$, ver 6.

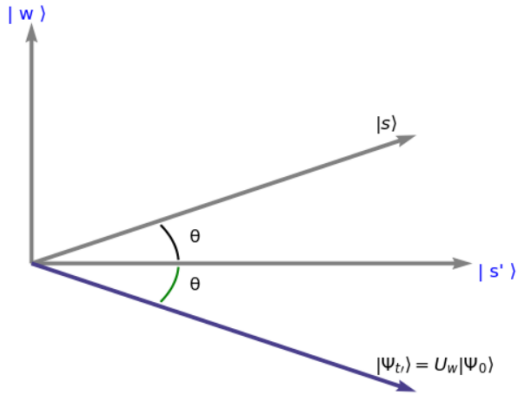

Figura 6: Reflexión respecto de $|s'\rangle$ realizada por el oráculo.

Una perspectiva equivalente, pero enfocada en las amplitudes de probabilidad, sería decir que la amplitud de probabilidad del estado ganador pasa a ser negativa después de la intervención del oráculo ver figura 7. Nuestro oráculo ha identificado y marcado efectivamente la esfera que se diferencia de las demás. Es importante mencionar que este marcaje, que involucra el cambio de signo de la amplitud de probabilidad, es intrínseco al oráculo y no afecta las probabilidades. Esto se debe a que al calcular el módulo cuadrado de estas amplitudes para determinar las probabilidades, el signo no tiene ningún impacto. El encargado de aprovechar esta marca para potenciar la probabilidad y así encontrar la esfera diferente es el amplificador.

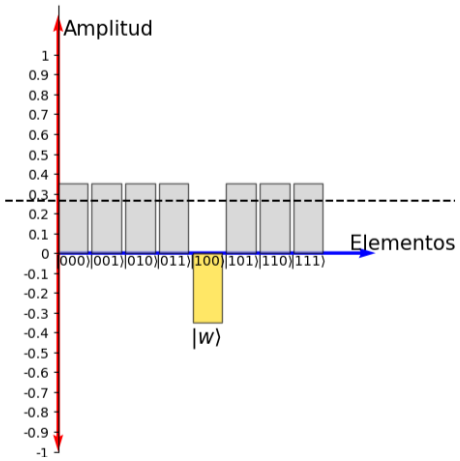

Figura 7: En la siguiente gráfica se demuestra como se acaba destacando al aplicar un menos al estado que estamos buscando, $|w\rangle$. Por otro lado se muestra una linea discuntinua de puntos que representa la media de las amplitudes de probabilidad, clave para el siguiente paso.

El amplificador consiste en la aplicación de la siguiente fórmula:

$$U_s = 2\,|s\rangle\,\langle s| - \mathbb{I}, \tag{9}$$

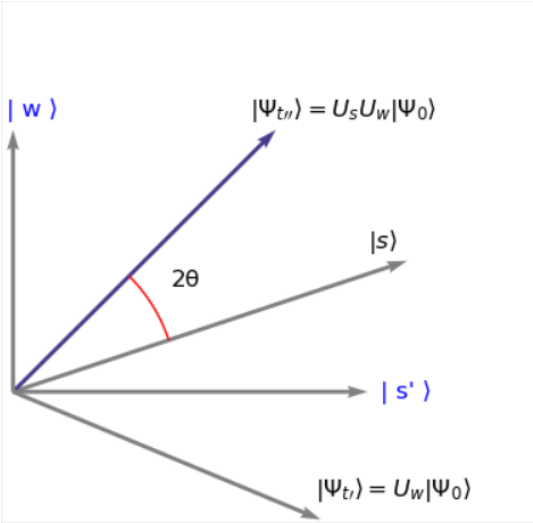

Figura 8: Esquema donde se demuestra que tras aplicar el amplificador vemos como $|\psi_{t''}\rangle$ está más cerca del estado ganador.

donde I es la matriz identidad y $\langle s|$ es la transposición conjugada de $|s\rangle$. La esencia de este paso es efectuar una reflexión del estado $|\psi_{t'}\rangle$ respecto al vector $|s\rangle$. Esto nos permite acercar el vector que estamos manipulando, $|\psi_{t'}\rangle$, al estado deseado, $|w\rangle$, ver 8.

Cuando lo examinamos desde la perspectiva de las amplitudes de probabilidad, aplicar la fórmula anterior provoca una reflexión en torno a la media de estas amplitudes, ver 9. Esto genera una disminución en la amplitud de los estados que no fueron afectados por el oráculo, mientras que el estado $|w\rangle$, que es nuestro objetivo, experimenta un incremento significativo en su amplitud.

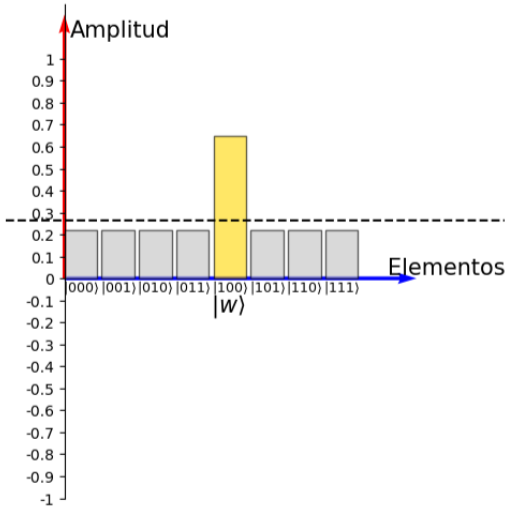

Figura 9: En la gráfica, observamos que el estado $|w\rangle$ posee la mayor amplitud. Esto ocurre porque el amplificador modifica las amplitudes, haciendo una rotación con respecto a la media. Como resultado, aquellos estados con amplitudes negativas se convierten en los más prominentes.

En este punto, hemos conseguido que la amplitud de probabilidad de $|w\rangle$ sea varias veces mayor que su valor inicial. Esto indica que conforme vayamos repitiendo las operaciones del oráculo y el amplificador como si fuese un ciclo, aumentamos la probabilidad de identificar el estado ganador, es decir, de seleccionar la esfera deseada. No obstante, es un error pensar que realizar más iteraciones siempre incrementará nuestra precisión o probabilidad de acierto. De hecho, si efectuamos demasiadas iteraciones, podríamos alejarnos del estado deseado $|w\rangle$. Es fácil visualizar este fenómeno observando la figura 8. Si aplicamos la rotación demasiado, nos excederemos, provocando que ahora los estados cerca del estado ganador no sean lo que estábamos buscando. Por ello, el número óptimo de iteraciones está dictado por una ecuación específica, tal como se detalla en la siguiente fórmula[2]:

$$\frac{\pi}{4}\sqrt{\frac{N}{t}}, \tag{10}$$

donde N es el número total de elementos y t el número de posibles resultados.

1.3.3. Planteamiento de un Grover heurístico.

A la hora de plantear el algoritmo no vamos a hacer uso directamente del algoritmo de Grover. Sino vamos a modificarlo, se tratará como un algoritmo cuántico heurístico.

La mayor diferencia que vamos a tratar en este algoritmo respecto al original va a residir en el oráculo. Mientras que antes el oráculo en el Grover original tenía la siguiente expresión[4]:

$$f(x) = \begin{cases} |x\rangle & \text{si } x \neq w \\ -|x\rangle & \text{si } x = w \end{cases} \tag{11}$$

El oráculo descrito en el artículo donde se planteaba la idea de un Grover heurístico [1] es el siguiente:

$$\hat{C}\,|T\rangle = e^{i\theta(T)}\,|T\rangle\,, \tag{12}$$

donde $|T\rangle$ representa el estado de todos los caminos posibles. Para entender $\theta(T)$, debemos volver a nuestro problema original. Estamos ante una serie de nodos interconectados por diversos caminos, cada uno con un coste asociado. Nuestro objetivo es identificar el camino con el menor coste. En este contexto, $\theta(T)$ va a ser una fase cuántica que van a tener todos los estados de posibles soluciones, tal que vamos a ser capaces de relaccionar dichas fases con los costes de cada uno de los caminos.

Es crucial determinar el rango de trabajo en relación con las fases. Al observar la conexión entre el heurístico y el clásico, resulta evidente que el signo negativo se puede interpretar como una exponencial compleja de fase π, mientras que los demás estados, aquellos que no son de interés, presentan una exponencial compleja de fase 0.

Por consiguiente, en este trabajo me planteo la idea de graduar las fases en función de la conveniencia de los estados o caminos: aquellos más favorables se aproximarán a π, mientras que los menos propicios, aquellos que implican un mayor costo o incluso los estados no contemplados que se abordarán más adelante, tendrán una fase más cercana a 0. En esencia, mientras el algoritmo de Grover originalmente introduce una fase solo al estado solución, el enfoque actual busca asignar fases a un amplio espectro de soluciones posibles. Se ajusta el oráculo de tal manera que los estados que denoten una ruta más ventajosa culminen con una fase cercana a π tras aplicar el oráculo. Como resultado, tras aplicar el amplificador, estos estados tendrán una probabilidad elevada de ser seleccionados.

Por otro lado, aunque en el estudio previamente citado sobre el enfoque heurístico de Grover se sugieren algunas modificaciones en el amplificador, he decidido explorar qué sucedería con un operador de Grover original. Este será uno de los aspectos que abordaremos en los resultados: el comportamiento del difusor ante diferentes fases en varios estados superpuestos. Uno de los objetivos propuestos es entender mejor cómo estas variaciones en las fases pueden influir en la eficacia del algoritmo de Grover en contextos prácticos.

1.4. Codificación Propuesta

Para poder trabajar con un número de caminos es inevitable traducir esta información para trabajar con ella a base de qubits. Surge entonces la necesidad de codificar todos los posibles caminos.

Basándonos en la codificación propuesta en el trabajo de grado previamente citado [5], buscamos minimizar el uso de qubits. El objetivo es prevenir un crecimiento factorial cuando se extienda a N ciudades. En lugar de enumerar directamente cada ciudad, la propuesta se centra en enumerar los saltos necesarios para desplazarse de una ciudad a otra. El término saltos se refiere a cuantas ciudades pasamos "de largo" entre la ciudad origen y la ciudad destino. Por ejemplo, si estoy en la ciudad 1 y digo de hacer dos saltos se referirá a evitar las ciudades 2 y 3, por lo tanto estaremos hablando de la ciudad 4. Otro ejemplo sería el de realizar 0 saltos, esto significaría ir a la ciudad colindante, si estamos en la ciudad 4 y hacemos 0 saltos, significa que nos referiremos a la ciudad 5.

Este enfoque de saltos en vez de numerar ciudades está siendo investigado en la actualidad, como se refleja en el estudio [11],donde no se codifica a qué ciudad se va a viajar, sino cuántos saltos desde la ciudad actual serían necesarios.

Pero la característica distintiva de esta codificación es que cuando hablamos de saltos es sobre ciudades que aún no hemos visitado. Ejemplo, estamos en la ciudad 2, pero ya hemos visitado la ciudad 3, en este caso, hablar de hacer una unidad de salto significaría hablar de la ciudad 5, porque cuando hablamos de salto, hablamos de la ciudad 4 en este caso. Como vamos eliminando ciudades que tenemos planteadas conforme se va avanzando, esto reduce significativamente los qubits necesarios para la codificación, como se puede ver en el recuadro:

Salto 1	Salto 2	Salto 3
$\|00\rangle$	$\|00\rangle$	$\|0\rangle$
$\|01\rangle$	$\|01\rangle$	$\|1\rangle$
$\|10\rangle$	$\|10\rangle$	
$\|11\rangle$		

Cuadro 3: Tabla que muestra las posibles opciones en función del salto donde estemos, hay que fijarse que los dos últimos saltos no requerirán de ningún tipo de asignación gracias a cómo funciona la codificación propuesta.

Se puede ver que la relacción entre el número de ciudades y número de qubits necesarios para su codificación crece siguiendo la expresión que expresó el anterior TFG [5]

$$\sum_{i=2}^{N-1} \lceil \log_2 i \rceil \tag{13}$$

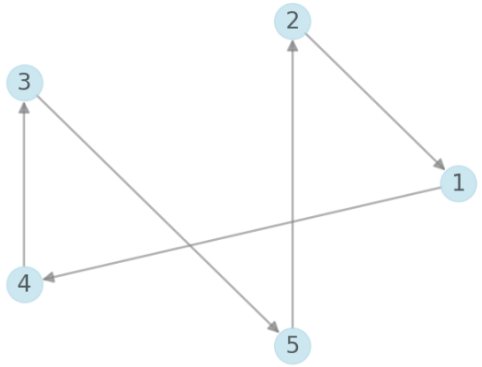

Figura 10: Ejemplo de camino de cinco ciudades.

Para afianzar esta información, paso a presentar un ejemplo completo para 5 ciudades. El objetivo es codificar un conjunto específico de "saltos" entre ciudades para una ruta de viaje específica, ver 10. Empezamos y terminamos siempre en la misma ciudad, en nuestro caso la ciudad 1. El primer destino será la ciudad 4, lo que implica que tendremos que "saltar" la ciudad 2 y 3. Este salto se codifica como $|10\rangle$.

El próximo destino en nuestra ruta es la ciudad 3. Como siempre estamos progresando en orden creciente (es decir, no damos saltos hacia atrás), tenemos que "saltar" las ciudades 5 y 2 (no tenemos en cuenta la ciudad 1). Por lo tanto, codificamos estos dos saltos de nuevo como el valor $|10\rangle$.

El último tramo de nuestro viaje nos lleva a elegir entre las ciudades 5 y 2. Si fuéramos a la ciudad 2, codificaríamos este tramo como $|1\rangle$, ya que solo "saltaríamos" la ciudad 5 pero en este caso, estamos yendo a la ciudad 5, por lo que codificamos este tramo como $|0\rangle$.

Es importante darse cuenta de que no necesitamos codificar ninguna información adicional en este punto. Dado que uno de los requisitos de los ciclos hamiltonianos es que pasemos por cada nodo exactamente una vez, y ya solo queda una ciudad sin visitar aparte de la ciudad 1, podemos asumir que el tramo restante de nuestra ruta será ir a la ciudad 2 y luego volver a la ciudad 1.

En conclusión, la codificación para el ejemplo que estamos considerando sería:

$$|T_n\rangle = |10\rangle \otimes |10\rangle \otimes |0\rangle \,. \tag{14}$$

Antes de pasar al desarrollo del oráculo hay que realizar una puntualización. Al emplear 5 qubits, se generan 32 posibles estados. Sin embargo, no todos representarán un camino válido dentro del conjunto de ciclos de Hamilton que podemos considerar. A aquellos estados que emergen en nuestra superposición, pero que no representan un ciclo, los denominaremos *estados no contemplados*. Específicamente, los estados donde los qubits 1 y 2[1] se encuentren en el estado $|11\rangle$ pertenecen a esta categoría. Esto se justifica al revisar la tabla 1.4, donde observamos que ese salto específico no está planteado. En términos sencillos, no es lógico considerar 4 saltos cuando solo existen tres opciones de salto. Por lo tanto, no existe un camino que se pueda asociar. En otras palabras, tenemos que usar 5 qubits, porque sino no podemos codificar todos los posibles caminos, pero eso no significa que los 32 estados que vamos a generar en la superposición representen información relevante.

[1]Siguiendo la convención estándar, numeramos los qubits comenzando desde el menos significativo. En términos prácticos, esto se traduce a empezar a contar desde el qubit que se encuentra más a la derecha.

2. Desarrollo del Oráculo

2.1. Desarrollo en programa clásico

Como mencioné anteriormente, he decidido mantener el amplificador de Grover en su forma original. Por lo tanto, el enfoque inicial fue crear un oráculo que asignara fases a los distintos estados en función del coste del viaje entre cada ciudad. Con la finalidad de poder desarrollar adecuadamente el oráculo, implementé un programa clásico que cumpliese dicha función. Una vez finalizado, se procedería a adaptar ese modelo para trabajar en Qiskit.

Al abordar el diseño del oráculo, el desafío principal residía en que nuestra implementación fuese capaz de descifrar la codificación que se le pasaba, esperando que pudiese identificar correctamente las ciudades y su respectivo orden, permitiendo así asignar el coste correspondiente de una matriz de costes, la cuál es de tamaño n×n, donde n es el número de ciudades en nuestro problema.

La idea inicial consistió en reconfigurar la codificación existente, lo que llevó a la creación de un nuevo vector que reflejaba la secuencia adecuada de ciudades a visitar. Para ello, se partió de un vector con un orden preestablecido (desde la ciudad 2 hasta la ciudad 5, ya que siempre comenzamos en la misma ciudad) y, mediante una serie de verificaciones, se reordenó dicho vector, ver 11. Posteriormente, este vector reorganizado se introdujo en una subrutina que, gracias a la matriz de costes, sumaba los valores correspondientes al estado en consideración.

La necesidad de generar dicho vector surge de la intrincada naturaleza de esta codificación, la cual es inherentemente dependiente del contexto o, dicho de otra forma, posee *memoria*. Realizar una unidad de salto tiene un significado distinto en función de lo que hayamos recorrido hasta el momento. Una unidad de salto puede referirse a dos ciudades distintas. Por ejemplo, si estamos en la ciudad 2 y aplicamos un salto, nos vamos a la ciudad 4, a no ser que hayamos pasado ya por la ciudad 3, en cuyo caso saltar una ciudad se referirá al salto de la propia ciudad 4, por lo que en este caso estaríamos hablando de visitar la ciudad 5, porque una vez más, contamos las ciudades a saltar en función de las que no hemos visitado.

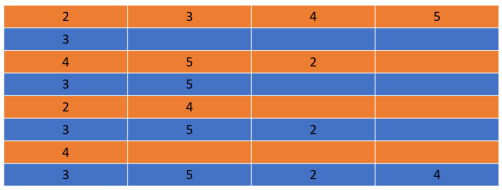

Figura 11: Ejemplo de como el vector predeterminado (color naranja) se va modificando y acaba entrando las ciudades al vector auxiliar (color azul) en el orden propuesto para la codificación concreta 01010

Aunque este código funcionaba en una computadora clásica, presenta limitaciones para una cuántica. En un entorno clásico, usar variables auxiliares como lo era el vector de las ciudades reorganizadas no era un problema; sin embargo, en un ordenador cuántico, esto se convierte en un desafío, especialmente si buscamos expandirlo a N ciudades, dada la escasa disponibilidad de qubits.

Por lo que los siguientes códigos se intentaron ir haciendo cada vez más eficientes con la intención de que sean más factibles usar dicha estructura para un ordenador cuántico.

En versiones posteriores, se buscó optimizar el código en dos puntos. Primero, se enfocó en minimizar el uso de variables auxiliares, ya que la versión inicial empleaba dos vectores de más. Por otro lado, se simplificaron las operaciones. Esto se debe a que cuanto más simple fuese el código, más facil sería la conversión del código a puertas lógicas cuánticas, el formato utilizado en Qiskit.

La versión final propuesta adoptó un enfoque similar al utilizado anteriormente para reordenar las ciudades. Sin embargo, en esta ocasión, el proceso se realizaba ciudad por ciudad. Es decir, en lugar de primero organizar y definir todas las ciudades a visitar y posteriormente aplicar los costes correspondientes, ahora descomprimíamos simultáneamente el próximo salto a realizar y su coste asociado. Esto eliminó la necesidad de utilizar variables auxiliares.

Al aplicar este método, se creaban una serie de condicionales que se activaban según el trayecto realizado hasta ese punto. Esto equivale a implementar un bucle *for* con determinadas condiciones, pero de forma manual, dado que los ordenadores cuánticos no ofrecen este tipo de operaciones de manera directa. A pesar de que esta metodología incrementó el número de líneas de código, eliminó la necesidad de recurrir a variables auxiliares, lo que representó un avance significativo. El principal inconveniente era, efectivamente, la extensión del código, pero se consideró un pequeño precio a pagar en comparación con el beneficio de evitar variables adicionales.

2.2. Desarrollo en Qiskit

Tras validar el correcto funcionamiento del código clásico, estamos listos para traducirlo al lenguaje de Qiskit. Para facilitar la comprensión, opté por presentar el código de manera progresiva en lugar de mostrarlo todo de una vez.

2.2.1. Primer salto

```
qc13 = QuantumCircuit(5)
qc13.h(4)
qc13.h(3)
qc13.h(1)
qc13.h(2)
qc13.h(0)

qc13.x(3)
qc13.x(4)
qc13.cp(m[0][1],3, 4) # Aplicar fase al qubit 00
qc13.x(4)
qc13.x(3)

qc13.p(m[0][2], 3)  # Aplicar fase al qubit 01
qc13.p(m[0][3], 4)  # Aplicar fase al qubit 10

qc13.cp(m[0][4]-(m[0][2]+m[0][3]),3, 4) # Aplicar fase al qubit 11
```

El primer paso consiste en superponer todos los qubits como se puede ver en la siguiente ecuación:

$$|\psi\rangle = \frac{1}{\sqrt{32}} \sum_{k=0}^{31} |k\rangle, \tag{15}$$

donde $|k\rangle$ son los estados desde $|00000\rangle$ hasta $|11111\rangle$.

Al comenzar, en vez de abordar cada salto de forma individual, tenemos la capacidad de asignar varios al mismo tiempo gracias a los estados superpuestos generados con las puertas Hadamard. Recalcar que los costes de los distintos caminos van a estar relacionados con las fases.

Iniciamos asignando las fases correspondientes según el estado de los qubits. El programa clásico avisaba de la sencillez que supone establecer la primera ciudad. Solo necesitamos codificar las cuatro

opciones iniciales, que se corresponden con los cuatro saltos iniciales posibles. Estas opciones se vinculan con un estado de los dos qubits, ya que debemos representar 2^2 rutas.

A la hora de explicar como se van a signando las fases a los distintos estados, es preferible comenzar explicando el proceso de asignación de fases a los estados $|01\rangle$ y $|10\rangle$, ya que son los mas sencillos de realizar, para ello solo va a ser necesario una puerta *phase* de un solo qubit.

Si ponemos la puerta *phase* en el qubit menos significativo de los dos que hemos escogido, estamos hablando del estado $|01\rangle$ el cual está relacionado con la ciudad 3, una vez hecho esto, se puede indicar la posición de la matriz a la que corresponde, donde está el coste asociado del camino hacia la siguiente ciudad . Misma ejecución para el estado $|10\rangle$ solamente que indicamos el otro qubit.

Ahora aplicamos una puerta de fase controlada de dos qubits, esta operación nos indica que agregará una fase si, y solo si, ambos qubits están en el estado $|1\rangle$, se representa de la siguiente manera:

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & e^{i\theta_4} \end{bmatrix} \tag{16}$$

Sin embargo, al revisar las puertas de un solo qubit, observamos que no solo han asignado una fase a los estados $|10\rangle$ y $|01\rangle$, sino que también han sumado la fase que tenían asignado de la matriz de costes al estado $|11\rangle$. Sabemos que asigna la fase cuando está en el estado $|1\rangle$ en qubit donde ponemos la puerta, pero su acción no está condicionada a otro qubit. Para compensar este efecto, debemos añadir una fase opuesta a la asignada en los dos estados anteriores cuando operamos con el estado $|11\rangle$. Visualmente, esto se traduce en la siguiente operación para la puerta de fase controlada con dos qubits:

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & e^{i\theta_4} \times e^{-i\theta_4} \times e^{-i\theta_3} \end{bmatrix} \tag{17}$$

Por lo tanto llegados a éste punto, el conjunto de puertas lógicas sería la siguiente matriz

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & e^{i\theta_2} & 0 & 0 \\ 0 & 0 & e^{i\theta_3} & 0 \\ 0 & 0 & 0 & e^{i\theta_4} \end{bmatrix} \tag{18}$$

Pretendemos añadir una fase al estado $|00\rangle$ que representaría en este momento ir a la ciudad 2. Directamente no hay una puerta que lo haga, porque el primer estado se suele usar como referencia (al fin y al cabo las fases en un ordenador cuántico no son absolutas, sino relativas entre los distintos estados). Pero como en nuestro programa será un tema a debatir más adelante, he propuesto hacer uso otra vez de una puerta controlada de dos qubits, solo que ahora se añaden una puerta *Not* a cada uno de los dos qubits de manera previa. El razonamiento es el siguiente: al aplicar dos puertas Not a los dos qubits pasamos de estar en el estado $|00\rangle$ al estado $|11\rangle$, es aquí cuando aplicamos la puerta de fase, una vez hecho esto deshacemos las puertas *Not*, por consiguiente habremos podido aplicar una puerta de fase al estado $|00\rangle$. Recalcar que cada vez que se haga uso de una puerta *Not*, debemos de aplicarla de nuevo para volver al estado anterior, porque sino el programa no saldrá como se pretende. Al final hemos conseguido el objetivo de este paso, que todos nuestros estados estén con la fase correspondiente al coste de este primer salto:

$$\mathbf{C_1} = \begin{bmatrix} e^{i\theta_1} & 0 & 0 & 0 \\ 0 & e^{i\theta_2} & 0 & 0 \\ 0 & 0 & e^{i\theta_3} & 0 \\ 0 & 0 & 0 & e^{i\theta_4} \end{bmatrix} \tag{19}$$

Es crucial recalcar que, al añadir las fases a los dos qubits, todos los estados resultantes de la superposición que tengan cualquiera de los cuatro estados heredarán dicha fase. Esta propiedad emerge de la naturaleza del producto tensorial. Por ejemplo, al aplicar una puerta a los dos primeros qubits, observamos el siguiente comportamiento:

$$U \otimes I \otimes I \otimes I \otimes I \left|\psi\right\rangle, \tag{20}$$

donde U es la matriz que representa las puertas lógicas que acabamos de aplicar e I matrices unitarias.

Imaginemos que estamos en el caso del estado $|11\rangle$, tendríamos lo siguiente:

$$U \otimes I \otimes I \left|11000\right\rangle = U|11\rangle \otimes I|00\rangle \otimes I|0\rangle \otimes I = e^{i\theta_4}|11\rangle \otimes |00\rangle \otimes |0\rangle = e^{i\theta_4} \left|11000\right\rangle. \tag{21}$$

Cualquiera de los caminos resultantes que contenga ese primer estado formado por los dos qubits tendrán la misma fase relaccionada con la primera ciudad visitada:

$$U \otimes I \otimes I \left|11011\right\rangle = U|11\rangle \otimes I|01\rangle \otimes I|1\rangle \otimes I = e^{i\theta_4}|11\rangle \otimes |01\rangle \otimes |1\rangle = e^{i\theta_4} \left|11011\right\rangle. \tag{22}$$

2.2.2. Segundo salto, introducción de la puerta auxiliar de asignación de fase

```
def coste2q(a,b, c) :
    qc = QuantumCircuit(2)

    qc.p(b, 0)  # Aplicar fase al qubit 01
    qc.p(c, 1)  # Aplicar fase al qubit 10
    qc.x(0)
    qc.x(1)
    qc.cp(a,0, 1) # Aplicar fase al qubit 00
    qc.x(0)
    qc.x(1)

    qc.cp(-(b+c),0, 1) # Eliminar fase al qubit 11

    return qc

def coste2_q(a,b, c, d) :
    qc = QuantumCircuit(2)

    qc.p(b, 0)  # Aplicar fase al qubit 01
    qc.p(c, 1)  # Aplicar fase al qubit 10
    qc.x(0)
    qc.x(1)
    qc.cp(a,0, 1) # Aplicar fase al qubit 00
    qc.x(0)
    qc.x(1)

    qc.cp(-(b+c+d),0, 1) # Eliminar fase al qubit 11

    return qc
```

```
cPhase00=coste2_q(m[1][2], m[1][3], m[1][4], m[0][1]+np.pi/2).to_gate
    ().control(2)
qc13.x([3, 4])
qc13.append(cPhase00, [4, 3, 1, 2])
qc13.x([ 3, 4])


cPhase01=coste2_q(m[2][3], m[2][4], m[2][1], m[0][2]).to_gate().
    control(2)
qc13.cx(3, 4)
qc13.append(cPhase01, [4, 3, 1, 2])
qc13.cx(3, 4)


cPhase10=coste2_q(m[3][4], m[3][1], m[3][2], m[0][3]).to_gate().
    control(2)
qc13.cx(4,  3)
qc13.append(cPhase10, [4, 3, 1 ,2])
qc13.cx(4, 3)


cPhase11=coste2_q(m[4][1], m[4][2], m[4][3], m[0][4]).to_gate().
    control(2)
qc13.append(cPhase11, [4, 3, 1, 2])
```

Para el primer salto, contamos con cuatro rutas posibles. Ahora, para cada una de esas rutas, hay tres opciones disponibles para el segundo salto, resultando en un total de doce rutas distintas. Para representar estas rutas en un sistema cuántico, necesitaremos trabajar con 4 qubits.

El diseño del segundo código toma inspiración directa del programa clásico. Iniciamos definiendo de qué ciudad venimos. Posteriormente, empleamos una puerta adicional que asigna la fase adecuada a las rutas elegidas para el segundo salto.

Centrándonos en el primer bloque del código que se ha expuesto, observamos una puerta personalizada, un tipo de código frecuentemente empleado en subrutinas. En nuestro contexto, esta estructura se repetirá a lo largo del programa. Disponemos de dos funciones las cuales cumplen objetivos muy similares. Examinemos específicamente la función coste2_q, destinada exclusivamente al segundo salto. Esta acepta cuatro entradas: las tres primeras corresponden a los tres caminos posibles en este punto, es decir, $|00\rangle$, $|01\rangle$ y $|10\rangle$. La asignación de fases para cada uno de los estados se logran mediante un conjunto de puertas lógicas en una subrutina, la cuál está inspirada en la estructura del primer salto.

Sin embargo, para este paso en concreto, tuve que introducir una cuarta fase para el estado $|11\rangle$. Esto responde a la necesidad de rectificar fases previamente añadidas debido a que el producto tensorial antes mencionado también afecta a los estados no contemplados. Para aquellos estados que no se consideran ya que no contienen información de algún ciclo de Hamilton y no tienen ninguna relevancia en el algoritmo, se lleva a cabo un esfuerzo adicional para asegurar que no posean ningún valor de fase, para que tengan valor de 0. Por lo tanto, para contrarrestar la fase indeseada que estos estados obtenienen durante la asignación del primer salto, aplico una fase negativa en este punto. Es más conveniente realizar este ajuste ahora, dado que estamos manejando menos qubits.

2.2.3. Tercer salto

```
    cPhase00=coste2q(m[2][4], m[3][2], m[4][3]).to_gate().control(3)

    qc13.x([3, 4 ])

    qc13.append(XGate().control(4), [4,3,2, 1,0])
```

```
qc13.append(cPhase00, [4,3,0,1,2])

qc13.append(XGate().control(4), [4,3,2, 1,0])

qc13.x([3, 4])




cPhase01=coste2q(m[3][1], m[4][3], m[1][4]).to_gate().control(3)

qc13.cx( 3, 4)


qc13.append(XGate().control(4), [ 4,3,2, 1,0])
qc13.append(cPhase01, [4,3,0,1,2])
qc13.append(XGate().control(4), [ 4,3,2, 1, 0])


qc13.cx( 3, 4)







cPhase10=coste2q(m[4][2], m[1][4], m[2][1]).to_gate().control(3)
qc13.cx(4, 3)

qc13.append(XGate().control(4), [4,3,2, 1,0])

qc13.append(cPhase10, [4,3,0,1,2])

qc13.append(XGate().control(4), [4,3,2, 1,0])

qc13.cx(4, 3)




cPhase11=coste2q(m[1][3], m[2][1], m[3][2] ).to_gate().control(3)
qc13.append(XGate().control(4), [4,3,2, 1,0])

qc13.append(cPhase11, [4,3,0,1,2])

qc13.append(XGate().control(4), [4,3,2, 1,0])




qc13.x(0)   #qubit 0 pasa a asignar cuando est  a 0

cPhase00=coste2q(m[2][3], m[3][4], m[4][2]).to_gate().control(3)

qc13.x([3, 4 ])
```

```
qc13.append(XGate().control(4), [4,3,2, 1,0])

qc13.append(cPhase00, [4,3,0,1,2])

qc13.append(XGate().control(4), [4,3,2, 1,0])

qc13.x([3, 4])




cPhase01=coste2q(m[3][4], m[4][1], m[1][3]).to_gate().control(3)

qc13.cx( 3, 4)


qc13.append(XGate().control(4), [ 4,3,2, 1,0])
qc13.append(cPhase01, [4,3,0,1,2])
qc13.append(XGate().control(4), [ 4,3,2, 1, 0])


qc13.cx( 3, 4)




cPhase10=coste2q(m[4][1], m[1][2], m[2][4]).to_gate().control(3)
qc13.cx(4, 3)

qc13.append(XGate().control(4), [4,3,2, 1,0])

qc13.append(cPhase10, [4,3,0,1,2])

qc13.append(XGate().control(4), [4,3,2, 1,0])

qc13.cx(4, 3)




cPhase11=coste2q(m[1][2], m[2][3], m[3][1] ).to_gate().control(3)
qc13.append(XGate().control(4), [4,3,2, 1,0])

qc13.append(cPhase11, [4,3,0,1,2])

qc13.append(XGate().control(4), [4,3,2, 1,0])


qc13.x(0)
```

Para la siguiente ciudad, adoptamos una estructura similar al salto de la ciudad previa, aunque debemos considerar ahora también el qubit menos significativo. Dependiendo de si este qubit está en el estado 1 o 0, seleccionaremos una ruta u otra entre las dos que quedan por elegir. Esto es equivalente en el código clásico a un condicional *if*. He traducido esta condición al lenguaje cuántico mediante una puerta *CNot* que actúa condicionada por los demás qubits. Así, somos capaces de intervenir en el estado deseado. Consideremos, por ejemplo, el primer bloque "cPhase01". Primero, establecemos una

condición en los dos qubits iniciales para que adopten el estado 01; para ello, simplemente requerimos un *CNot* con el tercer qubit como control y el cuarto como objetivo. Posteriormente, utilizamos la puerta condicionada para operar en el qubit 0 cuando está en el estado 1. Con tres qubits ya condicionados, implementamos la puerta personalizada, permitiendo asignar la fase adecuada al estado deseado. Después de esta operación, si invertimos el estado del qubit 0 utilizando una puerta *Not*, alteramos la condición: ahora operamos sobre el qubit cuando está en el estado 0. Este enfoque no solo nos permite condicionar las rutas en función del último qubit, sino que también optimiza el uso de las puertas lógicas, resultando en un programa más eficiente.

2.2.4. Cuarta y quinta ciudad

En la cuarta ciudad se hizo un programa muy similar al usado para la tercera ciudad, ya que el número de fases a asignar era la misma y tenemos que considerar la misma matriz 4x4, únicamente se debe de especificar la nueva fase correspondiente.

En el último escenario, nos enfrentamos a un desafío previamente mencionado, ligado a la memoria necesaria para esta codificación. La interpretación del último qubit, ya sea en el estado $|0\rangle$ o $|1\rangle$, cambia drásticamente dependiendo del resto del string. Por lo tanto, el código desarrollado es bastante similar al anterior, a pesar de que únicamente requerimos determinar un solo valor del vector 1x4 de la primera columna cuyo coste representa el volver de las otras ciudades a la ciudad inicial/final. Es decir, a pesar de que el conjunto de datos de la matriz es bastante más reducido, el código no se simplificó proporcionalmente, sino que se asemeja al usado para los dos saltos anteriores.

Con esto ya tendríamos el código del oráculo creado.

3. Implementación del oráculo

Tras la construcción del oráculo, lo siguiente es relativamente más directo. Es esencial mencionar que el difusor, el cual es del original de Grover, puede adquirirse directamente desde `quiskit.com`. Ahora mismo, el aspecto destacable a considerar es el proceso que escala las distancias y las convierte en fases.

Antes de adentrarnos en el código, es vital recordar que nuestra meta es que los caminos más eficientes alcancen una fase cercana a π. Es importante considerar cómo se manejan las fases de los caminos según su coste. Los caminos con un coste más elevado deberían tener una fase cercana a 0. Si partimos inicializando todos los estados con una fase de 0, nos encontramos con dos posibles escenarios:

1. Los caminos más eficientes podrían no llegar a una fase cercana a π a menos que asignemos una fase sustancial a cada salto. Sin embargo, esto conlleva el riesgo de exceder dicha fase.

2. Si optamos por fases más bajas, nos alejaríamos demasiado de nuestro objetivo.

Contrariamente, si comenzamos con una fase de π y reducimos las fases de los caminos menos óptimos, enfrentaríamos problemas similares pero de manera inversa.

Por estas razones, se decide inicializar todos los estados con una fase de $\pi/2$. Así, para los caminos más favorables, añadiremos pequeñas fases que acercarán estos estados a una fase de π, mientras que para los demás, las fases se reducirán, aproximándose a 0. A continuación se presenta el código correspondiente:

```
import numpy as np
from fractions import Fraction

def assign_intervals(matrix):
    min_val = np.min(matrix)
```

```
    max_val = np.max(matrix)
    interval = (max_val - min_val) / 5
    intervals = [min_val + i*interval for i in range(5)]  # Creamos 5
        puntos de intervalo
    intervals.append(max_val + 1)  # A adimos un valor extra a los
       intervalos para asegurar que el max_val sea incluido
    pi_values = [-np.pi/10, -np.pi/15, 0, +np.pi/15, np.pi/10]
    pi_values.reverse()
    new_matrix = np.zeros((5,5))

    for i in range(5):
        for j in range(5):
            for k in range(5):
                if intervals[k] <= matrix[i][j] < intervals[k+1]:  #
                    Ajusta la condici n para incluir el l mite
                    superior del  ltimo  intervalo
                    new_matrix[i][j] = pi_values[k]
                    break

    return new_matrix

# Creaci n de la matriz aleatoria con valores entre 0 y 10
matrix = np.random.randint(0, 11, size=(5, 5))

# Asignar los intervalos
m = assign_intervals(matrix)
```

Inicialmente, generamos una matriz aleatoria con valores entre 0 y 10, que simulan las distancias entre diferentes ciudades. En un contexto real, estos datos serían proporcionados como parte de la definición del problema. A continuación, iniciamos la función \textitassign_intervals. Esta función comienza identificando los valores máximos y mínimos dentro de la matriz. Con estos valores extremos, se establecen cinco intervalos que se correlacionarán con distintas fases. Cada valor dentro de la matriz es entonces mapeado a una fase específica basándose en el intervalo en el que se encuentra. Los intervalos definidos son los siguientes:

$$\left[\frac{\pi}{10} \quad \frac{\pi}{15} \quad 0 \quad -\frac{\pi}{15} \quad -\frac{\pi}{10}\right] \tag{23}$$

Si un valor de coste se encuentra dentro del intervalo más favorable, este se asociará con un incremento de fase más significativo. En contraposición, los costes elevados serán penalizados con fases que restar para tender hacia 0. La elección de valores de fase como $\pi/10$ es deliberada. Si consideramos el mejor escenario, en el que todos los saltos tienen un coste extremadamente bajo, la suma total de las fases sería de $\pi/2$, esta fase se suma a la fase inicializada a cada uno de los estados que es de $\pi/2$ y el resultado sería una fase total de π. Si se hubieran asignado fases mayores, podríamos enfrentar el riesgo de superar π, lo que podría comprometer que ese estado tenga la probabilidad más alta de ser observado.

Por último solo quedaría configurar cuantas veces hay que aplicar el oráculo y el difusor por el motivo explicado en la introducción. Aplicando la fórmula correspondiente, ver de nuevo 10, hay que iterarlo 4 veces. Cosa que se puede comprobar en el anexo, en el código comentado como *asignación de iteraciones*.

4. Resultados

Para testear el funcionamiento del algoritmo se optó por realizar cierto número de implementaciones para ver la capacidad que tiene de resolver un problema real. Para ello se optó por probarlo en un ordenador cuántico simulado y luego contrastar los resultados medidos en un ordenador real.

4.1. Implementación en un ordenador cuántico simulado

4.1.1. Simulación 1

Observamos que el estado 10101, está asociado con el coste más bajo de valor 17 y relacionado a un valor asignado de dos radianes[2]. Vamos a comprobar primero con cuantas repeteciciones debemos trabajar. La matriz de coste asignada al siguiente ejemplo:

$$\begin{bmatrix} 7 & 9 & 1 & 5 & 7 \\ 9 & 8 & 4 & 9 & 2 \\ 5 & 10 & 5 & 8 & 4 \\ 5 & 9 & 0 & 0 & 4 \\ 0 & 6 & 4 & 5 & 3 \end{bmatrix} \tag{24}$$

Con $N = 2$

Se observa cierta equiprobabilidad, no tenemos un resultado claro.

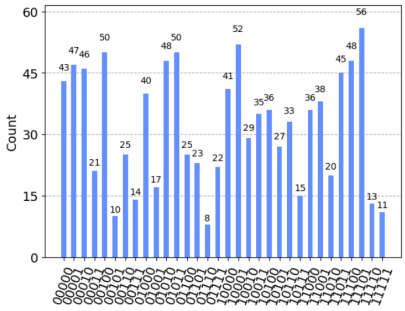

Figura 12: Simulación con solo dos iteraciones con el algoritmo de Grover.

Con $N = 3$

Seguimos sin tener un buen resultado, incluso dependiendo de la simulación el estado más medido va variando.

[2]Se han medido directamente los valores de fase asignados por el oráculo en el simulador, ya que con las líneas de código adecuadas, se puede medir las fases de cada uno de los estados. Sin embargo, en un ordenador cuántico real, estas fases no se pueden medir directamente. El código está al final del anexo

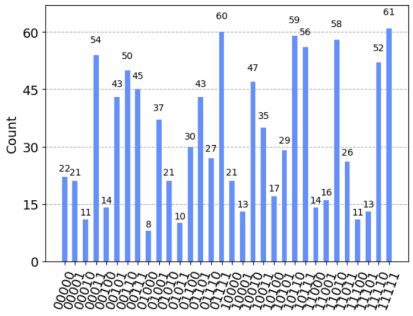

Figura 13: Simulación con tres iteraciones del algoritmo de Grover.

Con $N = 4$

Se ve claramente que mide correctamente el estado que estamos buscando, el que refleja menor coste en este caso.

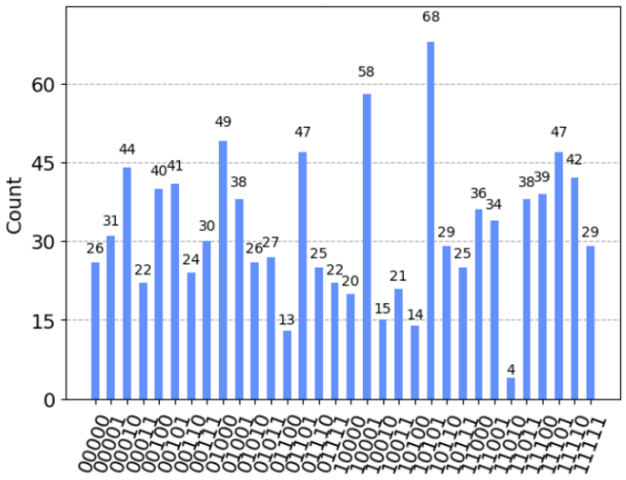

Figura 14: Resultado de la simulación con cuatro simulaciones.

Con $N = 5$

Vemos que hemos dejado de medir el estado correcto.

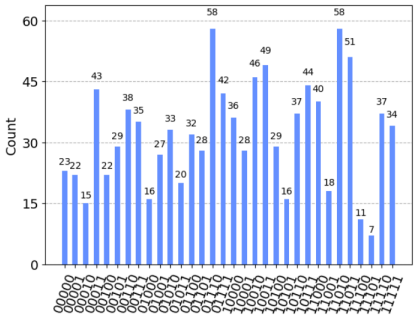

Figura 15: Resultado de la simulación del ejemplo 1 con cinco iteraciones del algoritmo.

4.1.2. Simulación 2

Es evidente que el valor 01000 es el más sobresaliente, correspondiendo a 1.88 radianes con un coste de 20. Este siempre se observa como el más medido. En este ejemplo tenemos otros estados con valores de fase los cuales tienen valores próximos a la fase mayor, por lo tanto, a la hora de medir la diferencia entre las opciones no es tan marcada como en el ejemplo anterior. Aunque el camino con la mayor probabilidad sigue siendo el más medido, los caminos que le siguen no necesariamente reflejan el orden de rutas más eficiente, apartando el caso más óptimo. Concluyendo, el segundo estado medido no es el segundo estado más eficiente.

$$\begin{bmatrix} 3 & 6 & 5 & 6 & 10 \\ 0 & 8 & 7 & 6 & 9 \\ 1 & 10 & 10 & 0 & 4 \\ 4 & 10 & 5 & 1 & 9 \\ 0 & 5 & 5 & 9 & 7 \end{bmatrix} \tag{25}$$

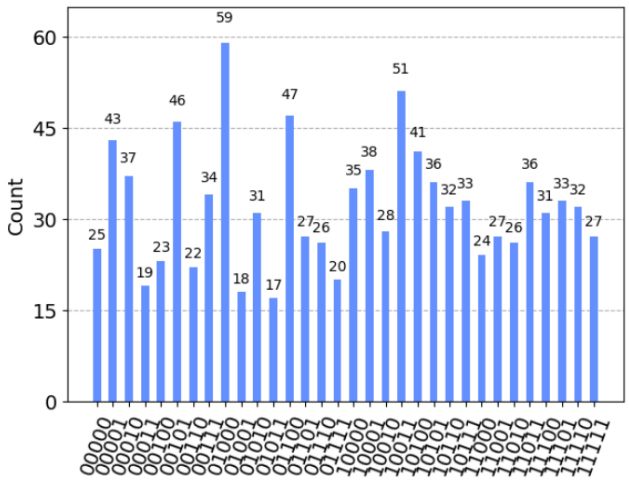

Figura 16: Resultado de la simulación del ejemplo 2.

4.1.3. Simulación 3

He realizado una pequeña adaptación a los estados que no estaban previamente contemplados. Si varios de estos tenían valores cercanos a 2π, el sistema podía presentar inconsistencias. Para resolver esto, decidí añadirles 0,3 radianes. Con este ajuste, el sistema funcionó adecuadamente. Aún queda determinar si este comportamiento es una limitación intrínseca de la simulación.

Nuevamente, los valores con mayor fase (y por lo tanto, con menor coste) son los más prominentes. Sin embargo, según la simulación, el resultado puede variar. Aunque teóricamente, el valor 11010 debería tener una probabilidad (tiene un coste de 11) ligeramente mayor que el estado 11101(coste 14), en función de la simulación destaca más uno u otro.

```
[[ 3  3  5  3  0]
 [ 2 10 10  6 10]
 [ 2  5  7  2 10]
 [ 0  4  0  1  5]
 [ 8 10  3  4  3]]
```

Figura 17: Matriz de coste del ejemplo 3.

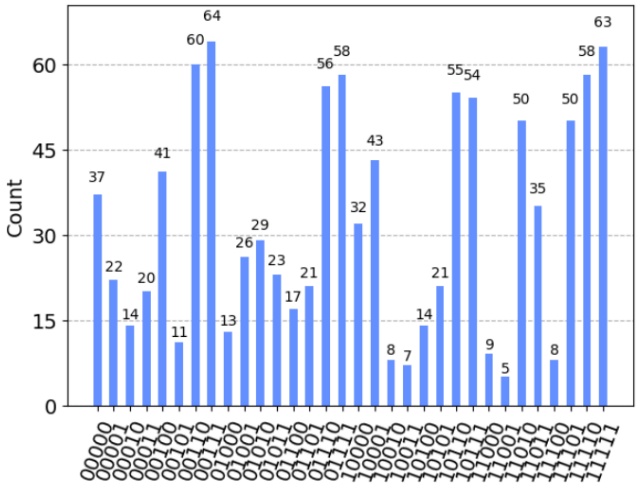

Figura 18: Resultado de la simulación del ejemplo 3.

4.1.4. Simulación 4

Vuelve a salir con mayor probabilidad el valor con más fase (menor coste). Vuelve a darse el caso de que la segunda opción no es el segundo caso con menor coste.

$$\begin{bmatrix} 1 & 3 & 1 & 3 & 7 \\ 8 & 9 & 5 & 8 & 7 \\ 3 & 6 & 8 & 6 & 0 \\ 10 & 2 & 6 & 7 & 3 \\ 0 & 3 & 4 & 2 & 0 \end{bmatrix} \tag{26}$$

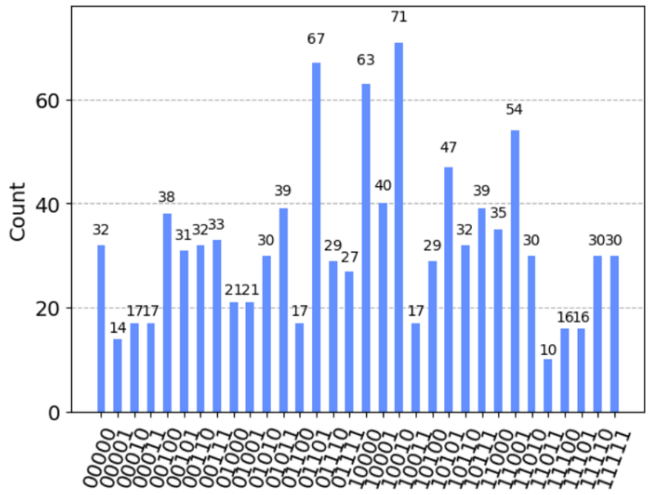

Figura 19: Resultado de la simulación del ejemplo 4.

4.1.5. Simulación 5

Una vez más los estados con más fase vuelven a estar entre los que más probabilidades tiene.

$$\begin{bmatrix} 7 & 0 & 4 & 10 & 3 \\ 9 & 3 & 7 & 1 & 2 \\ 3 & 1 & 2 & 10 & 7 \\ 3 & 8 & 8 & 5 & 10 \\ 3 & 0 & 9 & 10 & 4 \end{bmatrix} \tag{27}$$

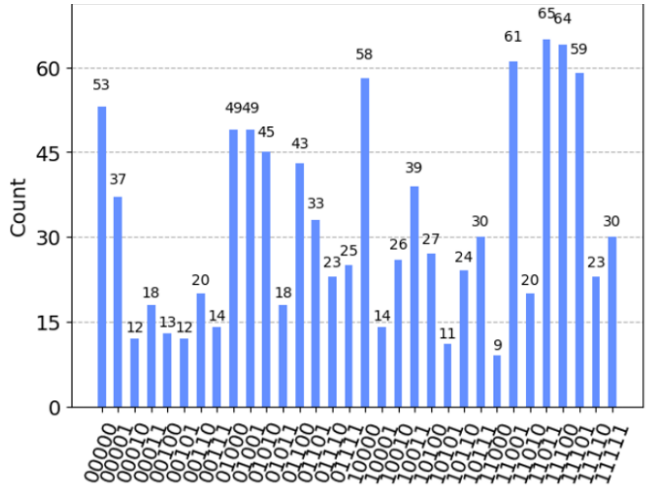

Figura 20: Resultado de la simulación del ejemplo 5.

4.1.6. Simulación 6

Caso perfecto. Señala claramente la mejor opción, por otro lado el segundo estado también sería a tener en cuenta.

$$\begin{bmatrix} 10 & 5 & 2 & 9 & 9 \\ 4 & 0 & 8 & 3 & 6 \\ 0 & 2 & 4 & 6 & 4 \\ 0 & 6 & 7 & 1 & 8 \\ 0 & 9 & 5 & 6 & 2 \end{bmatrix} \tag{28}$$

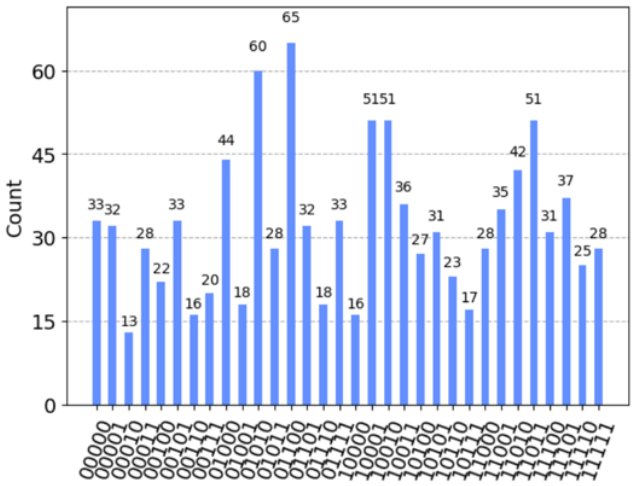

Figura 21: Resultado de la simulación del ejemplo 6.

4.1.7. Simulación 7

Estamos ante un ejemplo óptimo, existe un camino que supera significativamente a los demás, con un costo de solo 8, equivalente a 2.42 radianes. El algoritmo identifica este caso de manera precisa.

$$\begin{bmatrix} 4 & 4 & 10 & 1 & 10 \\ 0 & 4 & 1 & 9 & 4 \\ 10 & 1 & 10 & 5 & 2 \\ 7 & 4 & 8 & 6 & 3 \\ 0 & 0 & 6 & 1 & 7 \end{bmatrix} \tag{29}$$

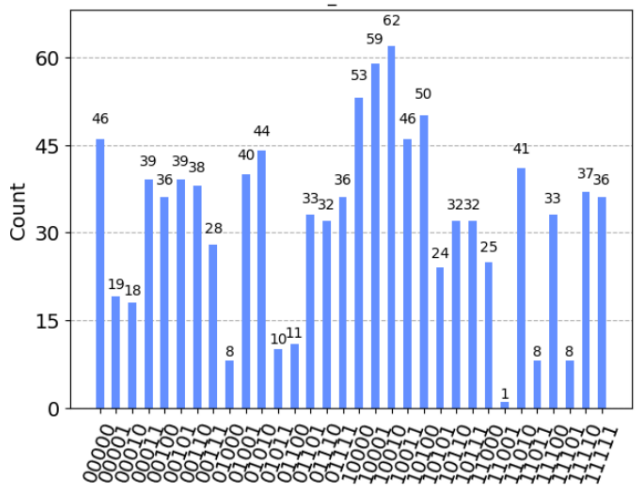

Figura 22: Resultado de la simulación del ejemplo 7.

4.1.8. Simulación 8

Aunque generalmente identifica correctamente los estados con mayor fase (y por lo tanto, los mejores casos), no lo hace con la precisión ni eficacia observadas en situaciones anteriores. A menudo destaca los casos más favorables; sin embargo, por razones que aún desconozco, tiende a asignar mayor probabilidad a un estado con una fase ligeramente menor que el caso óptimo.

$$\begin{bmatrix} 1 & 5 & 9 & 4 & 2 \\ 1 & 6 & 10 & 7 & 3 \\ 2 & 4 & 4 & 8 & 4 \\ 4 & 2 & 9 & 9 & 3 \\ 9 & 7 & 1 & 0 & 10 \end{bmatrix} \tag{30}$$

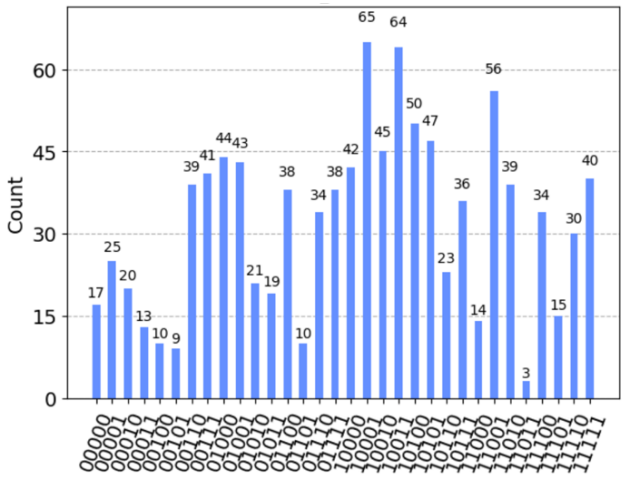

Figura 23: Resultado de la simulación del ejemplo 8.

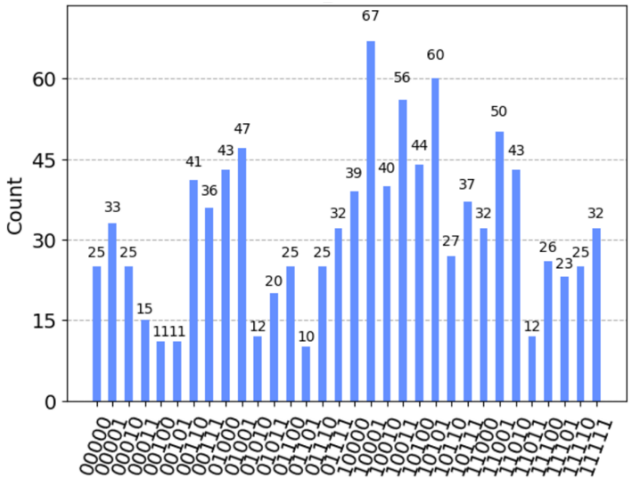

Figura 24: Resultado de la simulación del ejemplo 8, segundo ejemplo.

4.1.9. Simulación 9

Se presenta nuevamente el problema anterior, pero esta vez de manera más pronunciada. Aunque el estado más eficiente es el 00101, con un coste de 11 y equivalente a 2.42 radianes (lo que debería hacerlo el más probable con diferencia), el algoritmo destaca en cambio el estado 00001. Este último tiene un coste de 15 y un valor en radianes de 2. Aunque teóricamente, a medida que nos acercamos a π, debería incrementarse la probabilidad, el algoritmo no toma en cuenta el estado óptimo en esta ocasión.

$$\begin{bmatrix} 8 & 3 & 4 & 8 & 6 \\ 0 & 0 & 8 & 8 & 3 \\ 2 & 9 & 5 & 7 & 2 \\ 1 & 8 & 2 & 8 & 9 \\ 8 & 8 & 4 & 1 & 9 \end{bmatrix} \tag{31}$$

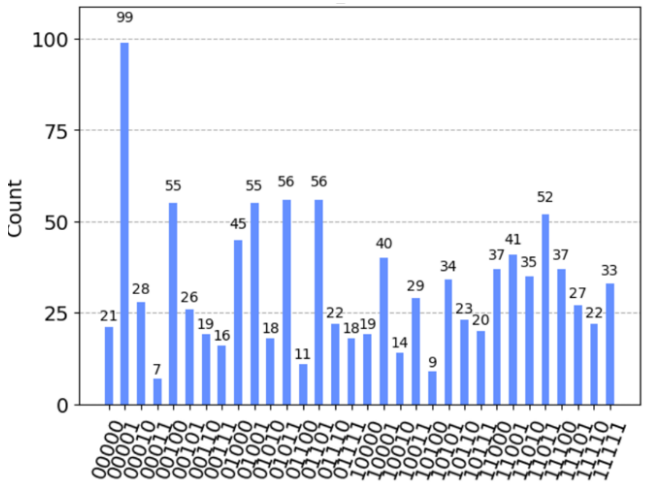

Figura 25: Resultado de la simulación del ejemplo 9.

4.1.10. Simulación 10

Nuevamente enfrentamos el mismo inconveniente: el algoritmo parece favorecer a los estados con valores alrededor de 2 radianes en lugar de aquellos más cercanos a π.

$$\begin{bmatrix} 2 & 8 & 0 & 7 & 10 \\ 1 & 7 & 7 & 9 & 0 \\ 6 & 1 & 4 & 0 & 6 \\ 5 & 2 & 2 & 8 & 7 \\ 4 & 6 & 7 & 2 & 10 \end{bmatrix} \tag{32}$$

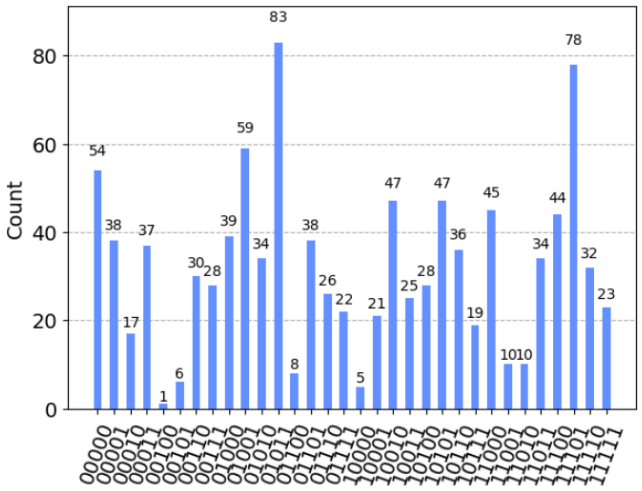

Figura 26: Resultado de la simulación del ejemplo 10.

Al ajustar a 5 iteraciones, ahora el algoritmo efectivamente favorece a los estados más cercanos a π, siendo 2.42 radianes.

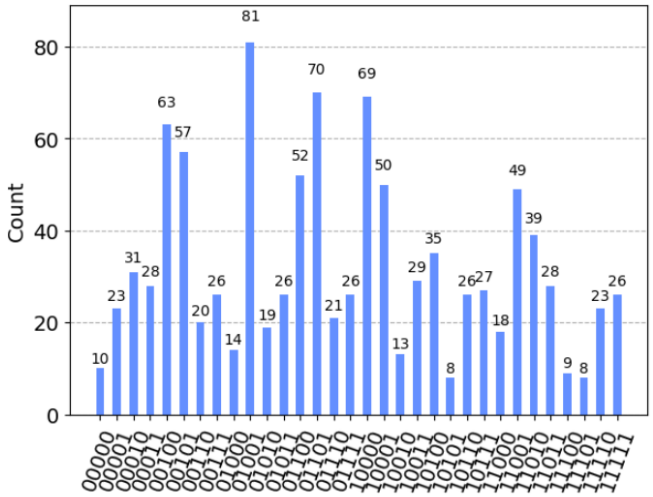

Figura 27: Resultado de la simulación del ejemplo 10, segundo ejemplo modificando el número de iteraciones.

4.1.11. Simulación 11

Esta era una simulación que al igual que los dos casos anteriores no favorecía el caso mejor, entonces decidí probar con cinco iteraciones en vez de cuatro, en ésta ocasión el sistema respondió de la mejor manera posible.

$$\begin{bmatrix} 6 & 9 & 8 & 9 & 0 \\ 7 & 7 & 6 & 0 & 9 \\ 9 & 1 & 4 & 9 & 5 \\ 5 & 10 & 0 & 8 & 5 \\ 6 & 4 & 5 & 7 & 3 \end{bmatrix} \tag{33}$$

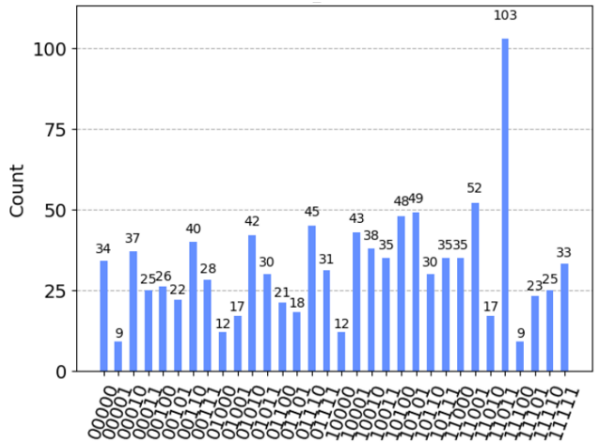

Figura 28: Resultado de la simulación del ejemplo 11 con $N = 5$.

4.1.12. Simulación 12

Me fijé que el sistema volvía a fallar con cinco iteraciones, si volvía a poner solamente 4 sí que funcionaba correctamente, luego me fijé que podría darse debido a que hay varias soluciones válidas, esto supondría que la fórmula 10 acertaría al ser $t \neq 1$.

$$\begin{bmatrix} 4 & 3 & 9 & 4 & 8 \\ 9 & 9 & 5 & 6 & 2 \\ 2 & 3 & 7 & 2 & 10 \\ 10 & 8 & 7 & 3 & 9 \\ 10 & 8 & 4 & 4 & 8 \end{bmatrix} \tag{34}$$

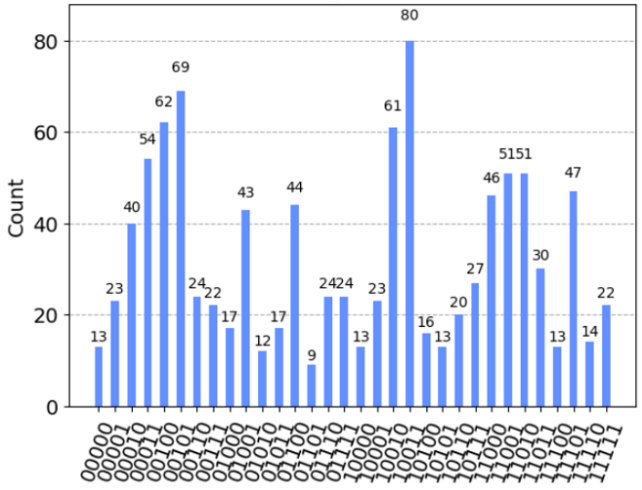

Figura 29: Resultado de la simulación del ejemplo 12 con $N = 4$.

4.2. Implementación en ordenador cuántico real

Al probar el algoritmo en un ordenador simulado, el siguiente paso lógico fue implementarlo en un ordenador cuántico real. Elegí el ordenador IBM_Perth para realizar estas pruebas, ya que tenía el valor

más alto de QV[3]. Se introdujo el código junto con la matriz de coste para la ejecución, posteriormente, se evaluaron los resultados obtenidos. A continuación se presentan dichos resultados:

4.2.1. Aplicación de la simulación 1

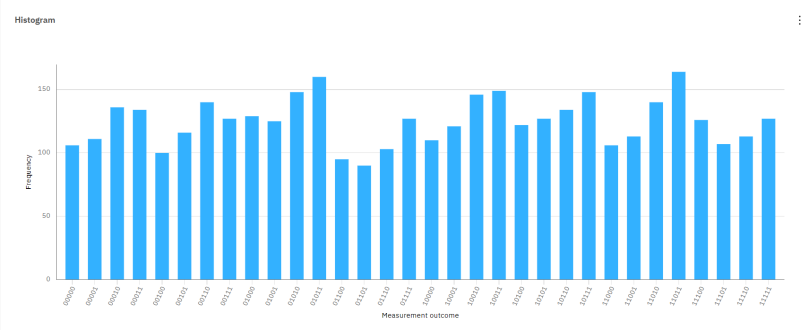

Figura 30: Resultado de la aplicación del algoritmo en el IBM Perth

Los resultados no presentan un valor definido; en su mayoría, las mediciones arrojan cifras muy similares, lo que indica una fallo del algoritmo. Este comportamiento ha sido una constante en casi todas las simulaciones realizadas.

4.2.2. Aplicación de la simulación 12

Se decidió probar un caso que debiese de finalizar con una probabilidad muy alta, a la vez un caso en el cual hubiese una notable diferencia entre usar 4 o 5 iteraciones, esto lo utilicé para comprobar si era más efectivo usar un número u otro. Se optó por la simulación 12. Con cuatro iteraciones tenemos lo siguiente:

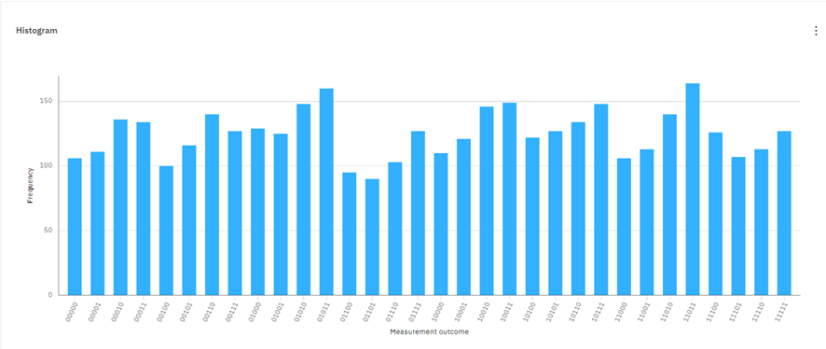

Figura 31: Resultado de la aplicación del algoritmo en el IBM Perth

Se puede ver que el estado que debía de salir sobresale un poco por encima del resto de estados, parece que el sistema ha funcionado. Lo inesperado es que ha ocurrido con cuatro iteraciones cuando esto en un ordenador simulado ocurría con 5. Por otro lado, con cinco repeticiones del oráculo y el amplificador en un ordenador real no funciona el sistema correctamente.

[3]QV, que proviene de "Quantum Volume", mide el tamaño del circuito aleatorio más grande que, manteniendo un ancho y una profundidad uniformes, la computadora puede ejecutar exitosamente

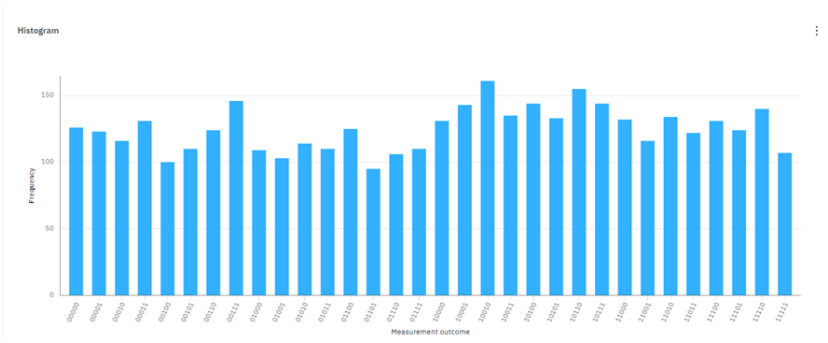

Figura 32: Resultado de la aplicación del algoritmo en el IBM Perth con cinco iteraciones.

5. Discusión de resultados

La primera observación derivada de nuestros experimentos es que, durante la simulación, el algoritmo muestra una preferencia por ciertos estados en comparación con otros. Para ilustrar esto, consideremos el siguiente escenario simplificado a modo de ejemplo como si el agoritmo solo dispusiese de 2 qubits: poseemos cuatro estados distintos, siendo ψ aquel con el menor coste, ver 33.

En esta figura, es evidente que no se destaca un único estado, sino que hay múltiples. Esta variedad se debe a que hemos introducido una fase específica a cada uno de ellos. Sin embargo, lo más relevante a destacar es que el estado $\psi_{t\prime}$ acumula la mayor fase de todos. Aunque no alcanza un valor de fase exacto de π, como en 6, pero es indiscutible que $\psi_{t\prime}$ ostenta la mayor fase entre todos los estados considerados.

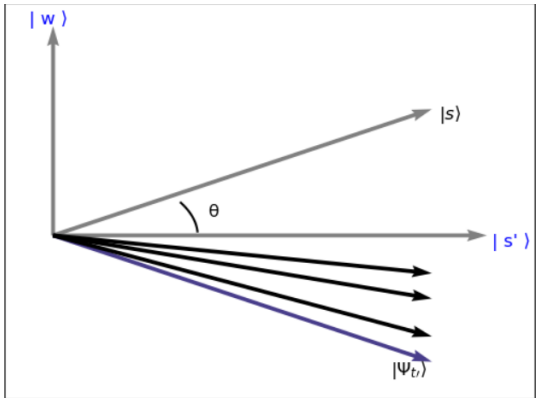

Figura 33: Ejemplo de lo que está ocurriendo con los estados en el oráculo diseñado

Tras la aplicación del difusor, el estado $\psi_{t\prime\prime}$ se acerca al estado ganador, ver 34. A raíz de varias iteraciones, este estado converge hacia al estado asociado con el menor coste. Como resultado, poseerá una amplitud de probabilidad más elevada, lo que se reflejará en una mayor frecuencia de medición.

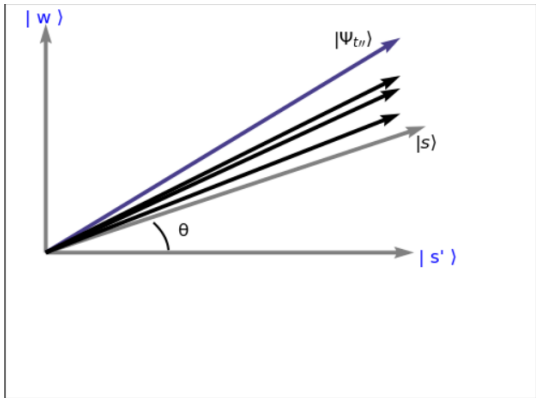

Figura 34: Tras aplicar el difusor.

A partir de las simulaciones en condiciones ideales, hemos notado que el sistema es efectivo en un 75 % de los casos, identificando con frecuencia el camino de menor costo. Específicamente, 4 iteraciones resultaron ser el número óptimo en la mayoría de situaciones. Sin embargo, cuando ciertos valores se aproximaban a los dos radianes y medio, incrementar a 5 iteraciones mostró resultados más favorables. Este fenómeno nos lo podría explicar la fórmula expresada anteriormente, la formula para obtener el número adecuado de iteraciones para Grover, 10. Tras realizar los cálculos, se determinó un valor de 4; sin embargo, el resultado exacto obtenido fue 4.5 y según la fórmula, es necesario redondear al entero inferior más cercano. El propio artículo advierte que si se desconoce el número de soluciones posibles, esta fórmula podría arrojar errores significativos. Por lo tanto, se observó que el algoritmo funcionaba óptimamente con 4 iteraciones en casos donde la señal era, como máximo, de 2 radianes y otros estados estaban cerca del estado solución. Sin embargo, en ocasiones si había un resultado que destacase notablemente sobre los demás, y el resto de fases eran bastante menores al de la solución, el sistema parecía ser más eficaz con 5 iteraciones.

Al implementar estas soluciones en un ordenador cuántico, su eficacia disminuyó considerablemente, yo lo achaco a las limitaciones actuales de esta tecnología. Cabe destacar que, en situaciones donde en simulación había trabajado mejor con 5 iteraciones, en un ordenador cuántico real con solo cuatro iteraciones, se obtenía un resultado que podría llegar a ser satisfactorio.

6. Conclusión y futuras líneas de investigación

Hemos desarrollado un oráculo diseñado para cinco ciudades que funciona adecuadamente en un simulador ideal. Además, hemos introducido una asignación de fases que ha demostrado su eficacia, el cual a pesar de que el diseño actual se limita a cinco ciudades, el código podría adaptarse fácilmente para abordar hasta N ciudades. Si bien el algoritmo ha mostrado ser efectivo en simulaciones ideales, al implementarlo en un ordenador cuántico real, los resultados han diferido en todas excepto en una.

Tras demostrar que el algoritmo funciona en escenarios ideales, es esencial investigar en detalle cómo el oráculo y el amplificador gestionan los diferentes estados. Aunque hemos desarrollado una asignación de fases que se puede generalizar con facilidad, sería recomendable trabajar en un oráculo más escalable, ya que la falta de escalabilidad es una limitación en el oráculo desarrollado en este estudio. Sería conveniente revisar las fases asignadas, a pesar de que funcionan en un contexto simulado en la mayoría de las ocasiones, no ocurre lo mismo en un ordenador cuántico real. Por ello, una opción sería intentar refinar la asignación de fases para obtener un algoritmo verdaderamente funcional en un entorno real.

7. Bibliografía

Referencias

[1] Ryu J. Lee C. Yoo S. Lim J. Lee J. Bang, J. A quantum heuristic algorithm for the traveling salesman problem. *Journal of the Korean Physical Society, 61(12), 1944-1949.*, (2012). URL: https://doi.org/10.3938/jkps.61.1944.

[2] Høyer y Tapp Boyer, Brassard. Tight bounds on quantum searching. *School of Electronic Engineering Computer Systems, University of Wales, Bangor, Gwynedd LL57 1UT, United Kingdom*, (1997).

[3] Michael R Garey and David S Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W.H. Freeman, 1979.

[4] Lov K. Grover. A fast quantum mechanical algorithm for database search. In *Proceedings of the 28th Annual ACM Symposium on Theory of Computing (STOC '96)*, pages 212–219, New York, NY, USA, 1996. ACM. doi:10.1145/237814.237866.

[5] Antonio Díaz Hernández. Algoritmo cuántico para solución eficiente de problemas de enrutamiento. *Universidad Politécnica de Cartagena*, 2022. URL: https://repositorio.upct.es/bitstream/handle/10317/11632/tfg-dia-alg.pdf?sequence=1.

[6] A. H. Land and A. G. Doig. An automatic method of solving discrete programming problems. *Econometrica: Journal of the Econometric Society*, pages 497–520, 1960.

[7] Maad Mijwil. Travelling salesman problem mathematical description. *College of Science, Baghdad University*, 2016. doi:10.13140/RG.2.2.27113.62563.

[8] Michael A Nielsen and Isaac L Chuang. *Quantum Computation and Quantum Information*. Cambridge University Press, 2000.

[9] Learn Qiskit. Quantum circuits. https://learn.qiskit.org/course/ch-algorithms/quantum-circuits.

[10] Maximilian Schlosshauer. *Decoherence and the Quantum-to-Classical Transition*. Springer-Verlag Berlin Heidelberg, 2007.

[11] Jieao Zhu, Yihuai Gao, Hansen Wang, Tiefu Li, and Hao Wu. A realizable gas-based quantum algorithm for traveling salesman problem. 2022.

8. Anexo

```
from qiskit import QuantumCircuit, Aer, transpile
from qiskit.circuit.library import QFT
from qiskit.providers.aer import StatevectorSimulator
from qiskit.quantum_info import Statevector
from qiskit.visualization import plot_state_city
from math import pi
import numpy as np
from qiskit.circuit.library import XGate
from qiskit.visualization import plot_histogram



def coste2q(a,b, c) :
    qc = QuantumCircuit(2)

    qc.p(b, 0)  # Aplicar fase al qubit 01
    qc.p(c, 1)  # Aplicar fase al qubit 10
    qc.x(0)
    qc.x(1)
    qc.cp(a,0, 1) # Aplicar fase al qubit 00
    qc.x(0)
    qc.x(1)

    qc.cp(-(b+c),0, 1) # Eliminar fase al qubit 11

    return qc

def coste2_q(a,b, c, d) :
    qc = QuantumCircuit(2)

    qc.p(b, 0)  # Aplicar fase al qubit 01
    qc.p(c, 1)  # Aplicar fase al qubit 10
    qc.x(0)
    qc.x(1)
    qc.cp(a,0, 1) # Aplicar fase al qubit 00
    qc.x(0)
    qc.x(1)

    qc.cp(-(b+c+d),0, 1) # Eliminar fase al qubit 11

    return qc


def phasing():
    qc13 = QuantumCircuit(5)

    qc13.p(np.pi/2, 4)
    qc13.x(4)
    qc13.p(np.pi/2, 4)
    qc13.x(4)


    # Suma primera ciudad

    qc13.x(3)
    qc13.x(4)
    qc13.cp(m[0][1],3, 4) # Aplicar fase al qubit 00
    qc13.x(4)
```

```
qc13.x(3)

qc13.p(m[0][2], 3)  # Aplicar fase al qubit 01
qc13.p(m[0][3], 4)  # Aplicar fase al qubit 10

qc13.cp(m[0][4]-(m[0][2]+m[0][3]),3, 4) # Aplicar fase al qubit
    11

# Suma Segunda Ciudad



cPhase00=coste2_q(m[1][2], m[1][3], m[1][4], m[0][1]+np.pi/2-0.3)
    .to_gate().control(2)
qc13.x([3, 4])
qc13.append(cPhase00, [4, 3, 1, 2])
qc13.x([ 3, 4])


cPhase01=coste2_q(m[2][3], m[2][4], m[2][1], m[0][2]+np.pi/2-0.3)
    .to_gate().control(2)
qc13.cx(3, 4)
qc13.append(cPhase01, [4, 3, 1, 2])
qc13.cx(3, 4)


cPhase10=coste2_q(m[3][4], m[3][1], m[3][2], m[0][3]+np.pi/2-0.3)
    .to_gate().control(2)
qc13.cx(4,  3)
qc13.append(cPhase10, [4, 3, 1 ,2])
qc13.cx(4, 3)



cPhase11=coste2_q(m[4][1], m[4][2], m[4][3], m[0][4]+np.pi/2-0.3)
    .to_gate().control(2)
qc13.append(cPhase11, [4, 3, 1, 2])









#Suma Tercera Ciudad




cPhase00=coste2q(m[2][4], m[3][2], m[4][3]).to_gate().control(3)

qc13.x([3, 4 ])

qc13.append(XGate().control(4), [4,3,2, 1,0])

qc13.append(cPhase00, [4,3,0,1,2])

qc13.append(XGate().control(4), [4,3,2, 1,0])

qc13.x([3, 4])
```

```
cPhase01=coste2q(m[3][1], m[4][3], m[1][4]).to_gate().control(3)

qc13.cx( 3, 4)


qc13.append(XGate().control(4), [ 4,3,2, 1,0])
qc13.append(cPhase01, [4,3,0,1,2])
qc13.append(XGate().control(4), [ 4,3,2, 1, 0])


qc13.cx( 3, 4)







cPhase10=coste2q(m[4][2], m[1][4], m[2][1]).to_gate().control(3)
qc13.cx(4, 3)

qc13.append(XGate().control(4), [4,3,2, 1,0])

qc13.append(cPhase10, [4,3,0,1,2])

qc13.append(XGate().control(4), [4,3,2, 1,0])

qc13.cx(4, 3)







cPhase11=coste2q(m[1][3], m[2][1], m[3][2] ).to_gate().control(3)
qc13.append(XGate().control(4), [4,3,2, 1,0])

qc13.append(cPhase11, [4,3,0,1,2])

qc13.append(XGate().control(4), [4,3,2, 1,0])




qc13.x(0)   #qubit 0 pasa a asignar cuando est  a 0

cPhase00=coste2q(m[2][3], m[3][4], m[4][2]).to_gate().control(3)

qc13.x([3, 4 ])

qc13.append(XGate().control(4), [4,3,2, 1,0])

qc13.append(cPhase00, [4,3,0,1,2])

qc13.append(XGate().control(4), [4,3,2, 1,0])
```

```
qc13.x([3, 4])




cPhase01=coste2q(m[3][4], m[4][1], m[1][3]).to_gate().control(3)

qc13.cx( 3, 4)


qc13.append(XGate().control(4), [ 4,3,2, 1,0])
qc13.append(cPhase01, [4,3,0,1,2])
qc13.append(XGate().control(4), [ 4,3,2, 1, 0])


qc13.cx( 3, 4)






cPhase10=coste2q(m[4][1], m[1][2], m[2][4]).to_gate().control(3)
qc13.cx(4, 3)

qc13.append(XGate().control(4), [4,3,2, 1,0])

qc13.append(cPhase10, [4,3,0,1,2])

qc13.append(XGate().control(4), [4,3,2, 1,0])

qc13.cx(4, 3)




cPhase11=coste2q(m[1][2], m[2][3], m[3][1] ).to_gate().control(3)
qc13.append(XGate().control(4), [4,3,2, 1,0])

qc13.append(cPhase11, [4,3,0,1,2])

qc13.append(XGate().control(4), [4,3,2, 1,0])




qc13.x(0)

# Cuarta Ciudad



cPhase00=coste2q(m[4][3], m[2][4], m[3][2]).to_gate().control(3)
```

```
qc13.x([3, 4 ])

qc13.append(XGate().control(4), [4,3,2, 1,0])

qc13.append(cPhase00, [4,3,0,1,2])

qc13.append(XGate().control(4), [4,3,2, 1,0])

qc13.x([3, 4])




cPhase01=coste2q(m[1][4], m[3][1], m[4][3]).to_gate().control(3)

qc13.cx( 3, 4)


qc13.append(XGate().control(4), [ 4,3,2, 1,0])
qc13.append(cPhase01, [4,3,0,1,2])
qc13.append(XGate().control(4), [ 4,3,2, 1, 0])


qc13.cx( 3, 4)







cPhase10=coste2q(m[2][1], m[4][2], m[1][4]).to_gate().control(3)
qc13.cx(4, 3)

qc13.append(XGate().control(4), [4,3,2, 1,0])

qc13.append(cPhase10, [4,3,0,1,2])

qc13.append(XGate().control(4), [4,3,2, 1,0])

qc13.cx(4, 3)




cPhase11=coste2q(m[3][2], m[1][3], m[2][1] ).to_gate().control(3)
qc13.append(XGate().control(4), [4,3,2, 1,0])

qc13.append(cPhase11, [4,3,0,1,2])

qc13.append(XGate().control(4), [4,3,2, 1,0])
```

```
qc13.x(0)
cPhase00=coste2q(m[3][4], m[4][2], m[2][3]).to_gate().control(3)

qc13.x([3, 4 ])

qc13.append(XGate().control(4), [4,3,2, 1,0])

qc13.append(cPhase00, [4,3,0,1,2])

qc13.append(XGate().control(4), [4,3,2, 1,0])

qc13.x([3, 4])




cPhase01=coste2q(m[4][1], m[1][3], m[3][4]).to_gate().control(3)

qc13.cx( 3, 4)


qc13.append(XGate().control(4), [ 4,3,2, 1,0])
qc13.append(cPhase01, [4,3,0,1,2])
qc13.append(XGate().control(4), [ 4,3,2, 1, 0])


qc13.cx( 3, 4)







cPhase10=coste2q(m[1][2], m[2][4], m[4][1]).to_gate().control(3)
qc13.cx(4, 3)

qc13.append(XGate().control(4), [4,3,2, 1,0])

qc13.append(cPhase10, [4,3,0,1,2])

qc13.append(XGate().control(4), [4,3,2, 1,0])

qc13.cx(4, 3)




cPhase11=coste2q(m[2][3], m[3][1], m[1][2] ).to_gate().control(3)
qc13.append(XGate().control(4), [4,3,2, 1,0])

qc13.append(cPhase11, [4,3,0,1,2])

qc13.append(XGate().control(4), [4,3,2, 1,0])
```

```
qc13.x(0)


#Quinta ciudad


cPhase00=coste2q(m[3][0], m[4][0], m[2][0]).to_gate().control(3)

qc13.x([3, 4 ])

qc13.append(XGate().control(4), [4,3,2, 1,0])

qc13.append(cPhase00, [4,3,0,1,2])

qc13.append(XGate().control(4), [4,3,2, 1,0])

qc13.x([3, 4])




cPhase01=coste2q(m[4][0], m[1][0], m[3][0]).to_gate().control(3)

qc13.cx( 3, 4)


qc13.append(XGate().control(4), [ 4,3,2, 1,0])
qc13.append(cPhase01, [4,3,0,1,2])
qc13.append(XGate().control(4), [ 4,3,2, 1, 0])


qc13.cx( 3, 4)







cPhase10=coste2q(m[1][0], m[2][0], m[4][0]).to_gate().control(3)
qc13.cx(4, 3)

qc13.append(XGate().control(4), [4,3,2, 1,0])

qc13.append(cPhase10, [4,3,0,1,2])

qc13.append(XGate().control(4), [4,3,2, 1,0])

qc13.cx(4, 3)




cPhase11=coste2q(m[2][0], m[3][0], m[1][0] ).to_gate().control(3)
qc13.append(XGate().control(4), [4,3,2, 1,0])

qc13.append(cPhase11, [4,3,0,1,2])
```

```
qc13.append(XGate().control(4), [4,3,2, 1,0])


qc13.x(0)

cPhase00=coste2q(m[4][0], m[2][0], m[3][0]).to_gate().control(3)

qc13.x([3, 4 ])

qc13.append(XGate().control(4), [4,3,2, 1,0])

qc13.append(cPhase00, [4,3,0,1,2])

qc13.append(XGate().control(4), [4,3,2, 1,0])

qc13.x([3, 4])




cPhase01=coste2q(m[1][0], m[3][0], m[4][0]).to_gate().control(3)

qc13.cx( 3, 4)


qc13.append(XGate().control(4), [ 4,3,2, 1,0])
qc13.append(cPhase01, [4,3,0,1,2])
qc13.append(XGate().control(4), [ 4,3,2, 1, 0])


qc13.cx( 3, 4)







cPhase10=coste2q(m[2][0], m[4][0], m[1][0]).to_gate().control(3)
qc13.cx(4, 3)

qc13.append(XGate().control(4), [4,3,2, 1,0])

qc13.append(cPhase10, [4,3,0,1,2])

qc13.append(XGate().control(4), [4,3,2, 1,0])

qc13.cx(4, 3)




cPhase11=coste2q(m[3][0], m[1][0], m[2][0] ).to_gate().control(3)
qc13.append(XGate().control(4), [4,3,2, 1,0])

qc13.append(cPhase11, [4,3,0,1,2])
```

```
    qc13.append(XGate().control(4), [4,3,2, 1,0])


    qc13.x(0)



    return(qc13)



( Fuente :qiskit.org )
def diffuser (nqubits) :
    qc = QuantumCircuit (nqubits)
    # Apply transformation | s> => | 0 0 . . 0 > (H=g a t e s )
    for qubit in range (nqubits) :
        qc.h (qubit)
    # Apply transformation | 0 0 . . 0 > => | 1 1 . . 1 > (X=g a t e
       s )
    for qubit in range ( nqubits ) :
        qc.x( qubit )
    # Do multi=controlled=Zgate
    qc.h(nqubits-1)
    qc.mct ( list ( range ( nqubits -1) ) , nqubits -1)
    # multi=controlled=toffoli

    qc.h(nqubits -1)
    # Apply transformation | 11..1 > => | 00..0>
    for qubit in range (nqubits ) :
        qc.x (qubit )
    # Apply transformation | 00. . 0 > => | s>
    for qubit in range (nqubits ) :
        qc.h (qubit )
    # We will returnt the diffuser as a gate
    U_s = qc.to_gate()
    U_s.name = " I "
    return U_s

    import numpy as np
from fractions import Fraction

def assign_intervals(matrix):
    min_val = np.min(matrix)
    max_val = np.max(matrix)
    interval = (max_val - min_val) / 5
    intervals = [min_val + i*interval for i in range(5)]  # Creamos 5
        puntos de intervalo
    intervals.append(max_val + 1)  # A adimos un valor extra a los
       intervalos para asegurar que el max_val sea incluido
    pi_values = [-np.pi/10, -np.pi/15, 0, +np.pi/15, np.pi/10]
    pi_values.reverse()
    new_matrix = np.zeros((5,5))

    for i in range(5):
        for j in range(5):
            for k in range(5):
                if intervals[k] <= matrix[i][j] < intervals[k+1]:  #
                    Ajusta la condici n para incluir el l mite
                    superior del  ltimo  intervalo
                     new_matrix[i][j] = pi_values[k]
```

```
                    break

    return new_matrix

# Creaci n de la matriz aleatoria con valores entre 0 y 10
matrix = np.random.randint(0, 11, size=(5, 5))

# Asignar los intervalos
m = assign_intervals(matrix)


tsp5 = QuantumCircuit(5)
tsp5.h([0, 1, 2, 3, 4])
phase = phasing().to_gate()
phase.name = "C"
for i in range(4): #asignaci n de iteraciones
    tsp5.append(phase, [0, 1, 2, 3, 4])
    tsp5.append(diffuser(5), [0, 1, 2, 3, 4])
tsp5.measure_all()
aer_sim = Aer.get_backend('aer_simulator')
transpiled_grover_circuit = transpile(tsp5, aer_sim)
results = aer_sim.run(transpiled_grover_circuit).result()
counts = results.get_counts()
plot_histogram(counts, title='First_iteration')
```

```
from qiskit import  execute, Aer
from qiskit.quantum_info import  partial_trace
from qiskit.visualization import plot_bloch_multivector,
    plot_state_city
import numpy as np

np.set_printoptions(precision=2, suppress=True)
qc13 = QuantumCircuit(5)
phase = phasing().to_gate()

for i in range(5):
    qc13.h(i)

qc13.append(phase, [0, 1, 2, 3, 4])



simulator = Aer.get_backend('statevector_simulator')
result = execute(qc13, simulator).result()
statevector = result.get_statevector(qc13)

# Fase del estado base (0011)
base_phase = np.angle(statevector[6])

# Calcular la diferencia de fase de cada estado respecto al estado
    base
phase_diff = np.angle(statevector) - base_phase

print("Diferencias de fase:", phase_diff)


print(qc13)
```