Q6

QX06-BernsteinVaziraniAlgorithmComplete.ipynb
File

Necesito que me expliques detalladamente cada una de las celdas del siguiente proyecto

Aquí tienes una explicación detallada, celda por celda, de cómo está construido el notebook que implementa el algoritmo de Bernstein–Vazirani en Qiskit. Cada celda se acompaña de la teoría necesaria para entender su propósito.

Celda 0 (código)

```
python

from qiskit import *
```

Qué hace
Importa todo el contenido principal de Qiskit: proveedores de backends (simuladores, hardware), objetos para construir circuitos (**`QuantumCircuit`**), y funciones para ejecutar circuitos (**`execute`**, etc.).
Por qué es importante
Qiskit es el framework de IBM para programación cuántica en Python. Con esta instrucción, dispones de todas las clases y funciones necesarias sin tener que importar módulo por módulo.

Celda 1 (código)

```
python

%matplotlib inline
from qiskit.tools.visualization import plot_histogram
```

Qué hace

- **`%matplotlib inline`**: magia de Jupyter que permite incrustar gráficos de Matplotlib directamente en la salida de la celda.
- **`plot_histogram`**: importa la función para dibujar histogramas de resultados cuánticos (distribuciones de conteos).

Teoría
Tras ejecutar un circuito cuántico y obtener las “frecuencias” de los distintos estados medidos, un histograma es la forma más intuitiva de visualizar cuántas veces ha salido cada bitstring.

Celda 2 (código)

```
python

secretNumber = '10101010100'
```

Qué hace
Define en Python la cadena de bits que hace de "secreto" en el problema de Bernstein–Vazirani. Cada carácter `'1'` o `'0'` corresponde a un bit de la serie **s**.

Teoría del problema
El problema de Bernstein–Vazirani consiste en descubrir un bitstring oculto **s** consultando una función oráculo f que, para una entrada $x \in \{0,1\}^n$, devuelve

$$f(x) = s \cdot x \bmod 2 = \bigoplus_{i=0}^{n-1} (s_i \wedge x_i).$$

Clásicamente se necesitan n consultas para descubrir todos los bits de **s**, pero con un solo circuito cuántico y una única llamada al oráculo podemos leer **s** completa de una vez.

Celda 3 (código)

```
python

circuit = QuantumCircuit(len(secretNumber)+1, len(secretNumber))
```

Qué hace
Crea un objeto `QuantumCircuit` con:

- **n+1 qubits**: los primeros n para codificar la entrada x, y un qubit extra (ancilla) preparado en $|1\rangle$ para el oráculo.
- **n bits clásicos** para almacenar el resultado de medir cada uno de los n qubits de entrada.

Por qué n+1 qubits y no solo n
El qubit ancilla permite implementar la suma modular de un bit de la entrada con el bit secreto, sin destruir la superposición de los qubits de entrada.

Celda 4 (código)

```
python


circuit.h(range(len(secretNumber)))
circuit.x(len(secretNumber))
circuit.h(len(secretNumber))

circuit.barrier()

for index, one in enumerate(reversed(secretNumber)):
    print(f"index{index} is {one}")
    if one == "1":
        circuit.cx(index, len(secretNumber))
```

```
circuit.barrier()

circuit.h(range(len(secretNumber)))

circuit.barrier()

circuit.measure(range(len(secretNumber)), range(len(secretNumber)))
```

Desglose paso a paso

1. **Hadamard inicial en los primeros n qubits**

   ```
   python

   circuit.h(range(len(secretNumber)))
   ```

   — Transforma cada qubit $|0\rangle$ en $\frac{|0\rangle+|1\rangle}{\sqrt{2}}$, generando una superposición de todos los 2^n posibles valores de x.

2. **Preparación del qubit ancilla**

   ```
   python

   circuit.x(len(secretNumber))
   circuit.h(len(secretNumber))
   ```

   — Primero un X para pasar $|0\rangle \rightarrow |1\rangle$. Luego un H para convertirlo en $\frac{|0\rangle-|1\rangle}{\sqrt{2}}$. Esta fase es necesaria para que el oráculo implemente $f(x)$ sumando el resultado en fase al ancilla.

3. **Barrera**

   ```
   python

   circuit.barrier()
   ```

   — Marca visualmente un corte entre la preparación del estado y la implementación del oráculo.

4. **Implementación del oráculo**

   ```
   python

   for index, one in enumerate(reversed(secretNumber)):
       if one == "1":
           circuit.cx(index, len(secretNumber))
   ```

   — Para cada bit secreto `s[i] = 1`, añade una compuerta CNOT entre el qubit de entrada i y el qubit ancilla.
   — Debido a la preparación en fase del ancilla, cada CNOT introduce un factor de $(-1)^{x_i}$ cuando $s_i = 1$, de modo que el conjunto de CNOTs en total aplica $|x\rangle|-\rangle \mapsto (-1)^{s\cdot x}\,|x\rangle|-\rangle$.

5. **Segunda ronda de Hadamard**

```
python

circuit.barrier()
circuit.h(range(len(secretNumber)))
```

— Aplica H otra vez a los qubits de entrada. Por álgebra de transformadas de Hadamard, este bloque convierte

$$\frac{1}{2^{n/2}}\sum_{x}(-1)^{s\cdot x}|x\rangle \xrightarrow{H^{\otimes n}} |s\rangle.$$

Así el estado de los primeros n qubits colapsa al bitstring secreto **s**.

6. **Medición**

```
python

circuit.barrier()
circuit.measure(range(len(secretNumber)), range(len(secretNumber)))
```

— Mide cada qubit de entrada; con probabilidad 1 obtendremos exactamente el bitstring `secretNumber`.

Celda 5 (código)

```
python

circuit.draw(output='mpl')
```

Qué hace

Genera un diagrama visual (usando Matplotlib) del circuito cuántico completo, con todas las compuertas y barreras. Es muy útil para verificar gráficamente la estructura del algoritmo.

Celda 6 (código)

```
python

simulator = Aer.get_backend('qasm_simulator')
result = execute(circuit, backend=simulator, shots=1).result()
counts = result.get_counts()
print(counts)
```

Qué hace

1. Selecciona el simulador de Qiskit Aer que emula un backend QASM (cuando se miden los qubits).
2. Ejecuta el circuito **una sola vez** (`shots=1`), porque sabemos que el algoritmo da el resultado correcto con probabilidad 1.
3. Obtiene un diccionario `counts` con la frecuencia de cada resultado medido, p.ej. `{'10101010100': 1}`.

Celda 7 (código)

```
python

plot_histogram(counts)
```

Qué hace
Dibuja un histograma de los resultados almacenados en `counts`. Dado que `shots=1`, verás una única barra en la posición del bitstring secreto.

Celdas 8, 9 y 10

Estas tres celdas aparecen en blanco (sin contenido). Probablemente sean espacios reservados o restos de edición. No alteran el funcionamiento del notebook y pueden eliminarse para mantenerlo limpio.

Conclusión

Con estas 7 celdas activas (más las 3 vacías al final), el notebook:

1. **Prepara** los qubits necesarios para el algoritmo de Bernstein–Vazirani.
2. **Implementa** el oráculo con compuertas CNOT según el bitstring secreto.
3. **Transforma** la superposición de fase en el registro de entrada para obtener directamente el secreto.
4. **Mide** y **visualiza** el resultado con un histograma.

Este ejemplo ilustra de forma clara cómo un solo circuito cuántico puede resolver en una única consulta un problema que requeriría n consultas clásicas.