Q5

QX05-BernsteinVaziraniAlgorithmSimple.ipynb
File

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

Aquí tienes una explicación detallada, celda por celda, del notebook **QX05-BernsteinVaziraniAlgorithmSimple.ipynb**:

Celda 1

```
python

from qiskit import *
```

Qué hace y por qué es importante

- Importa en un solo paso todas las clases y funciones principales de Qiskit (p. ej. **`QuantumCircuit`**, **`Aer`**, **`execute`**, etc.).
- Esto facilita escribir más adelante **`QuantumCircuit(...)`**, **`Aer.get_backend(...)`**, **`execute(...)`** sin importaciones individuales.

Celda 2

```
python

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

1. **`%matplotlib inline`**
   - Es una “magic” de Jupyter que hace que los gráficos de Matplotlib (o de Qiskit) se muestren directamente dentro de la celda.
2. **`from qiskit.tools.visualization import plot_histogram`**
   - Importa la función **`plot_histogram`**, que permite representar gráficamente en barras la distribución de resultados de medición (counts) tras ejecutar el circuito.

Celda 3

```
python

secretNumber = '1000101'
```

- Aquí se define la **cadena binaria** oculta (el "secreto") de 7 bits, que el algoritmo de Bernstein–Vazirani debe descubrir.
- Cada carácter de la cadena corresponde a un bit $s_i \in \{0, 1\}$. En este caso:

```
ini

s = [1, 0, 0, 0, 1, 0, 1]
```

Celda 4

```
python

circuit = QuantumCircuit(8, 7)
```

- Se crea un **circuito cuántico** con:
  - **8 qubits** (7 qubits "de entrada" para codificar x y 1 qubit ancilla para la fase).
  - **7 bits clásicos** donde almacenaremos el resultado de medir los 7 qubits de entrada.
- **Convención**:
  - Qubit 0–6 $\rightarrow$ registro de entrada $|x\rangle$
  - Qubit 7 $\rightarrow$ ancilla (se usa para aplicar la función oracle $f(x)$)

Celda 5

```
python

circuit.h([0,1,2,3,4,5,6])    # 1) Poner los qubits de entrada en superposición
circuit.x(7)                  # 2) Preparar el qubit ancilla en |1⟩
circuit.h(7)                  # 3) Crear superposición de fase en el ancilla

circuit.barrier()             # ——————————— Separador visual en el diagrama ———————————

# 4) Oracle: aplicar CX desde cada qubit i (donde s_i = 1) al ancilla
circuit.cx(6, 7)              # bit 6 del secreto es 1
circuit.cx(2, 7)              # bit 2 del secreto es 1
circuit.cx(0, 7)              # bit 0 del secreto es 1

circuit.barrier()             # ———— Fin del oracle ————

circuit.h([0,1,2,3,4,5,6])    # 5) Transformada inversa de Hadamard en los qubits de entrada
circuit.measure([0,1,2,3,4,5,6], [0,1,2,3,4,5,6])
                              # 6) Medir cada qubit de entrada en su bit clásico correspondiente
```

Línea a línea

1. **`circuit.h([0…6])`**: lleva cada qubit $|0\rangle \rightarrow \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle)$. Prepara superposición en todo el espacio de entradas.

2. **`circuit.x(7)`**: cambia el ancilla de $|0\rangle \to |1\rangle$.
3. **`circuit.h(7)`**: Hadamard sobre ancilla $\to \frac{1}{\sqrt{2}}(|0\rangle - |1\rangle)$. Esta fase permitirá "kickback" tras los CNOT.
4. **Oracle (`circuit.cx`)**: por cada índice i tal que el bit secreto $s_i = 1$, se hace un CNOT del qubit i al ancilla (qubit 7). Así la fase del ancilla se invierte condicionalmente según la paridad $s \cdot x$.
5. **`circuit.h([0…6])`**: tras el oracle, volvemos a aplicar Hadamard en las 7 líneas de entrada. Esto reconvierte la información de fase en amplitudes, de modo que la medición revela directamente el string secreto.
6. **`circuit.measure(...)`**: mide cada qubit de entrada en su bit clásico; el vector de bits resultante debe coincidir con **`secretNumber`**.

Celda 6

```
python

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

- Genera y muestra el **diagrama de circuito** en formato Matplotlib.
- Verás en la parte superior las Hadamards iniciales, luego el ancilla con X y H, los CNOT del oracle y finalmente las Hadamards inversas y las mediciones.

Celda 7

```
python

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

1. **`Aer.get_backend('qasm_simulator')`**: selecciona el simulador de circuitos QASM (basado en muestreos).
2. **`execute(...)`**: envía el circuito al simulador, pidiendo **`shots=1`** (una sola corrida es suficiente en BV, pues el resultado es determinista).
3. **`get_counts()`**: devuelve un diccionario con las cuentas de cada cadena de bits medida.
4. **`print(counts)`**: muestra algo como **`{'1000101': 1}`**, confirmando que el algoritmo ha recuperado el secreto.

Celdas 8–11

- **Están vacías.** Probablemente quedaron como placeholders o para agregar código/marcas de separación. No ejecutan ninguna operación.

Resumen del flujo

1. **Preparación**: imports, visualización y definición del secreto.
2. **Construcción**: circuito con superposiciones, oracle y Hadamards finales.
3. **Visualización**: diagrama del circuito.
4. **Ejecución**: simulación y obtención del resultado.

Este notebook implementa el **algoritmo de Bernstein–Vazirani**, que determina en una sola evaluación (un solo "shot") un string secreto s escondido en una función lineal booleana $f(x) = s \cdot x \pmod 2$. Si tienes dudas sobre algún paso teórico o quieras profundizar en la fase de oracle o de transformadas de Hadamard, ¡dímelo y lo vemos!