UVSQ

Informatique quantique

Réalisation du TD2 - Algorithme de Bernstein-Vazirani

8 septembre 2025

Table des matières

1 Question 1.1 : Construction d'un oracle générique

Dans cette première partie, nous souhaitons construire la matrice unitaire U associée à un oracle quantique. L'oracle réalise la transformation :

$$|x\rangle|y\rangle \mapsto |x\rangle|y \oplus f(x)\rangle,$$

où x est un entier sur n qubits et y est un entier sur m qubits.

— Nous avons choisi de construire explicitement la matrice, de taille $2^{n+m} \times 2^{n+m}$, afin de *contrôler la transformation de chaque* état d'entrée (x, y).

— On itère donc sur toutes les valeurs de x et y pour placer des 1 aux positions correctes de la matrice, ce qui correspond exactement à la transformation souhaitée.

— Nous vérifions l'unitarité en testant que $U \times U^T = I$.

1.1 Code Python

```python
import numpy as np

def build_matrix(n: int, m: int, f) -> np.ndarray:
    """
    Construit la matrice d'un oracle quantique associ     une fonction bool enne f.

    Param tres:
        n (int): Nombre de bits d'entr e
        m (int): Nombre de bits de sortie
        f (function): Fonction bool enne prenant un entier x (cod  sur n bits)
                      et retournant un entier y (cod  sur m bits)

    Retourne:
        np.ndarray: Matrice unitaire repr sentant l'oracle
    """
    N = 2 ** (n + m)  # Taille totale de la matrice (nombre d' tats )
    U = np.eye(N)

    # Parcours de tous les  tats  d'entr e pour les n bits et m bits ancillaires
    for x in range(2 ** n):
        fx = f(x)  # Calcul de f(x)
        for y in range(2 ** m):
            # Construction de l' tat  d'entr e : concat nation de x et y
            input_state = (x << m) | y
            # Construction de l' tat  de sortie : concat nation de x et (y     f(x))
            output_state = (x << m) | (y ^ fx)
            # Mise    jour de la matrice de permutation
            U[input_state, input_state] = 0
            U[input_state, output_state] = 1

    return U

def is_unitary(U: np.ndarray) -> bool:

    return np.allclose(U @ U.T, np.eye(U.shape[0]))


```

```python
# Exemple d'utilisation
n, m = 2, 1

def example_f(x: int) -> int:

    #Exemple de fonction bool enne qui retourne le dernier bit de x.

    return x & 1

U = build_matrix(n, m, example_f)
print("Matrice oracle U :\n", U)
print("Est unitaire ?", is_unitary(U))
```

Listing 1 – Construction de la matrice d'un oracle quantique

Pour illustrer la transformation réalisée par l'oracle, considérons $n = 2$ et $m = 1$ avec $f(x) = x \,\&\, 1$. La table suivante présente, pour chaque valeur de x et y, la valeur de $f(x)$ et le résultat de $y \oplus f(x)$:

x (décimal)	x (binaire sur 2 bits)	$f(x)$	y	$y \oplus f(x)$
0	00	0	0	0
0	00	0	1	1
1	01	1	0	1
1	01	1	1	0
2	10	0	0	0
2	10	0	1	1
3	11	1	0	1
3	11	1	1	0

Vérification de l'Unitarité

Une matrice U est unitaire si :

$$U \times U^T = I,$$

où I est la matrice identité
Pour notre oracle, la matrice U est donnée par :

$$U = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \end{pmatrix}.$$

on a :

$$U^T = U.$$

Nous calculons alors :

$$U \times U^T = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix} = I.$$

Cette égalité, $U \times U^T = I$, confirme que la matrice U est bien unitaire.

2 Question 1.2 : Implémentation de l'algorithme de Bernstein-Vazirani

L'algorithme de Bernstein-Vazirani permet de retrouver le bitstring secret s qui définit une fonction

$$f(x) = s \cdot x \pmod 2.$$

- Allouer $n+1$ qubits (n de données et 1 ancilla).
- Initialiser l'ancilla en $|1\rangle$ (porte X) et appliquer H sur tous les qubits.
- Appliquer l'oracle via des portes $CNOT$ conditionnées par les qubits de données.
- Réappliquer H sur les qubits de données.
- Mesurer les qubits de données pour révéler le secret avec une probabilité 1.

Le code ci-dessous illustre l'implémentation de l'algorithme de Bernstein-Vazirani sur 6 qubits (5 qubits de données et 1 qubit ancillaire) avec un secret.

2.1 Implémentation en Python

```python
from qat.lang.AQASM import Program, H, CNOT, X
from qat.qpus import PyLinalg
import numpy as np

def bernstein_vazirani_6qubits(n):

    n = 5  # Nombre de qubits de donn es
    prog = Program()
    qbits = prog.qalloc(n + 1)  # 5 qubits de donn es + 1 qubit ancillaire
    prog.apply(X, qbits[n])

    # Appliquer Hadamard sur les qubits de donn es
    for i in range(n):
        prog.apply(H, qbits[i])

    # Appliquer l'oracle
    # Fonction bool enne f(x) = 111011
    prog.apply(CNOT, qbits[0], qbits[n])
    prog.apply(CNOT, qbits[1], qbits[n])
    prog.apply(CNOT, qbits[2], qbits[n])
    prog.apply(CNOT, qbits[4], qbits[n])
```

```
22
23
24      # Appliquer Hadamard sur les qubits de donn es
25      for i in range(n):
26          prog.apply(H, qbits[i])
27
28
29  circuit = prog.to_circ()
30
31  circuit.display()
32
33  job = circuit.to_job()
34  job.nbshots = 1
35
36  qpu = PyLinalg()
37  result = qpu.submit(job)
38
39  print("\n tats et probabilit s :")
40  for sample in result:
41      print("State", sample.state, "with amplitude", sample.amplitude,
42            "and probability", round(sample.probability * 100, 2), "%")
```

Circuit quantique résultant

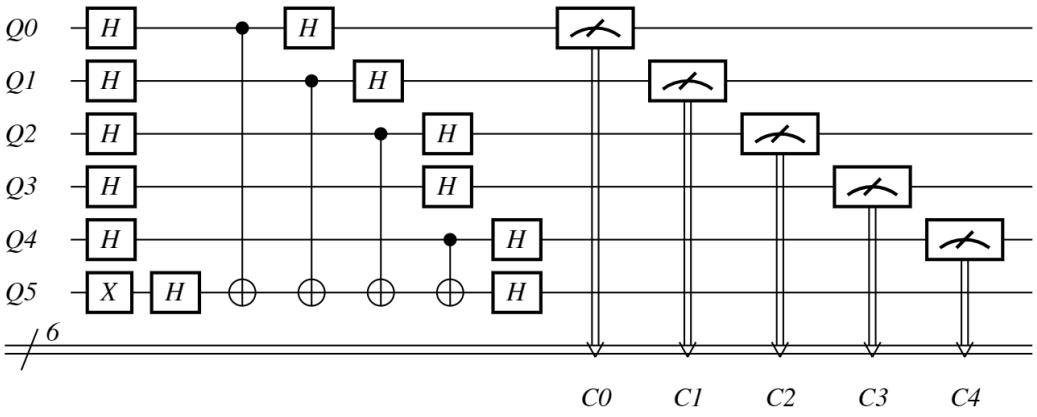

FIGURE 1 – Circuit quantique pour l'algorithme de Bernstein-Vazirani.

Une fois le programme exécuté, nous obtenons la sortie suivante :

```
State |111011> with probability 100.0 %
```

En considérant que le dernier bit correspond au qubit ancillaire, nous ne retenons que les 5 premiers bits, ce qui nous donne le bitstring $|11101\rangle$,il représente le secret que nous souhaitions retrouver. $(11101)_2 = (29)_{10}$.

3 Question 1.3 et 1.4 : Encodage d'un oracle pour n'importe quel valeur secrete

3.1 Code Python

```python
from qat.lang.AQASM import Program, H, CNOT, X
from qat.qpus import PyLinalg

def bernstein_vazirani_custom(n, secret):
    """
    Impl mente l'algorithme de Bernstein-Vazirani pour un secret donn .

    Param tres :
        n (int)    : Nombre de qubits de donn es (sans compter l'ancilla).
        secret (int): Secret    encoder sur n bits.

    L'algorithme utilise un total de n+1 qubits (les n qubits de donn es + 1 qubit ancilla).
    """
    size = n + 1  # Nombre total de qubits
    prog = Program()
    qbits = prog.qalloc(size)

    # Appliquer la porte X sur le dernier qubit (ancilla)
    prog.apply(X, qbits[-1])

    # Appliquer une porte Hadamard sur tous les qubits
    for i in range(size):
        prog.apply(H, qbits[i])

    # Conversion du secret en une cha ne binaire sur n bits (sans inversion)
    secret_bin = format(secret, f'0{n}b')

    for i, bit in enumerate(secret_bin):
        if bit == '1':
            prog.apply(CNOT, qbits[i], qbits[-1])

    # Appliquer une seconde fois Hadamard sur tous les qubits
    for i in range(size):
        prog.apply(H, qbits[i])

    return prog

# Exemple d'utilisation :
n = 5          # Nombre de qubits de donn es
secret = 25    # Secret    encoder

# Construction du circuit avec l'algorithme de Bernstein-Vazirani
prog = bernstein_vazirani_custom(n, secret)
circuit = prog.to_circ()


circuit.display()

job = circuit.to_job()
```

```
job.nbshots = 1
qpu = PyLinalg()
result = qpu.submit(job)

for sample in result:
    print("State", sample.state, "with amplitude", sample.amplitude,
          "and probability", round(sample.probability * 100, 2), "%")
```

Listing 2 – Encodage d'un produit pointé

J'ai trouvé $|100111\rangle$ et cela correspond à la valeur :

$$(11001)_2 = (25)_{10}.$$

Le circuit resultant :

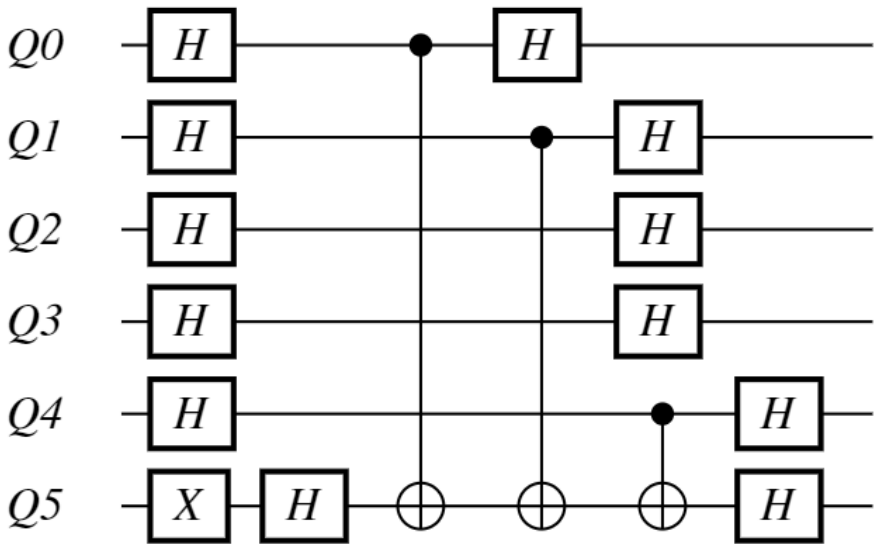

FIGURE 2 – Circuit quantique pour l'algorithme de Bernstein-Vazirani avec le secret `"11001"`.

Conclusion

Grâce à ces étapes, nous avons :

- Expliquer comment construire la matrice unitaire d'un oracle.
- Implémenter et justifié l'algorithme de Bernstein-Vazirani.
- Montrer la construction d'un oracle personnalisé.
- Vérifier le bon fonctionnement en identifiant correctement un secret choisi.

Ce TD illustre la puissance de l'approche quantique pour résoudre le problème de Bernstein-Vazirani en une seule requête à l'oracle (en théorie), soulignant l'avantage de la superposition et de l'interférence quantiques par rapport aux méthodes classiques.