BSQ 110 - Introduction aux sciences quantiques et à leurs applications

Deuxième projet de programmation

Construction et automatisation de circuits quantiques

1 Introduction

Pour ce second projet, vous allez utiliser la librairie Python pour le calcul quantique de IBMQ appelée Qiskit pour construire des circuits quantiques, les exécuter et analyser les résultats obtenus. Les circuits quantiques que vous allez aborder sont liés à trois algorithmes quantiques jouets qui sont les algorithmes de Deutsch, de Deustch-Jozsa et de Bernstein-Vazirani. Chacun d'eux fait intervenir un circuit quantique qui définit l'algorithme dont une partie, nommée l'oracle, définit le problème à résoudre. Vous devrez construire les circuits quantiques pour chacun de ces algorithmes et pour construire leurs oracles.

1.1 Mise en situation

Vous participez à un jeu où vous devez deviner un nombre mystère s situé entre 0 et 63. Vous devez poser un certain nombre de questions qui se répondent toutes par oui ou non pour acquérir de l'information sur le nombre mystère.

Une mauvaise stratégie classique Vous pourriez utiliser une stratégie systématique en demandant « Est-ce que le nombre est 0 ? » et ensuite « Est-ce que le nombre est 1 ? » et ainsi de suite. Vous pourriez également mélanger l'ordre des nombres pour éviter de vous faire prendre si le nombre mystère est 63. Avec ce type de stratégie, vous pourriez être chanceux et trouver le bon nombre au premier coup, ou être très malchanceux et le trouver en dernier. En moyenne, il vous faudra poser 32 questions pour y parvenir. Pouvez-vous faire mieux ?

Une meilleure stratégie classique Évidemment ! Une stratégie à employer pourrait ressembler à la suivante. Pour éliminer la moitié des nombres possibles, vous pourriez demander « Est-ce que le nombre est plus grand ou égal à 32 ? ». En utilisant ce type de question, vous pourrez diviser en deux le nombre de possibilités à chaque question. Pour un nombre situé entre 0 et 63 (2^6 possibilités) il vous faudra poser six questions. Il s'avère qu'on peut convertir ce type de question en une fonction $f_s(x)$ où s est le nombre à découvrir et x le nombre auquel le comparer. Cette fonction retourne 0 ou 1 (Faux ou Vrai). Ainsi, lorsqu'on affirme qu'il faut poser six questions, cela revient à évaluer la fonction $f_s(x)$ six fois.

Une stratégie quantique Il s'avère qu'il existe un algorithme quantique, l'algorithme de Bernstein-Vazirani, qui implémente une version quantique de ce jeu, et qui permet de deviner le nombre mystère avec une seule question ! Par une seule question, on sous-entend qu'on aura à évaluer la fonction $f_s(x)$ une seule fois.

La description du fonctionnement de cet algorithme fait intervenir son lot de mathématiques et de concepts nouveaux. Afin d'adoucir la pente vers la compréhension de cet algorithme, nous allons l'aborder étape par étape. D'abord, nous nous intéresserons à l'algorithme de Deustch qui fonctionne avec seulement deux qubits et qui permettra d'introduire l'évaluation de fonctions dans un ordinateur quantique. Nous passerons ensuite à l'algorithme de Deustch-Jozsa qui est une généralisation du premier et qui utilise un plus grand nombre de qubits. Nous pourrons finalement aborder l'algorithme de Bernstein-Vazirani.

1.2 Critères d'évaluation

Votre travail devra être remis sous la forme de scripts écrits dans un ou plusieurs fichiers `.py` ou encore sous la forme d'un *notebook* Jupyter `.ipynb`. Votre travail sera évalué selon les points suivants :

— La validité de vos solutions. Est-ce que ça fonctionne ?

— La clarté de votre code. Est-ce que votre code est facile à lire et à comprendre ? Vous pouvez consulter la section 1.3 « Bonnes habitudes de programmation » pour vous guider.

— La clarté de vos explications.

— L'originalité de votre solution. Avez-vous utilisé des approches alternatives à celles proposées ?

1.3 Bonnes habitudes de programmation

Pourquoi faire du bon code ? Qu'est-ce que c'est du bon code ? Un élément de réponse commun à ces deux questions est la lisibilité. Lire et comprendre du code écrit par quelqu'un d'autre (ou vous-même dans le passé) est rarement évident. Il est donc essentiel de maximiser sa clarté. Cela augmentera grandement les chances que vos solutions soient réutilisées dans le futur et donne de la valeur à votre travail. Avec cet objectif en tête, voici quelques bonnes habitudes de programmation à prendre :

— Utilisez des noms de variables informatifs.

— Soyez cohérents dans vos conventions de noms de variables, fonctions, etc.

— Un code qui se comprend sans commentaire est un bon code. Utilisez des commentaires lorsqu'il est difficile ou impossible de le rendre suffisamment lisible.

— Utilisez l'encapsulation. Évitez les répétitions. Si vous avez à changer comment vous faites quelque chose, il est préférable d'avoir à corriger votre code à un seul endroit.

— Faites en sorte que votre code soit réutilisable dans un autre contexte. Les outils développés pour construire la matrice pour un circuit devraient être utiles pour le second circuit également.

— Idéalement, chaque fonction/méthode ne devrait faire qu'une seule chose.

Cela étant dit, il y a une balance entre la qualité du code qu'on produit et la quantité. Si vous tentez d'écrire un code parfait du premier coup, vous risquez de progresser très lentement. Il peut parfois être plus efficace de rapidement programmer une solution qui fonctionne et, dans un deuxième temps, repasser sur le code afin d'améliorer sa lisibilité.

1.4 Suggestions

Nous suggérons fortement l'utilisation de GitHub pour gérer votre code et faciliter votre travail en équipe. Ensuite, tentez d'identifier les grands objectifs que votre équipe devra accomplir pour vous séparer les tâches. Partagez vos résultats à l'intérieur de votre équipe, cela vous permettra de valider vos idées et vos solutions.

2 Description du projet

Pour chacun des trois algorithmes (Deustch, Deustch-Jozsa et Bernstein-Vazirani), vous devez :

— Écrire une fonction qui construit le circuit quantique d'un oracle ;

— Écrire une fonction qui, étant donné un oracle, construit le circuit quantique de l'algorithme ;

— Exécuter vos circuits quantiques à l'aide d'un simulateur et analyser les résultats obtenus pour démontrer qu'ils fonctionnent ;

— Exécuter vos circuits quantiques à l'aide de prototypes d'ordinateur quantique IBMQ.

Les sections qui suivent décrivent avec plus de détails ce qui est attendu pour chacun des algorithmes.

Les trois algorithmes sont décrits au chapitre 4 des notes de cours et ne seront pas décrits ici. Il est donc essentiel de lire ces descriptions avant d'aborder ce projet. Vous pouvez également facilement trouver sur internet de nombreuses explications plus ou moins complètes pour vous aider à les approfondir votre comprehension.

Remarque importante Vous constaterez que les explications sont parfois mathématiquement lourdes, en particulier pour l'algorithme de Deustch-Jozsa, et peuvent sembler rébarbatives au premier abord. Ne vous laissez pas décourager par cela, car il n'est pas nécessaire d'acquérir une compréhension profonde du fonctionnement de ceux-ci pour compléter ce projet. Une fois que vous aurez complété les circuits quantiques construits et que ceux-ci vous seront plus familiers, les explications seront plus faciles à comprendre.

2.1 Algorithme de Deustch

L'algorithme de Deustch ne fait intervenir que deux qubits. Son implémentation devrait être assez simple.

- Construire les circuits quantiques pour les quatre oracles basés sur les quatre types de fonction.
- Programmer une fonction qui prend en entrée un nombre de 0 à 3 et qui retourne l'oracle correspondant sous la forme d'une porte quantique.
- Construire le circuit quantique pour l'algorithme de Deustch à partir de la porte quantique obtenue à l'étape précédente.
- Exécuter ce circuit et vérifier que les résultats concordent avec l'oracle utilisé.

2.2 Algorithme de Deustch-Jozsa

L'algorithme de Deustch-Jozsa peut faire intervenir un nombre arbitraire de qubits. Il faut donc généraliser son implémentation à un nombre arbitraire de qubits.

- Trouver la forme des circuits quantiques pour les oracles qui implémentent des fonctions constantes (il y en a 2). Votre réponse devrait pouvoir s'appliquer à un nombre arbitraire de qubits.
- Trouver au moins deux circuits quantiques pour les oracles qui implémentent des fonctions balancées. Votre réponse devrait pouvoir s'appliquer à un nombre arbitraire de qubits.
- Programmer des fonctions qui prenne en entrée un nombre de qubits retournent ces oracles sous la forme de portes quantiques à n qubits.
- Programmer une fonction qui construit le circuit quantique pour l'algorithme de Deustch-Jozsa à partir d'une porte quantique obtenue à l'étape précédente.
- Exécuter ce circuit et vérifier que les résultats concordent avec l'oracle utilisé.

2.3 Algorithme de Bernstein-Vazirani

L'algorithme de Bernstein-Vazirani utilise le même circuit quantique que l'algorithme de Deustch-Jozsa. L'implémentation de son oracle est par contre différente.

- Programmer une fonction qui prend en entrée un nombre de qubits n ainsi qu'un entier entre 0 et $2^n - 1$ et qui retourne l'oracle correspondant sous la forme d'une porte quantique.
- Programmer une fonction qui construit le circuit quantique pour l'algorithme de Bernstein-Vazirani à partir d'une porte quantique obtenue à l'étape précédente.
- Exécuter ce circuit et vérifier que les résultats concordent avec l'oracle utilisé.

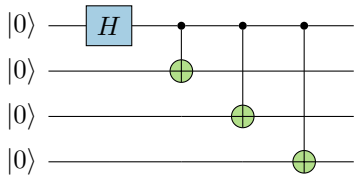


FIGURE 1 – Circuit GHZ à quatre qubits

3 Introduction aux circuits quantiques

Afin d'introduire le fonctionnement de la librairie Qiskit, nous allons programmer le circuit quantique présenté à la figure 1 qui permet prépare un état GHZ à quatre qubits

$$|\text{GHZ}\rangle = \frac{1}{\sqrt{2}}(|0000\rangle + |1111\rangle).$$

Il existe plusieurs approches pour construire un circuit quantique avec Qiskit. Nous commencerons avec une approche plus systématique pour mieux introduire toutes les parties d'un circuit quantique. Nous verrons ensuite qu'il existe des raccourcis pour produire rapidement des circuits quantiques simples.

3.1 Les registres

Un circuit quantique est sera composé d'un ou plusieurs registres quantiques et classiques. Un registre quantique représente un ensemble de plusieurs qubits. Il est possible de construire un circuit quantique avec un seul registre quantique, mais il est parfois utile de séparer les qubits en plusieurs groupes pour les structurer. Un registre classique représente un ensemble de plusieurs bits classiques. Un tel registre sert d'abord à stocker les résultats des mesures des qubits.

Pour construire notre premier circuit quantique avec Qiskit, nous avons besoin d'importer la classe `QuantumCircuit` qui permettra de créer un nouveau circuit quantique. Nous avons également besoin des classes `QuantumRegister` et `ClassicalRegister` pour créer des registres quantiques et classiques. Une fois ces importations faites, les classes seront disponibles pour instancier différents objets.

```
from qiskit import QuantumCircuit, QuantumRegister, ClassicalRegister
```

La création des registres quantique et classique se fait en spécifiant le nombre de qubits (ou de bits) et une étiquette (par exemple `"q"`).

```
qreg = QuantumRegister(4, "q")
creg = ClassicalRegister(4, "c")
```

Un registre quantique est en fait une `list` de qubits. On accède au qubit `q` du registre `qreg` grâce à `qreg[q]`. On accède aux qubits d'un registre classique de la même manière. Les qubits et les bits des registres sont indexés à partir de `0`.

3.2 Circuit quantique

Présentons le code qui construit le circuit quantique et décrivons-le ensuite.

Code Python 1 – Circuit GHZ à quatre qubits

```
1    qreg = QuantumRegister(4, "q")
2    creg = ClassicalRegister(4, "c")
3
4    circuit = QuantumCircuit(qreg,creg)
```

```
5
6       circuit.h(qreg[0])
7       circuit.cx(qreg[0],qreg[1])
8       circuit.cx(qreg[0],qreg[2])
9       circuit.cx(qreg[0],qreg[3])
10
11      circuit.barrier()
12
13      circuit.measure(qreg[0],creg[0])
14      circuit.measure(qreg[1],creg[1])
15      circuit.measure(qreg[2],creg[2])
16      circuit.measure(qreg[3],creg[3])
```

D'abord, après avoir instancié un registre quantique et un registre classique, on instancie un circuit quantique en lui fournissant les deux registres qui le constituent. Un circuit quantique peut impliquer autant de registres que nécessaire.

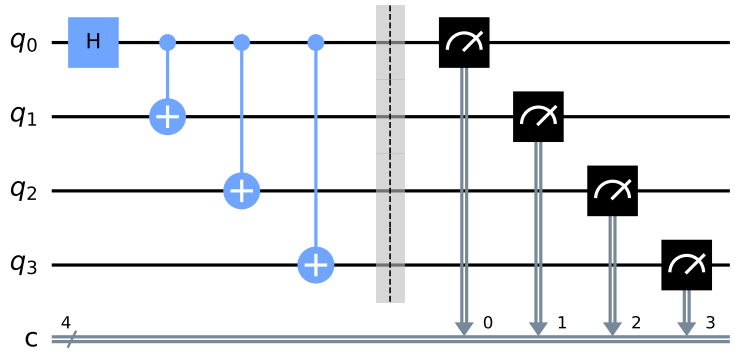
On ajoute ensuite des portes quantiques, une à une, à l'aide des différentes méthodes (`h()` pour Hadamard, `cx()` pour CNOT, etc.) en spécifiant sur quel(s) qubit(s) elles sont appliquées. Pour les portes à deux qubits, on spécifie d'abord le qubit de contrôle et ensuite le qubit cible.

On insère une *barrière* avant la mesure des qubits pour séparer ces deux parties du circuit. On ajoute une mesure au circuit quantique grâce à la méthode `measure()` en spécifiant le qubit mesuré et le bit où le résultat doit être écrit.

3.3 Visualisation

Une fois le circuit quantique construit, on peut le visualiser grâce à la méthode `draw()`. En spécifiant l'argument optionnel `output="mpl"`, une figure générée par la librairie `matplotlib` est produite.

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



Il est aussi possible de sauvegarder une figure dans différents formats.

```
fig = circuit.draw(output="mpl")
fig.savefig('circuit_ghz_0.pdf')
```

3.4 Mesure de tous les qubits

Il est très fréquent qu'on ait à mesurer tous les qubits. On peut utiliser la méthode `measure_all()` pour rapidement appliquer des mesures sur tous les qubits. Comme cette méthode inclut également la création automatique d'un registre classique et l'application d'une barrière, il est préférable de l'utiliser sur un circuit quantique qui n'en comporte pas.

Code Python 2 – Circuit GHZ à quatre qubits – Mesure de tous les qubits

```
qreg = QuantumRegister(4, "q")

circuit = QuantumCircuit(qreg)

circuit.h(qreg[0])
circuit.cx(qreg[0],qreg[1])
circuit.cx(qreg[0],qreg[2])
circuit.cx(qreg[0],qreg[3])

circuit.measure_all()
```

Sauf indication contraire, nous allons maintenant nous concentrer sur la partie unitaire des circuits quantiques. L'ajout des mesures pourra être fait une fois cette partie complétée.

3.5 Avec une boucle

On peut automatiser la création de certaines parties d'un circuit quantique. Dans le présent exemple, on appelle plusieurs fois la méthode `cx()` pour appliquer trois CNOT. En utilisant une boucle `for`, on peut construire le même circuit quantique plus efficacement.

Code Python 3 – Circuit GHZ à quatre qubits – Boucle

```
qreg = QuantumRegister(4, "q")

circuit = QuantumCircuit(qreg)

circuit.h(qreg[0])
for q in range(1,4):
    circuit.cx(qreg[0],qreg[q])
```

En utilisant le `range(1,4)`, l'index `q` débute à la valeur `1` et prend ensuite toutes les valeurs entières plus petites que `4`.

3.6 Dans une fonction

En plaçant la construction du circuit dans une fonction qui prend en entrée le nombre de qubits, il est alors possible de construire un circuit qui prépare un état GHZ pour un nombre arbitraire de qubits.

Code Python 4 – Circuit GHZ à n qubits – Fonction

```
def build_ghz_circuit(number_of_qubits):
    qreg = QuantumRegister(number_of_qubits, "q")

    circuit = QuantumCircuit(qreg)

    circuit.h(qreg[0])
    for q in range(1,number_of_qubits):
        circuit.cx(qreg[0],qreg[q])

    return circuit
```

La création d'un circuit GHZ à quatre qubits se fait alors en appelant la fonction `build_ghz_circuit()`.

```
circuit = build_ghz_circuit(4)
```

3.7 Version simplifiée

Pour des circuits quantiques simples, en particulier ceux avec un seul registre quantique, il est possible d'instancier directement un circuit quantique en spécifiant directement le nombre de qubits. Ensuite, l'application de portes quantiques se fait en spécifiant les index des qubits.

Code Python 5 – Circuit GHZ à quatre qubits – Version simplifiée

```
number_of_qubits = 4
circuit = QuantumCircuit(number_of_qubits)

circuit.h(0)
for q in range(1,number_of_qubits):
    circuit.cx(0,q)
```

L'écriture de circuit quantique sous cette forme est utile pour la préparation de circuits quantiques qui seront convertis à leur tour en portes quantiques à plusieurs qubits.

3.8 Définition et utilisation d'une porte quantique

Un circuit quantique qui ne comporte que des registres quantiques peut être converti en une porte quantique pour être incorporé dans un autre circuit plus tard. Cela permet entre autres de structurer un circuit quantique. On prépare ici une porte quantique qui applique le circuit GHZ à 4 qubits.

Code Python 6 – Définition d'une porte quantique

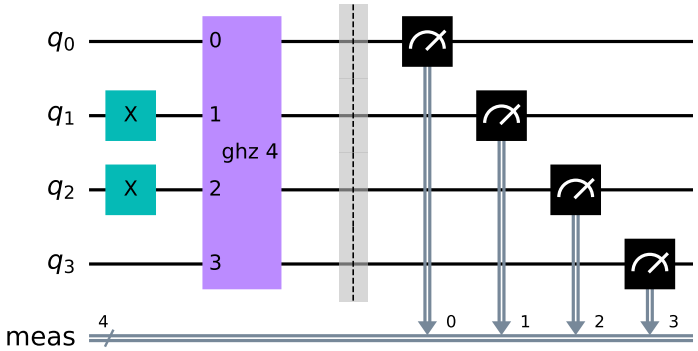
```
number_of_qubits = 4
ghz_gate_circuit = QuantumCircuit(number_of_qubits)
ghz_gate_circuit.h(0)
for q in range(1,number_of_qubits):
    ghz_gate_circuit.cx(0,q)

ghz_gate = ghz_gate_circuit.to_gate(label = "ghz 4")
```

On peut ensuite ajouter cette porte dans un circuit quantique à l'aide de la méthode `append` en fournissant la porte et les qubits sur lesquels l'appliquer.

Code Python 7 – Utilisation d'une porte quantique

```
circuit = QuantumCircuit(number_of_qubits)
circuit.x([1,2])
circuit.append(ghz_gate,[0,1,2,3])
circuit.measure_all()
circuit.draw('mpl')
```



3.9 Exécution

Finalement, voyons comment on peut exécuter un circuit quantique. Nous utiliserons ici un simulateur, mais il sera facile de passer à un prototype d'ordinateur quantique.

On doit d'abord importer deux éléments supplémentaires. D'abord, `Aer` est le sous-module qui contient tout ce qui permet de simuler l'exécution d'un circuit, en particulier les simulateurs. Ensuite, la fonction `execute` qu'on utilisera pour lancer l'exécution.

```
from qiskit import Aer, execute
```

Dans Qiskit, un circuit quantique s'exécute sur un *backend*. Un *backend* peut être un simulateur ou un prototype d'ordinateur quantique. Comme les deux utilisent la même interface, on peut facilement passer d'un à l'autre. Pour l'instant on veut utiliser le simulateur QASM. On utilise la commande suivante pour y avoir accès.

```
qasm_simulator = Aer.get_backend('qasm_simulator')
```

Pour exécuter un circuit, il ne nous reste qu'à utiliser la fonction `execute`. Cette fonction peut prendre un très grand nombre d'arguments optionnels. Pour l'instant, on se limitera au circuit à exécuter, au *backend* et au nombre de fois (*shots*) que le circuit devra être exécuté.

```
job = execute(circuit, qasm_simulator, shots = 1000)
```

3.10 Extraction des résultats

La fonction `execute` ne retourne pas directement les résultats, mais une structure qui contient l'information sur le circuit exécuté, l'exécution ainsi que les résultats. On accède aux résultats à l'aide de la méthode `result()`. Pour l'instant, on s'intéresse uniquement au nombre de fois que chaque état de base a été obtenu, on appelle donc immédiatement la méthode `get_counts()`.

```
counts = job.result().get_counts()
print(counts)
```

```
{'0110': 522, '1001': 478}
```

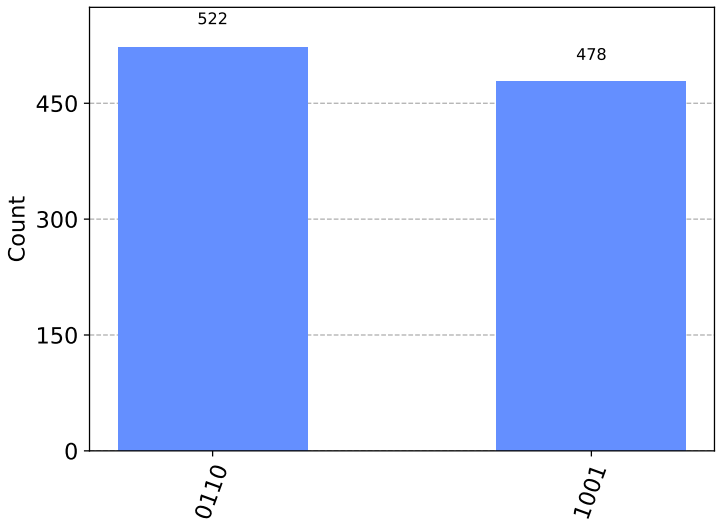
On obtient alors un `dict` qui contient le nombre de fois que chaque état de base à été obtenu. La somme de ces valeurs est égale au nombre total de fois que le circuit a été évalué.

3.11 Visualisation des résultats

Il est utile de pouvoir visualiser les résultats d'un calcul quantique. Qiskit offre plusieurs fonctions très utile dans son sous module `visualization`. Pour visualiser les `counts` on peut utiliser `plot_histogram`.

```
from qiskit.visualization import plot_histogram
```

```
plot_histogram(counts)
```


522
478
450
300
150
0
Count
0110
1001