BSQ 110 - Introduction aux sciences quantiques et à leurs applications

Rapport d'évaluation - Deuxième projet de programmation

Construction et automatisation de circuits quantiques

Guillaume Blouin

1 Rétroaction

1.1 Fonctionnalités

1.1.1 Construction des oracles

La construction des oracles pour l'algorithme de Deustch avec `choisirOracleDeDeustch()` est clair, simple et efficace. Le choix dépend d'un entier entre 0 et 3 comme demandé.

Pour l'algorithme de Deustch-Jozsa, la fonction `choisirOracleDeustchJozsa` m'apparait inutilement compliquée. Je n'ai pas compris pourquoi l'utilisation de la classe `VariablesStructure` est essentielle. Elle semble surtout servir à suivre le nombre de qubits d'un circuit. Il aurait été plus simple d'utiliser l'attribut `num_qubits` de `QuantumCircuit`.

La construction de l'oracle pour l'algorithme de Barnstein-Vazierani avec `choisirOracleBernstein_Vazirani` fonctionne correctement à l'exception que l'entier `s` devait être fourni en entrée.

1.1.2 Construction des circuits d'algorithmes

La fonction `DeustchAlgo` prend en entrée un oracle en construit le circuit de l'algorithme de Deustch comme demandé.

La fonction `deustchJozsaAlgo` construit le circuit pour l'algorithme de Deustch-Jozsa à l'aide d'un oracle sous la forme d'une porte quantique. L'argument `number_of_qubits` n'est pas vraiment nécessaire, car cette information est présente dans la porte de l'oracle. Pourquoi est-ce que le résultat est retourné sous la forme d'une porte quantique plutôt que d'un circuit ? Cela rend l'implémentation inutilement confuse. C'est une bonne idée d'utiliser cette même fonction construire l'algorithme de Berstein-Vazirani.

1.1.3 Exécution des circuits et analyse des résultats

Les mesures sont ajoutées avant l'exécution des circuits et non à la construction des circuits des algorithmes, ce qui m'apparait être une bonne idée. Les exécutions de circuits sont toutes effectuées correctement et les résultats sont affichés pour vérification. Par contre, aucune analyse des résultats n'est effectuée. Une vérification automatique des résultats aurait permis de confirmer le fonctionnement des algorithmes.

1.2 Qualité du code

1.2.1 Lisibilité

Les noms de variables et des fonctions sont informatifs. Cependant, plusieurs conventions différentes sont utilisées sans distinction, ce qui rend la lecture du code plus difficile. Comme pour le premier projet, le code m'apparait inutilement compliqué étant donnée la simplicité des tâches à accomplir. En particulier, la création de l'oracle pour l'algorithme de Deustch-Jozsa aurait pu être simplifiée et réécrite.

1.2.2 Encapsulation et réutilisabilité

Le fait que le code est compliqué limite sa réutilisabilité.

Une des tâches qui semble avoir été difficile à remplir est d'identifier sur quels qubits appliquer chaque opération. Je crois comprendre que la classe `VariablesStructure` est introduite pour répondre à cette problématique. Cependant, tout cela aurait pu être facilement évité en utilisant l'attribut `num_qubits` de `QuantumCircuit`.

La création d'une fonction qui ajoute des mesures à tous les qubits, sauf le dernier, aurait pu grandement simplifier le code à l'exécution.

1.2.3 Originalité

L'utilisation de décorateurs pour valider les entrées de certaines fonctions est judicieuse.

1.3 Commentaires généraux

Ton implémentation pour ce projet fonctionne, mais est, à mon avis, inutilement compliquée. La tâche à effectuer est relativement simple, et c'est possible de le faire avec un code plus court et simple. La situation risque de devenir plutôt confuse lorsque tu auras à coder des algorithmes plus complexes. Comme tu as de bonnes compétences en programmation, je t'encourage à tenter de simplifier tes implémentations et minimisant le nombre d'arguments à tes fonctions par exemple. Cela rendra ton code plus facile à lire et à améliorer par la suite.

En règle générale il est préférable d'utiliser une boucle `for` (et non une boucle `while`) lorsque le nombre d'itérations est connu à l'avance. En Python cela à l'avantage d'être aussi beaucoup plus facile à lire.

2 Évaluation

L'évaluation est en trois parties. D'abord, les fonctionnalités sont évaluées. L'unique critère est que votre implémentation produisent les résultats attendus et que chaque fonction se comporte correctement dans toutes les situations, même des situations que vous n'avez pas nécessairement rencontrées durant le projet.

Ensuite, la qualité de votre code est évaluée en fonction de sa lisibilité et au niveau de sa structure. Un code lisible est un code qui est d'abord bien présenté avec des noms de variable et de fonctions qui sont descriptifs et informatifs. L'utilisation assidue de convention de notation aide également à la lisibilité. Un code bien structuré permet une utilisation facile et flexible de ses fonctionnalités. Un code bien structuré est également plus lisible.

Finalement, l'originalité de vos implémentations peut vous permettre d'obtenir des points bonus. Les implémentations qui sont différentes de l'approche habituelle, particulièrement efficace ou avec une application beaucoup plus générale que nécessaire peuvent être considérées comme originales. L'originalité ne doit cependant pas se faire au détriment de l'efficacité. Il se peut qu'une implémentation originale nuise à la lisibilité. Cela n'est justifiable que s'il y a un gain considérable en efficacité et devrait alors être accompagné d'explications sous la forme de commentaires et d'un document explicatif.

Fonctionnalités			Commentaires
Deustch	Oracle	6/6	
	Circuit	2/2	
	Exécution et analyse	1/2	Voir 1.1.3
Deustch-Jozsa	Oracle	5/6	Voir 1.1.1
	Circuit	1/2	Voir 1.1.2
	Exécution et analyse	1/2	Voir 1.1.3
Berstein-Vazirani	Oracle	5/6	Voir 1.1.1
	Circuit	2/2	
	Exécution et analyse	1/2	Voir 1.1.3
Total		24/30	

Qualité du code		Commentaires
Lisibilité	2/4	Voir 1.2.1
Structure	4/6	Voir 1.2.2
Total	6/10	

Bonus		Commentaires
Orignalité	1/(2)	
Total	1/(2)	

Note finale : 31/40