Projet 1 - BSQ101

Algorithme de Grover

Christopher Sicotte

Présenté à
Maxime Dion et Karl Thibault

Département des Sciences
Université de Sherbrooke
30 janvier 2024

1 Introduction

L'informatique quantique est à la base de beaucoup de projets de recherche dût à son potentiel de calcul théoriquement exponentiel sur celui de l'informatique classique. En exploitant les principes de la mécanique quantique, on peut utiliser la superposition qui permet d'avoir un bit d'information en 2 états simultanément comparé à un bit classique. Donc, pour n qubits, nous avons 2^n bits. Par contre, même avec cet avantage, ce n'est pas garantie d'avoir de meilleures performances sur un ordinateur quantique. Effectivement, on doit avoir des algorithmes quantiques qui puissent exploiter cet avantage. Un des algorithmes les plus célèbres qui a un avantage quadratique sur son concurrent classique est l'algorithme de recherche de Grover.

L'algorithme de Grover permet une recherche dans une liste non structurée en temps $O(\sqrt{n})$, alors que son concurrent classique peut y arriver en temps $O(n)$, n étant la taille de la liste. L'algorithme utilise la superposition d'états et l'intrication pour exécuter en parallèle toutes les solutions possibles. Ensuite, il effectue un oracle qui inverse la phase des bonnes solutions dans le contexte. Finalement, on utilise un diffuseur qui amplifie la probabilité d'obtenir la bonne solution. En répétant ces étapes un nombre spécifique de fois, on obtient une meilleure probabilité d'obtenir les bonnes solutions en mesurant le système.

En utilisant un peu d'ingéniosité, cet algorithme peut même résoudre des problèmes de satisfiabilité. En ayant un qubit pour chaque variable de notre proposition logique et un qubit supplémentaire par clause, on peut trouver sa solution. C'est le sujet de ce projet.

2 Théorie

2.1 Structure

Pour résoudre un problème de proposition logique, on doit transformer notre proposition en sa forme normale conjonctive pour la séparer en conjonctions. Dans notre circuit quantique, On doit avoir un qubit associé à chaque variable de notre proposition. On doit aussi avoir un qubit associé à chaque conjonction de notre CNF. C'est en effectuant plusieurs itérations du circuit de Grover qu'on obtient une amplification des probabilités d'obtenir au final les bonnes solutions. Le nombre d'itérations optimal est donné par

$$nb\_iter = 4\pi * \sqrt{N/M}$$

où M est le nombre de bonnes solutions et N est le nombre total de solutions.

2.2 Superposition et intrication

L'algorithme de Grover exploite le phénomène de superposition pour explorer plusieurs solutions en parallèle. La stratégie est d'utiliser l'intrication entre nos qubits des variables à leur qubit de clause pour isoler les états solutions en inversant la phase de l'état.

2.3 Oracle

L'oracle est une boîte noire qui crée l'intrication entre les qubits des variables et les qubits ancillaires. Avec des portes Toffoli, on veut simuler un "OR" logique. Effectivement, chaque clause logique sera définie par une porte multicontrôle-X pour changer l'état du qubit ancillaire associé. On ajoute ensuite une porte X pour encore changer l'état de ce même qubit. C'est ce processus qui crée l'intrication entre les contrôles et la cible. Ensuite, on ajoute au système une porte multicontrôle-Z sur les qubits ancillaire, ce qui aura pour but d'inverser la phase de toutes les solutions qui satisfont les clauses. Il ne reste plus qu'à rajouter au circuit les mêmes portes Toffoli de la première partie pour enlever l'intrication et restaurer l'état des qubits ancillaires sans toutefois changer l'état des qubits de variables.

2.4 Diffuseur

Le diffuseur sert à amplifier la probabilité d'obtenir les bonnes solutions en effectuant une réflexion par rapport à l'état de superposition uniforme.

3 Implémentation

Le code se divise en 4 fichiers contenant des fonctions utiles à l'algorithme et un fichier principal (main.py) qui sert d'interface utilisateur. À noter que lors de l'explication, j'omettrai les paramètres des fonctions par souci de simplicité et puisqu'ils sont bien écrits dans le code.

3.1 main.py

C'est ce fichier qui sera lancé par l'utilisateur pour lancer l'algorithme.

3.2 IBMQ_credentials.py

Ce fichier contient toutes les fonctions utiles pour utiliser facilement les services d'IBMQ.

ibmq_connexion() : sauvegarde un compte IBM Quantum et le définit comme compte par défaut. À utiliser uniquement la première connexion.

ibmq_provider() : donne accès au "backend" et au "provider" d'IBMQ.

3.3 BooleanProblems.py

Ce fichier contient toutes les propositions logiques du projet.

create_cake_problem() : crée une forme normale conjonctive à partir d'une proposition logique pour le problème du gâteau.

create_pincus_problem() : crée une forme normale conjonctive à partir d'une proposition logique pour le problème de la planète Pincus.

3.4 QuantumUtils.py

Ce fichier contient toutes les fonctions qui aident à l'exécution de l'algorithme sans toutefois la construire.

quantum_results_to_boolean() : convertit les résultats de l'algorithme quantique dans un format facile à comprendre.

calculate_threshold() : calcule le seuil pour séparer les clusters True et False dans les données avec un algorithme de K-means.

save_histogram_png() : enregistre un histogramme de vos résultats sous forme d'un fichier PNG.

validate_grover_solutions() : valide les résultats de la simulation avec SymPy.

3.5 GroverUtils.py

Ce fichier contient le coeur de l'Algorithme de Grover. Toutes les fonctions qui servent à créer l'algorithme et l'exécuter se trouve ici.

disjunction_gate() : crée la porte de Toffoli en fonction d'une clause de disjonction.

create_oracle_gates() : crée les parties des oracles en fonction des portes de Toffoli construites à partir de porte de disjonction de la forme de portes de Toffoli. Appel disjonction_gate().

cnf_to_oracle() : traduit une formule logique normale conjonctive en un oracle qui prend la forme d'une porte quantique. Donc, applique une porte de disjonction suivie d'une porte multicontrôle-Z et finalement applique l'inverse de la porte de disjonction précédemment appliquée. Appel create_oracle_gates().

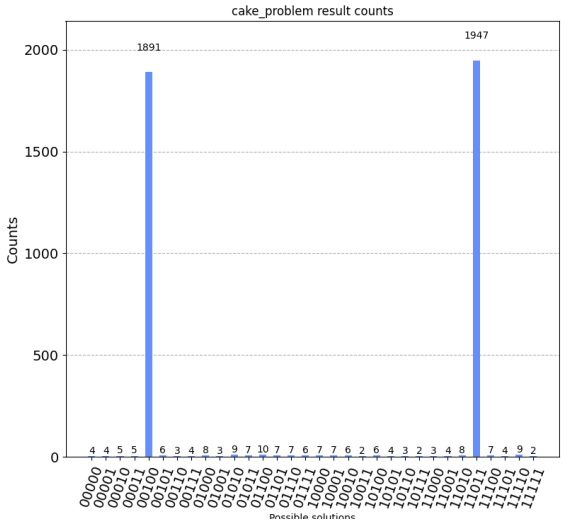
build_diffuser() : construit le diffuseur en fonction du nombre de qubits en entrée.

build_grover_circuit() : construit l'algorithme de Grover à partir des entrées. Donc, applique les oracle et le diffuseur au circuit et itère selon un nombre d'itérations. Appel cnf_to_oracle() et build_diffuser()

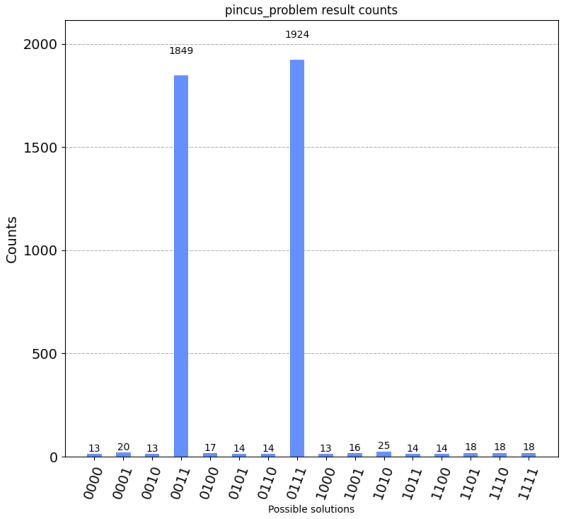
solve_sat_with_grover() : Étant donné une formule logique, convertit cette formule en un oracle et exécute l'algorithme de Grover sur un backend de IBMQ. Appel build_grover_circuit(), quantum_results_to_boolean() et save_histogram_png().

4 Résultats et Analyse

La figure 1 est un exemple de résultats obtenus avec l'algorithme de Grover. Après avoir répété l'expérience sur un simulateur quantique 1000 fois (par défaut), on voit que pour les 2 propositions, nous avons une forte probabilité d'obtenir 2 solutions. À première vue, il est difficile de voir quel est le bon résultat. C'est pourquoi le code renvoie aussi une liste de dictionnaire pour chaque résultat. Pour le problème du gâteau, nous obtenons 2 résultats. Soit Chris a mangé le gâteau ou bien tout le monde sauf Chris ont mangé le gâteau. Pour le problème de Pincus, on obtient aussi 2 solutions. Soit un Pincusien est bruyant et joyeux, soit un Pincusien est Bruyant, joyeux et malade.



(a) Histogramme des résultats pour le problème du gâteau



(b) Histogramme des résultats pour le problème de la planète Pincus

Figure 1: Les 2 histogrammes produits par l'algorithme de Grover.

Si on analyse ces résultats, on constate que l'algorithme de Grover nous a permis de trouver les bonnes solutions en utilisant $\sqrt{N} + c$ qubits, où c représente le nombre de conjonctions. Comme mentionné dans la structure de l'agorithme de Grover, on sait que pour augmenter optimalement les probabilités de mesurer un état bonne solution, on doit effectuer plusieurs itérations de Grover qui est égale à $4\pi * \sqrt{M/N}$. Si nous voulions effectuer une même résolution sur un ordinateur classique, il n'existe pas d'algorithme particulièrement plus efficace que d'exécuter une vérification naïve sur toutes les possibilités et vérifier sa véracité. Ce qui mènerait à un temps $O(N)$. On peut voir un net avantage quantique pour cet algorithme.