

RÉSOLUTION QUANTIQUE DU PROBLÈME D'ISOMORPHISME DE GRAPHES

Rapport écrit final

Septembre 2024 - Mai 2025

Noé Sol, Philibert Pappens, Steve Kenne-Wamba, Gabriel Cheval, Albert Thaury

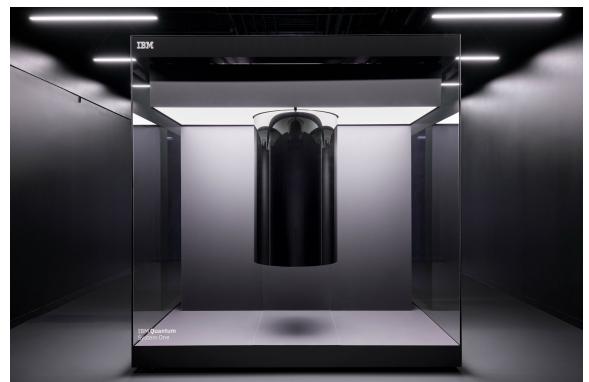
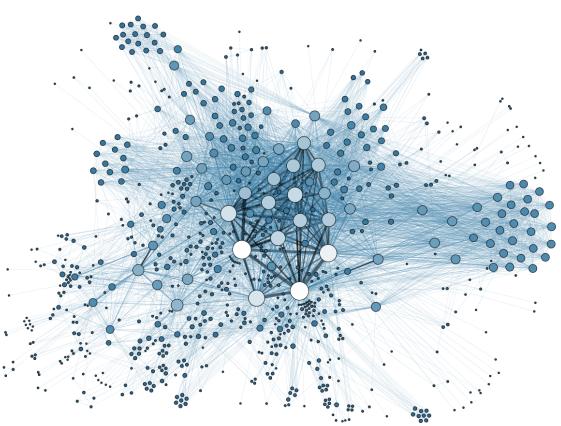


TABLE DES MATIÈRES

Introduction	3
Motivations	3
Contexte : le problème d'isomorphisme de graphes	3
Informatique quantique	5
1 Résolution théorique	7
1.1 Transformation en un problème d'état fondamental d'Hamiltonien	7
1.2 Algorithme adiabatique quantique	8
1.3 Trotterisation	9
1.4 Implémentation complète pour $N = 2$ et résultats numériques	10
2 Problèmes rencontrés en pratique	15
2.1 Temps de cohérence	15
2.2 Choix du pas de trotterisation	16
2.3 Longueur du Hamiltonien	21
2.4 Idle time	22
2.5 Mitigation d'erreur	22
3 Conduite du projet	28
3.1 Organisation temporelle du projet	28
3.2 Répartition du travail et relation avec le tuteur	28
3.3 Compétences acquises	29
Conclusion et remerciements	31
Annexe	32

INTRODUCTION

MOTIVATIONS

A travers ce PSC, nous avons souhaité découvrir les rudiments de l'algorithme quantique. Pour ce faire, nous avons sélectionné un problème complexe en informatique classique, un problème d'optimisation combinatoire appelé isomorphisme de graphe, et nous avons étudié la possibilité de sa résolution par ordinateur quantique. Nous avons pu tester cette résolution sur de vraies machines quantiques, mises à disposition gratuitement en ligne par IBM. L'informatique quantique étant un sujet pointu, nous nous efforcerons d'être le plus pédagogique possible dans ce rapport, en insistant uniquement sur les éléments nécessaires à la compréhension.

CONTEXTE : LE PROBLÈME D'ISOMORPHISME DE GRAPHS

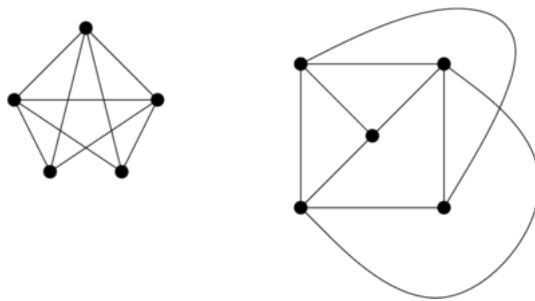


FIGURE 1 – Exemple de graphes isomorphes

Tout d'abord, nous présentons dans cette section le problème que nous nous attacherons à résoudre dans toute la suite du document. Il s'agit d'un problème d'optimisation combinatoire, dont l'énoncé est le suivant :

Étant donnés deux graphes G et G' non orientés, déterminer s'ils sont isomorphes, c'est-à-dire s'ils sont identiques à permutation (réindexation) des sommets près.

Voici un exemple pour mieux comprendre : ci-dessous, deux graphes à 4 sommets dont on étudie l'isomorphisme. Leurs sommets sont numérotés de 0 à 3.

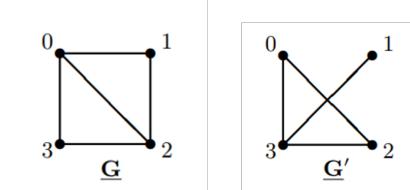


FIGURE 2 – Graphes à 4 sommets

Étudions l'effet de la permutation suivante sur G .

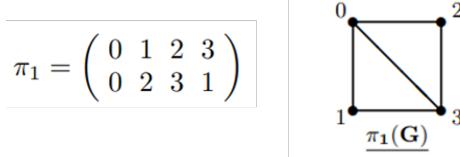


FIGURE 3 – Permutation et action sur G

Il s'agit de la manière conventionnelle pour noter une permutation : la première ligne montre les antécédents, et la seconde les images respectives. Ici par exemple, $\pi_1(1) = 2$ et $\pi_1(2) = 3$. Cela signifie que, par cette permutation, le sommet 1 est envoyé sur 2. Dans la suite, nous négligerons souvent l'écriture de la première ligne, en ce qu'elle sera toujours identique.

L'action d'une permutation peut simplement être vue comme un renommage des sommets du graphe. Si l'on compare désormais $\pi_1(G)$ à G' , on se rend compte qu'ils sont identiques : en effet, le sommet 0 est relié à 2 et à 3, le sommet 1 à 3, etc. On conclut ainsi que G et G' sont isomorphes, et que l'un des isomorphismes en question est π_1 .

En général, la difficulté est donc de trouver la bonne permutation.

Déterminer si deux graphes sont isomorphes est un problème algorithmique qui est étudié depuis le début de l'informatique. Ce problème d'isomorphisme de graphe (IG) a pris de l'ampleur dans la communauté théorique dans les années 1970 lorsqu'il est apparu comme l'un des rares problèmes naturels de la classe de complexité NP qui ne pouvait être ni classé comme difficile, c'est-à-dire NP-complet, ni démontré comme résoluble avec un algorithme efficace, c'est-à-dire un algorithme en temps polynomial.

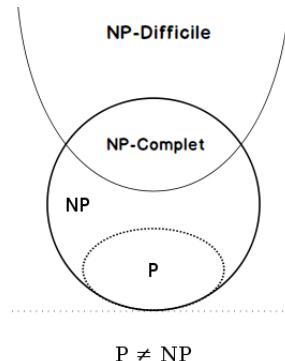


FIGURE 4 – Différence entre un problème NP-Complet et NP-Difficile

De nombreux ouvrages, à l'instar de *Reducibility among Combinatorial Problems* de Richard M. Karp [5] et *Computers and Intractability : A Guide to the Theory of NP-Completeness* de Garey et Johnson [2], le mentionnent comme un problème ouvert et, depuis lors, il reste un problème ouvert majeur de l'informatique théorique. Il convient tout de même de remarquer que les heuristiques (algorithmes d'approximation rapides mais ne fournissant pas toujours la bonne solution) sont très efficaces pour ce genre de problème.

Enfin, une façon pratique de représenter les graphes pour les calculs, est d'utiliser leur matrice d'adjacence (notées A et A'). L'intérêt est que deux graphes sont isomorphes si et seulement si leurs matrices d'adjacences sont semblables par une matrice de permutation (s'il existe une matrice de permutation P telle que $A' = PAP^T$). La matrice de permutation P représente alors ladite permutation des sommets. Il n'est pas important de réellement comprendre cela, mais simplement de savoir que nous agirons plutôt sur les matrices d'adjacence que sur les graphes eux-mêmes.

INFORMATIQUE QUANTIQUE

La notion d'ordinateur quantique fut introduite pour la première fois en 1982 par Richard Feynman, qui cherchait à résoudre des problèmes nécessitant des calculs trop volumineux pour un ordinateur classique. Un ordinateur quantique utilise des qubits, qui sont des unités de base de l'information quantique. Contrairement aux bits classiques, limités à deux états distincts, 0 ou 1, un qubit peut exister dans une superposition d'états, c'est-à-dire être simultanément dans une combinaison des états 0 et 1 avec des probabilités différentes : si on le mesure, on a une certaine probabilité de le mesurer dans l'état 0, et une autre de le trouver dans l'état 1. Tant qu'il n'est pas mesuré, le qubit n'est entièrement dans aucun des deux états. De plus deux qubits peuvent être **intriqués** (c'est-à-dire corrélés de telle sorte que l'état de l'un influence instantanément celui de l'autre). Enfin, les qubits peuvent également **interférer**.

De plus, l'informatique quantique est probabiliste : lorsque l'on mesure les qubits à la fin du programme, celle-ci n'est pas déterministe. C'est pourquoi on réalisera de nombreuses exécutions pour déterminer l'état final du circuit.

En pratique, un qubit représente un objet physique placé dans un état quantique : il peut s'agir d'ions piégés par laser, ou de supraconducteurs contrôlés par impulsions micro-ondes,...

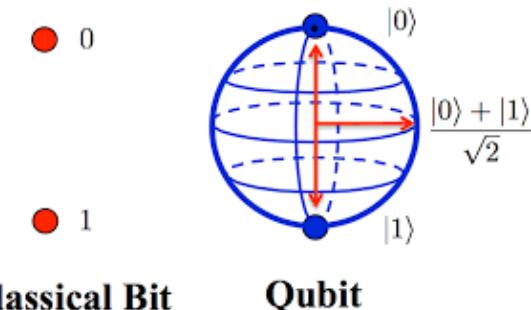


FIGURE 5 – Représentation d'un qubit

Un objet quantique est modélisé par sa **fonction d'onde**, une fonction complexe qui remplace la position de notre objet (celle-ci étant inconnue). Le module au carré de cette fonction désigne la probabilité de présence de notre objet quantique à une position et à un instant donné.

Ainsi, on peut voir la différence entre informatique classique et quantique via la métaphore du labyrinthe [4]. Dans un labyrinthe un algorithme classique se comportera comme un humain pour sortir, c'est à dire qu'il testera tous les chemins jusqu'à trouver la sortie. Un algorithme quantique lui fonctionnera plutôt comme un radar : il émettra une onde qui va simultanément parcourir tous les chemins, se réfléchir et interférer lorsque le chemin est bloqué et au contraire s'échapper si elle trouve la sortie.

Tout comme en informatique classique, les programmes quantiques se résument à une succession de portes quantiques traversées par un certain nombre de qubits. Le passage d'un qubit par une porte revient à appliquer un opérateur à ce qubit (et donc à modifier son état, c'est à dire les probabilités de mesure associées à chaque état). L'état et les interactions d'un système quantique (de plusieurs qubits) sont encodés par un opérateur que l'on appelle Hamiltonien.

Ainsi, si on veut simuler l'évolution d'un système quantique de Hamiltonien H pendant une durée ΔT , l'équation de Schrödinger nous enseigne qu'il faut appliquer au système un opérateur, et donc une porte, $e^{iH\Delta T}$ (ce qui revient à multiplier la fonction d'onde de notre système par ce même facteur). Dans les faits, un Hamiltonien peut être décrit par ses états propres qui sont d'une certaine façon les états physiquement acceptables de notre système. En mécanique classique cela correspond par exemple aux différents modes propres d'un système, typiquement une corde vibrante. Chacun de ces états propres est associé à une valeur propre : le niveau

d'énergie du système dans cet état.

En algorithmie classique, il existe de nombreuses façons de résoudre un problème. Par exemple, par force brute en testant toutes les possibilités, par diviser et conquérir en décomposant le problème en sous-problèmes... En informatique quantique, c'est pareil. L'un des grands domaines de l'informatique quantique est le **calcul quantique adiabatique**. Le principe de cette branche de l'informatique quantique est de construire un Hamiltonien tel que l'état fondamental de cet Hamiltonien soit une solution de notre problème. Une fois cela fait, l'**algorithme adiabatique quantique** (QAA), que nous décrivons en plus amples détails en partie 1.2, décrit une méthode pour trouver cet état fondamental en partant d'un Hamiltonien simple dont nous connaissons les états fondamentaux. Ainsi, dans notre approche, nous commençons par reformuler notre problème mathématique en un problème de recherche d'état fondamental, puis nous présentons et appliquons l'un des algorithmes permettant de résoudre ce type de problème, le QAA.

1

RÉSOLUTION THÉORIQUE

1.1 TRANSFORMATION EN UN PROBLÈME D'ÉTAT FONDAMENTAL D'HAMILTONIEN

Rappelons tout d'abord que nous souhaitons résoudre notre problème via la branche de l'informatique quantique basée sur le fait de trouver l'état fondamental d'un Hamiltonien. Il s'agit alors de transformer notre problème, pour l'instant mathématique, en un **problème de recherche d'état fondamental**.

Pour cela, on va suivre l'approche de Frank Gaitan et Lane Clark [1], et introduire une **fonction de coût**. L'idée est de reformuler le problème d'isomorphisme de graphes en un problème de minimisation d'une certaine fonction (appelée donc fonction de coût, notée C) : on veut trouver $C(G, G', P)$ telle que $C(G, G', P) = 0$ si et seulement si P est une permutation réalisant l'isomorphisme de G sur G' (et C toujours positive). En effet, une fois une telle fonction de coût déterminée, il s'agira alors de définir un Hamiltonien qui soit tel que son énergie fondamentale soit le minimum de la fonction de coût C . Si l'énergie fondamentale trouvée vaut 0, les graphes sont isomorphes. Sinon, ils ne le sont pas.

A priori, notre fonction C est définie sur l'espace des permutations de $\{1, \dots, n\}$ (pour des graphes à n sommets). Afin de faciliter les calculs, nous identifierons une permutation s des sommets par ses images : les $s(i)$, où i est un sommet du graphe G , puis nous décomposerons les $s(i)$ en binaire.

Cela permet de travailler dans l'espace des séquences binaires d'une certaine longueur plutôt que l'espace des permutations, ce qui fait plus sens en informatique. Il ne s'agit pas là de détailler ce processus, mais simplement d'avoir à l'esprit qu'il y a correspondance directe entre fonctions de $\{1, \dots, n\}$, et **séquences binaires** (de longueur $n \times \log(n)$) (cf Annexe 3.3.3).

Nous sommes désormais armés pour créer notre fonction de coût, qui est celle de Frank Gaitan et Lane Clark [1]. Celle-ci sera scindée en deux parties : une première contribution, donnée par C_1 et C_2 , telle que $C_1(s) = C_2(s) = 0$ si s est bien une permutation de $\{1, \dots, n\}$. En effet, il est compliqué de tester uniquement les permutations via notre reformulation binaire, donc nous testons toutes les fonctions de $\{1, \dots, n\}$, même si elles ne sont pas bijectives. Cela peut être une piste d'amélioration pour la suite en ce que C_1 et C_2 ajoutent beaucoup de termes à C , et cela uniquement pour vérifier que s est bien une permutation.

C'est C_3 , deuxième partie de C , qui incarne à proprement parler la condition d'isomorphisme. On pose $C_3(s) = \|\sigma(s)A\sigma^T(s) - A'\|_2$, où A est la matrice d'adjacence de G , A' celle de G' , et $\sigma(s)$ la matrice de permutation associée à s . **C_3 est nulle si $\sigma(s)A\sigma^T(s) = A'$** , c'est-à-dire si **les deux graphes sont isomorphes**.

Voici les trois contributions explicitées, mais leur connaissance précise n'est pas du tout nécessaire à la compréhension.

$$C(s) = C_1(s) + C_2(s) + C_3(s), \quad (1)$$

avec :

$$C_1(s) = \sum_{i=0}^{N-1} \sum_{\alpha=N}^M \delta_{s_i, \alpha}, \quad (2)$$

$$C_2(s) = \sum_{i=0}^{N-2} \sum_{j=i+1}^{N-1} \delta_{s_i, s_j}, \quad (3)$$

$$C_3(s) = \|\sigma(s)A\sigma^T(s) - A'\|_2, \quad (4)$$

où $\delta_{x,y}$ est le delta de Kronecker, N est le nombre de sommets, M est la plus petite puissance de 2 supérieure à N (en passant par le binaire, on ne peut agir que dans des espaces où N est une puissance de 2, ce qui introduit un défaut d'injectivité si N n'en est pas une), et $s_i = s(i)$.

Nous avons donc créé une fonction C qui est nulle uniquement si son argument est une permutation envoyant le graphe G sur le graphe G' . Comme évoqué brièvement précédemment, nous pouvons modéliser une permutation s par une chaîne binaire de longueur $n \log_2(n)$, où n désigne le nombre de sommets de notre graphe. Finalement, chaque bit de cette chaîne binaire est représenté informatiquement parlant par un qubit.

Nous avons donc construit une fonction C qui prend en entrée une chaîne de $n \log_2(n)$ qubit, renvoie une valeur positive et nulle si nos graphes sont isomorphes.

Pour transformer cette fonction de coût en Hamiltonien, on exploite avant tout la transformation du symbole de Kronecker $\delta(i, j)$ en portes quantiques via une combinaison bien choisie de portes de Pauli Z appliquées aux qubits i et j. La représentation du symbole de Kronecker en portes quantiques est détaillée dans l'Annexe 3.3.3. Est également explicitée en Annexe 3.3.3 la transformation de $C_3(s)$ en sommes et produits de delta de Kronecker. Finalement, on parvient à transformer notre fonction de coût C en un Hamiltonien dont on souhaite trouver l'état fondamental (Annexe 3.3.3).

1.2 ALGORITHME ADIABATIQUE QUANTIQUE

L'un des concepts fondamentaux de la mécanique quantique, au centre du calcul quantique adiabatique, est le **théorème adiabatique**. Ce théorème stipule que si le Hamiltonien $H(\lambda(t))$ d'un système quantique varie de manière suffisamment lente et continue, alors, si le système commence dans son état fondamental $|GS\rangle$ à $t = 0$, il restera dans l'état fondamental de $H(\lambda(t))$ à tout instant t de l'évolution. On peut voir ça comme si l'on plaçait une boule au fond d'une cuvette dans sa position d'équilibre d'énergie minimale (le fond de la cuvette). Puis, on 'tire' les bords de la cuvette de façon à l'aplanir (et donc réduire sa profondeur). Si l'on fait cela suffisamment lentement, notre boule reste dans l'état d'équilibre à chaque étape. Mais, faire cela brusquement risque de la faire décoller et donc de lui faire quitter son état fondamental.

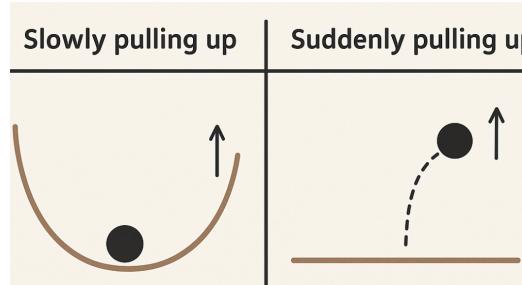


FIGURE 6 – Illustration de l'exemple de la cuvette

En pratique, cela signifie que, pour résoudre un problème de calcul, on prépare d'abord le système dans l'état fondamental du Hamiltonien initial H_0 , puis on fait évoluer lentement le Hamiltonien vers une forme finale H_1 qui contient la solution du problème. Si l'évolution est suffisamment lente, **le système quantique atteindra l'état fondamental de H_1** , fournissant ainsi la solution désirée.

L'évolution du Hamiltonien $H(\lambda)$ est généralement une combinaison linéaire des Hamiltoniens H_0 et H_1 :

$$H(\lambda) = (1 - \lambda)H_0 + \lambda H_1 \quad (5)$$

Par ailleurs, nous choisissons le Hamiltonien initial de sorte à connaître facilement son état fondamental. Il est d'usage de choisir $H_0 = -\sum_i X_i$ où X est la matrice de Pauli-X, d'états propres $|-\rangle$ et $|+\rangle$, et i parcourt l'ensemble des qubits. L'état fondamental de H_0 est donc la superposition de tous les qubits dans l'état $|+\rangle$, ce qui se prépare très facilement en appliquant initialement une porte de Hadamard à chacun des qubits.

1.3 TROTTERISATION

Cet algorithme repose sur la discrétisation du temps, un procédé appelé *trotterisation*, pour simuler des Hamiltoniens qui varient dans le temps. Pour rappel, appliquer un Hamiltonien indépendant du temps à un système quantique pendant un temps t revient à multiplier sa fonction d'onde par l'opérateur e^{-iHt} . Mais, dans le cadre de l'algorithme adiabatique, notre Hamiltonien évolue dans le temps, rendant les calculs nettement plus compliqués. Le principe de la trotterisation est de **diviser le temps d'évolution total ΔT en N intervalles**, où chaque intervalle est suffisamment petit pour que le Hamiltonien change peu et soit donc approximé comme constant. On définit un petit intervalle de temps $dt = \frac{\Delta T}{N}$. En vertu de (5), l'opération d'évolution prend donc la forme

$$U = \lim_{N \rightarrow +\infty} (e^{-iH_1 dt} \cdot e^{-iH(\lambda_1) dt} \cdot \dots \cdot e^{-iH_0 dt} + \mathcal{O}(dt^2))$$

qu'on réduira pour N suffisamment grand à

$$U = e^{-iH_1 dt} \cdot e^{-iH(\lambda_1) dt} \cdot \dots \cdot e^{-iH_0 dt} \quad (6)$$

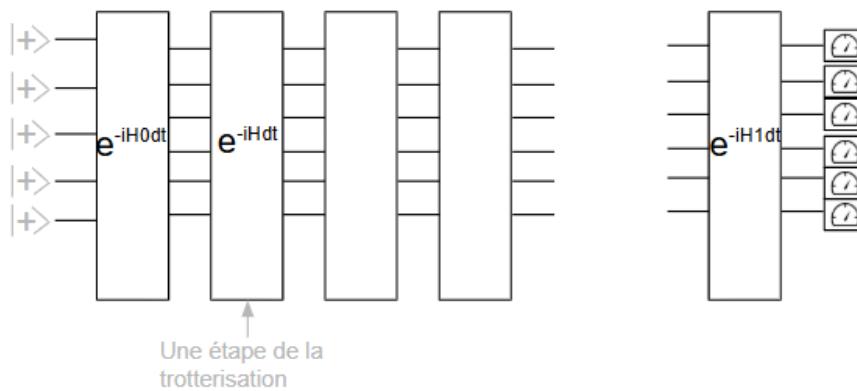


FIGURE 7 – Trotterisation de l'opérateur d'évolution U

La figure 7 représente un circuit quantique (on voit les qubits à gauche, tous dans l'état $|+\rangle$) dans lequel on applique l'opérateur unitaire $e^{-iH(\lambda_j) dt}$ par trotterisation.

Ainsi, l'évolution de l'état quantique $|\psi(t)\rangle$ est approximée par une séquence d'opérateurs exponentiels appliqués successivement. Chaque opérateur $\exp(-iH_j dt)$ correspond à le Hamiltonien à un instant j donné et peut être décrit par un ensemble de portes quantiques.

Cette évolution est modélisée par un circuit quantique où les opérateurs exponentiels sont appliqués à chaque qubit à chaque étape de la trotterisation.

Le grand défi est donc de transformer l'opérateur exponentiel associé à une étape de trotterization en une suite de portes quantiques que nous pourrons appliquer à nos qubits. L'enjeu est également de limiter ce nombre de portes, car c'est celui-ci qui caractérise en partie la complexité du QAA et qui peut mener à la décohérence des qubits. De plus, pour un λ donné :

$$e^{-iH(\lambda)dt} = e^{-i\lambda H_1 dt - i(1-\lambda)H_0 dt} \approx e^{-i\lambda H_1 dt} e^{-i(1-\lambda)H_0 dt}$$

Ceci nous permet d'estimer le nombre de portes par étape de trotterization, ce qui nous permettra également d'estimer notre plafond de nombre d'étapes de trotterization.

1.4 IMPLÉMENTATION COMPLÈTE POUR $N = 2$ ET RÉSULTATS NUMÉRIQUES

Nous choisissons de présenter dans cette partie les premiers résultats concrets, obtenus pour des graphes à 2 sommets. En effet, bien que l'intérêt mathématique de tels graphes soit quasi nul, ils permettent de bien illustrer tout le processus de construction, d'exécution et enfin d'analyse du circuit quantique, qui a pour l'instant seulement été expliqué en théorie.

Pour $N = 2$, nous opérons seulement sur deux qubits. De plus la fonction de coût se résout à seulement trois termes non constants, et le Hamiltonien se met facilement sous la forme de Ising. Il faut alors transformer l'exponentielle de ce Hamiltonien en une suite de portes. Nous le faisons dans un premier temps à la main pour estimer justement ce nombre de portes, mais Qiskit, la bibliothèque que nous utilisons, propose des outils permettant d'automatiser cela. Pour chaque étape de trotterisation, nous arrivons à un total de 7 portes quantiques, et le qubit qui en subit le plus en subit 5. La figure 8 est une représentation du circuit quantique pour une étape de trotterization, encodant l'opérateur $e^{-i\lambda H_1 dt} e^{-i(1-\lambda)H_0 dt}$.

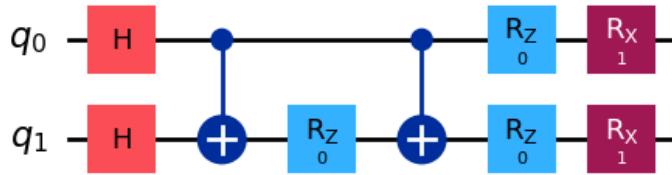


FIGURE 8 – Portes quantique pour une étape de discréétisation

On peut remarquer les portes H , de Hadamard, en début de circuit afin de se placer dans l'état fondamental $|+, +\rangle$ de notre Hamiltonien initial. On voit ensuite les 7 portes susmentionnées : deux C-NOT et cinq rotations (dont les paramètres dépendent de λ et des graphes).

Nous utilisons ensuite la bibliothèque **Qiskit** sur Python, mise à disposition par IBM pour d'abord coder une simulation du QAA, puis une véritable exécution sur QPU (Quantum Processor Unit, c'est à dire ordinateur quantique). Nous avons ici rencontré pas mal de difficultés liées à l'opacité des fonctions Qiskit, mais avons finalement réussi à obtenir des résultats probants.

L'exécution se fait à distance sur un des ordinateurs quantiques d'IBM, disponibles à hauteur de 10 minutes par mois. Ceux-ci comptent une centaine de qubits. Lors de l'exécution du programme, un QPU nous est attribué en fonction de la disponibilité. Nous entrons alors en file d'attente, de durée variable (nous avons eu des durées de 2 minutes à 10 heures d'attente), puis notre programme est exécuté. Pour $N=2$, l'exécution prenait de 3 à 10 secondes. Nous commençons toujours par une simulation : en effet il existe des outils permettant de simuler un ordinateur quantique parfait. Bien sûr, cette simulation est classique, donc pour de grands graphes, on aura les soucis de complexité classique.

Nous présentons sur les figures 10 et 11 les résultats dans deux cas différents : lorsque les graphes sont isomorphes, et lorsqu'ils ne le sont pas. Les résultats ont été obtenus pour 4 étapes de trotterisation, soit une profondeur d'une cinquantaine de portes. Voici tout d'abord les résultats de la simulation, pour 1024 shots, puis de l'exécution réelle sur le QPU d'IBM de Brisbane (127 qubits), pour deux graphes isomorphes représentés figure 9.

Graphe G (en bleu) et G' (en rouge) : isomorphes

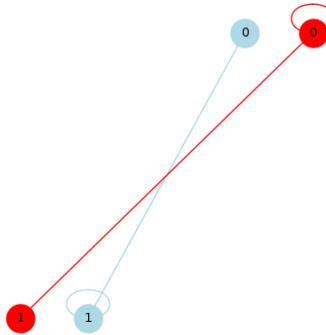


FIGURE 9 – Deux graphes isomorphes

On peut ici aisément déterminer l'isomorphisme envoyant G sur G' : il s'agit de la permutation envoyant le sommet 0 sur 1, et inversement. Ce que renvoie l'algorithme est affiché figure 10.

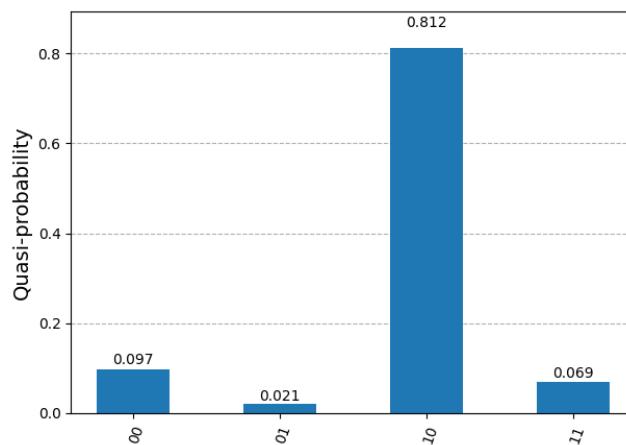


FIGURE 10 – Probabilité d'apparition pour chaque état, 1024 shots (simulation)

Comment lire ce graphique ?

Pour chaque "shot", la mesure finale a comme effet de **fixer**(projeter) chacun des qubits dans un des états de la superposition dans laquelle il se trouve, avec des probabilités différentes pour chaque état. C'est pourquoi pour obtenir une bonne estimation de l'état final réel du qubit (superposé ou pas), il faut faire un **très grand nombre de mesures**, donc de shots. On trace ensuite la fréquence à laquelle chacun des états a été mesuré en sortie, et cela donne une bonne idée de l'état final du qubit (si le nombre de shots est suffisant).

L'état 10 (lire $|1,0\rangle$) correspond donc à un tirage où le qubit 0 a été mesuré en sortie dans l'état $|1\rangle$, le qubit 1 dans l'état $|0\rangle$. Le QAA laisse le système dans l'état fondamental, ainsi l'état final est l'état fondamental du Hamiltonien "cible", c'est à dire l'état propre associé à la valeur propre égale au minimum de la fonction de coût. Dans ce cas, on mesure l'état 10 en sortie dans plus de 80% des cas. On en déduit qu'il **s'agit de notre état final**. En effet, la mécanique quantique étant **probabiliste**, on n'aura jamais 100% dans un seul état : il y a forcément du bruit (du fait notamment d'erreurs accumulées lors du passage dans les portes, ou lors de la mesure).

Afin de le relier simplement à notre problème de graphes, il suffit de savoir que pour $N = 2$, il se trouve que le premier chiffre de l'état vaut $s(0)$, le second $s(1)$. On va donc regarder si l'état final, 10, représente bien un isomorphisme de G vers G' . Pour cela il suffit de calculer notre fonction de coût pour la permutation $s(0) = 1; s(1) = 0$. Si on trouve 0, les graphes sont isomorphes. Sinon, ils ne le sont pas. Dans ce cas on trouve $C(s) = 0$, donc G et G' sont isomorphes par s .

On retrouve bien l'**isomorphisme que l'on avait intuité plus tôt**.

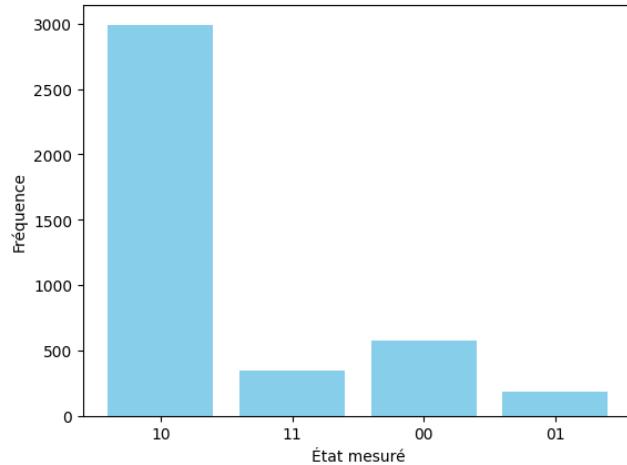


FIGURE 11 – Fréquence d'apparition de chaque état, 4096 shots (exécution réelle)

L'exécution réelle de la figure 11 nous donne le même résultat, avec une probabilité de mesurer 10 en état final qui vaut 73%. L'exécution sur QPU réel fonctionne.

Voici maintenant les résultats (cf figures 13 et 14) pour les deux graphes non isomorphes de la figure 12.

Graphe G (en bleu) et G' (en rouge) : non isomorphes

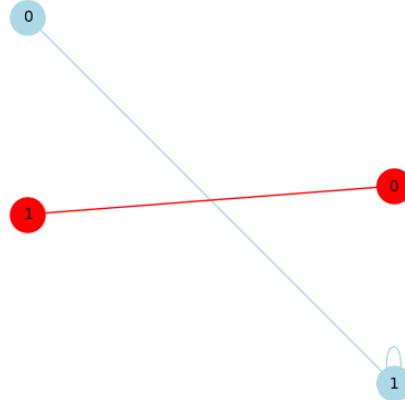


FIGURE 12 – Deux graphes non isomorphes

On peut facilement se convaincre que ces graphes ne sont pas isomorphes en comptant le nombre d'arêtes total de chacun des deux. Ils ne sont pas identiques, donc il n'y a aucune chance que les graphes soient isomorphes.

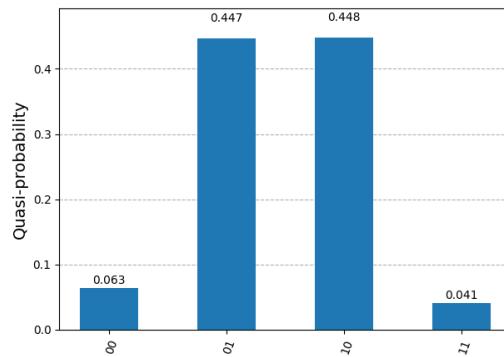


FIGURE 13 – Probabilité d'apparition pour chaque état, 1024 shots (simulation)

Cette fois-ci, on voit que les états 01 et 10 ont été les plus mesurés. Ils correspondent respectivement à la permutation "identité" et au "flip". On va donc regarder s'ils représentent bien une permutation de G vers G'. Pour cela on calcule notre fonction de coût pour les permutations associées. Si on trouve 0, les graphes sont isomorphes. Sinon, ils ne le sont pas.

Sur la figure 14, la fonction de coût vaut 1 dans les deux cas : les graphes ne sont pas isomorphes. C'est correct.

Ceci dit, l'observation de deux pics en 10 et 01 laisse à penser que l'état final n'est pas 01 ou 10, mais bien la *superposition équiprobable* des deux (ici intriquée).

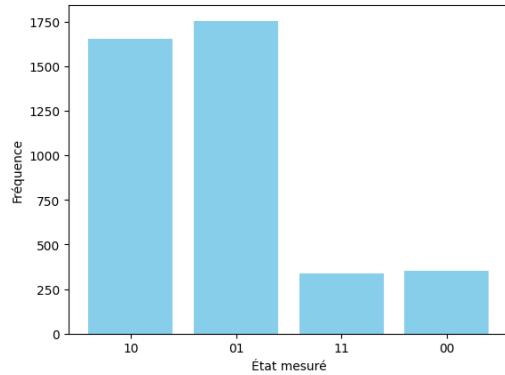


FIGURE 14 – Fréquence d'apparition de chaque état, 4096 shots (exécution réelle)

Il existe plusieurs façons de mesurer un système quantique. Nous avons pour l'instant choisi d'utiliser l'outil "Sampler" qui permet de projeter chaque qubit sur une observable, ici Z.

Le cas $N = 2$ est mathématiquement peu intéressant mais il permet de montrer concrètement comment se **construit**, s'**exécute**, et s'**analyse** le circuit quantique. **Simulations comme exécutions réelles fonctionnent parfaitement**. Il nous a également permis de nous familiariser avec les outils de Qiskit, à une échelle qui permet encore les calculs à la main.

2

PROBLÈMES RENCONTRÉS EN PRATIQUE

Dans cette partie, nous présenterons les problèmes rencontrés en pratique, donc les processus que nous avons tentés de mettre en place pour les contourner, et en parallèle nous exposerons nos résultats pratiques pour des graphes plus grands que 2 sommets.

2.1 TEMPS DE COHÉRENCE

L'une des principales limites associées à tout ordinateur quantique est le **temps de cohérence** de ses qubits. En effet, un système quantique comme un qubit n'est jamais coupé du monde : il interagit avec son environnement. Ces interactions résultent en une perte (progressive) des propriétés quantiques du qubit, notamment de sa capacité à exister dans une superposition d'états : on parle de **décohérence**.

Dès lors, un programme quantique **doit s'exécuter dans un temps inférieur au temps de cohérence**, sans quoi les qubits cessent de se comporter de manière quantique. Si cette limite est dépassée, les résultats obtenus ne sont plus exploitables : le calcul quantique perd tout son intérêt. C'est pourquoi le temps de cohérence constitue la première contrainte à prendre en compte dans le design d'un algorithme quantique : il borne toutes les autres décisions, notamment le nombre de portes, la profondeur du circuit, ou encore le nombre d'étapes de trotterisation.

Dans notre cas, les ordinateurs quantiques d'IBM que nous avons utilisés présentaient un **temps de cohérence moyen de 100 microsecondes**. Cela signifie que l'exécution de notre programme devait obligatoirement rester en-dessous de 100 microsecondes. C'est en fonction de cette contrainte que nous avons ensuite tenté d'optimiser les autres paramètres de l'algorithme, en particulier le nombre d'étapes de trotterisation.

ibm_sherbrooke			
Details			
Qubits	2Q error (best)	2Q error (layered)	CLOPS
127	2.85e-3	1.45e-2	150K
Status:	Region:	Processor type ⓘ:	Version:
● Online	us-east	Eagle r3	1.6.86
Total pending workloads:	Your instance usage:	Basis gates:	Median ECR error:
2 jobs	21 jobs	ECR, ID, RZ, SX, X	7.292e-3
Median SX error:	Median readout error:	Median T1:	Median T2:
2.117e-4	1.758e-2	290.8 us	181.6 us

FIGURE 15 – Propriétés d'une des machines d'IBM

Le temps de cohérence correspond à "Median T2" sur l'image. On retrouve plein d'autres caractéristiques du processeur, qu'il convient de prendre en compte lorsque l'on crée l'algorithme.

2.2 CHOIX DU PAS DE TROTTERISATION

On a en fait deux types d'étapes de Trotter :

- **Étape de type T** : qui détermine le pas de temps choisi pour diviser notre Hamiltonien ;
- **Étape de type D** : qui correspond au nombre d'étapes faites pour faire l'approximation :

$$(e^{\frac{A+B}{D}})^D \approx (e^{\frac{A}{D}} e^{\frac{B}{D}})^D$$

Ici A et B sont des matrices donc e^A et e^B ne commutent pas nécessairement.

Les étapes de trotterisation de type T sont celles qui ont été présentées figure 7 sous le nom N, et correspondent donc à l'application d'une porte $e^{-iH(\lambda)dt}$.

Les étapes de type D, quant à elles, correspondent à un découpage de chaque étape de type T en D sous-étapes, en ce que l'application de $e^{-iH(\lambda)dt}$ demande d'approximer l'exponentielle d'une somme d'opérateurs (ne commutant pas) par le produit des exponentielles. Cette approximation est valide pour des opérateurs "petits" : on préfère donc appliquer **D portes** $e^{-i\frac{H(\lambda)}{D}dt}$ successives qu'une seule $e^{-iH(\lambda)dt}$. **On fait donc en tout $T \times D$ étapes.**

On cherche donc à trouver le nombre optimal d'étapes de Trotter T et D en fonction du graphe pour minimiser le temps de calcul et avoir un résultat le plus fiable possible. Nous avons mené cette recherche sur les simulateurs (qui simulent un ordinateur quantique parfait non bruité) : de façon générale, on essaie de limiter les exécutions sur QPU car on est restreint à 10 minutes par mois d'exécution (et on arrive rapidement à quelques minutes d'exécution pour un seul programme).

• OPTIMISATION SUR SIMULATEUR

On observe que le nombre d'étapes de trotterisation semble lié au nombre de sommets mais aussi au nombre d'arêtes du graphe pour obtenir des résultats à la fois exploitables et corrects.

On effectue des tests avec $N = 2, 3$ et 4 . Au-delà, les simulations deviennent trop longues et les graphes trop denses, car le nombre d'états explose.

Pour les tests, nous avons fait varier les valeurs de T et D, tout en fixant un nombre de shots (nombre d'exécutions du circuit) constant et élevé ($\sim 10^6$), afin d'éliminer les échecs dus à un faible échantillonnage. Nous prenons des graphes générés aléatoirement.

On constate que T et D ne doivent être ni trop petits, ni trop grands. En effet, on a un **compromis** à faire : augmenter le nombre d'étapes de trotterisation améliore la précision de l'algorithme adiabatique (le pas de temps est plus court), mais augmente le nombre de portes traversées. Or chaque porte induit une petite erreur sur les qubits. En mettre trop engendre donc beaucoup de bruit. A l'inverse, si T et D sont trop faibles, on a un pas de temps insuffisant et un algorithme incorrect.

Par exemple pour 4 sommets ($n = 4$) et les graphes suivants (tirés de l'exemple en introduction) :

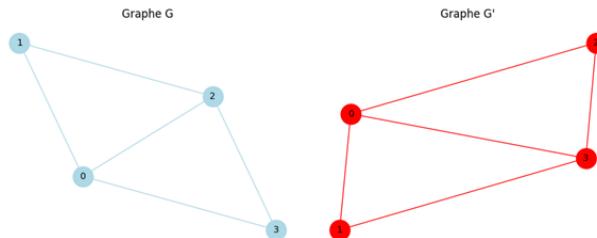


FIGURE 16 – Graphes à 4 sommets isomorphes

- Les résultats pour $T < 5$ et $D < 5$ sont **trop bruités** pour être exploitables ; [17] ;
- Pour $T = D = 5$, le résultat est **exploitable** et renvoie la bonne solution [18] ;
- Pour $T > 9$ ou $D > 8$, le résultat est en général soit **trop bruité**, soit **incorrect** [19].

Pour la lecture des résultats, en abscisse on a toutes les séquences binaires possibles, et en ordonnée leur fréquence d'apparition lors de la mesure (donc la probabilité que l'état final soit cette séquence binaire).

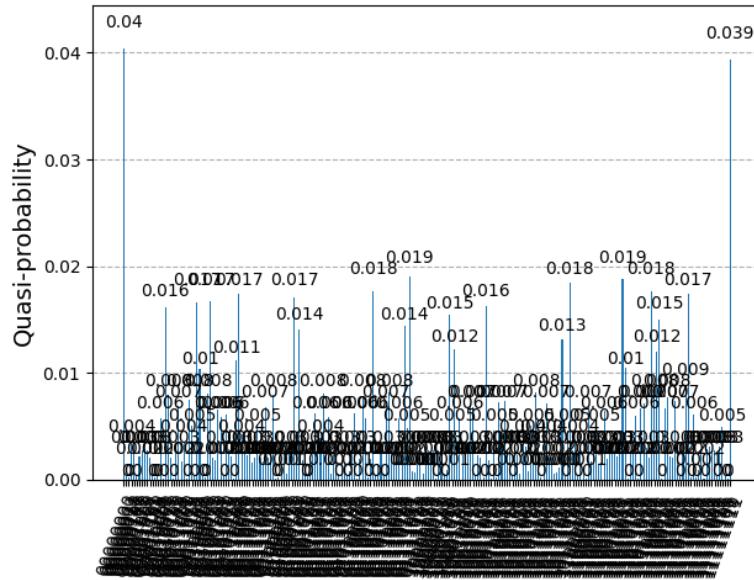


FIGURE 17 – Implémentation théorique pour 4 sommets et $T=2$, $D=2$

On observe deux pics, mais correspondant aux séquences 00000000, 11111111, ce qui correspond aux fonctions $(0, 0, 0, 0)$ et $(3, 3, 3, 3)$. Le résultat est évidemment incorrect, le nombre d'étapes de trotter est **insuffisant**.

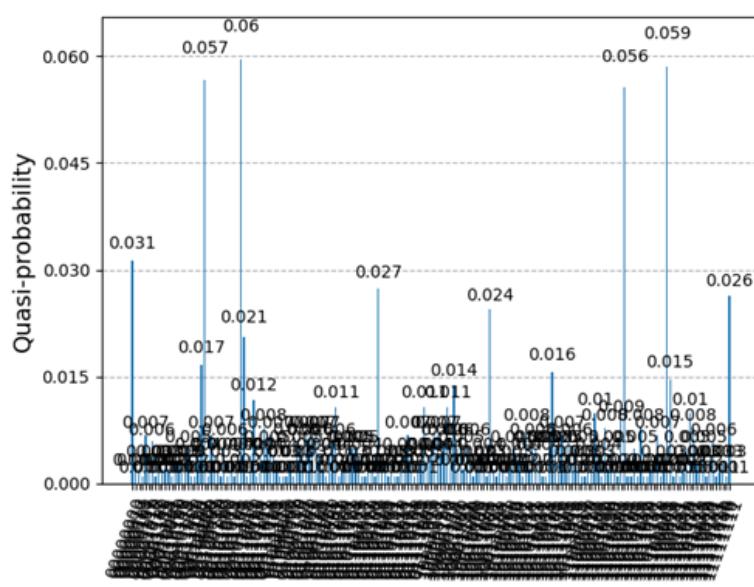


FIGURE 18 – Implémentation théorique pour 4 sommets et $T=5$, $D=5$

Les 4 permutations correspondant aux pics sont : 11010010, 11100001, 00101101, 00011110, soit, retraduites en décimal : (3, 1, 0, 2), (3, 2, 0, 1), (0, 2, 3, 1), (0, 1, 3, 2). On vérifie bien que ce sont des isomorphismes de G sur G' . Pour un nombre d'étapes de trotter correct, l'**algorithme simulé fonctionne bien**.

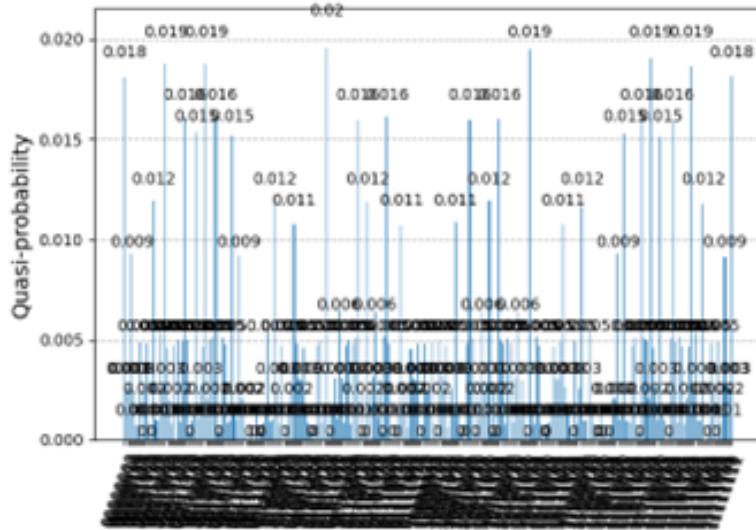
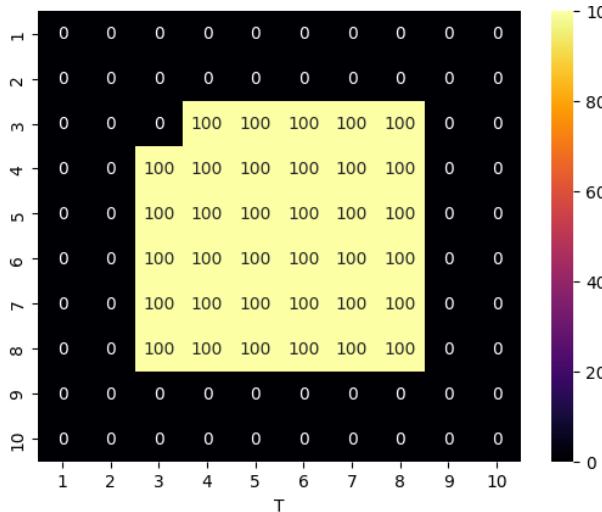


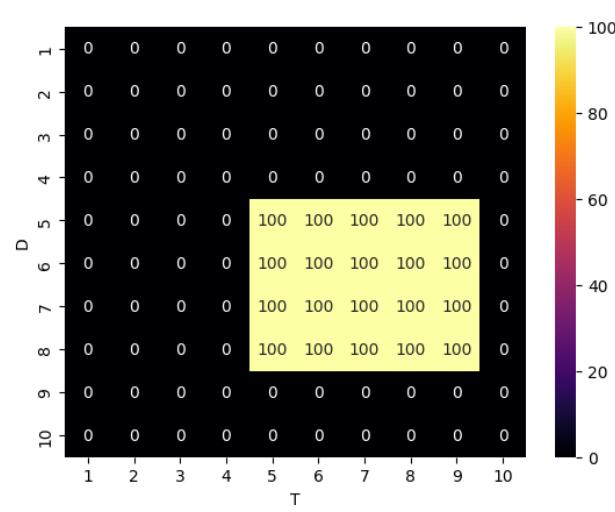
FIGURE 19 – Implémentation théorique pour 4 sommets et $T=10$, $D=5$

Dans ce cas, il y a **trop d'étapes** de trotterisation, le résultat est trop bruité et inexploitable.

Voici enfin, pour illustrer l'impact du nombre d'arêtes sur le choix du pas de trotterisation, les fréquences de réussite de la simulation en fonction des valeurs de T et D , pour un graphe à 4 sommets et 2 arêtes, puis 4 sommets, 5 arêtes.



(a) Heatmap 4 sommets, 2 arêtes



(b) Heatmap 4 sommets, 5 arêtes

FIGURE 20 – Fréquence de réussite T vs D , 5000 shots

On constate en effet dans les deux cas l'existence d'une zone pour laquelle l'algorithme simulé réussit (soit il réussit toujours, soit il ne réussit jamais dans ce cas) : il faut que T et D ne soient ni trop faibles, ni trop

grands. On remarque également que lorsque le nombre d'arêtes augmente, à sommets fixés, le minimum d'étapes nécessaires augmente également.

De plus, les deux types d'étapes sont *a priori* indépendants. On constate que pour 5 arêtes, pour le couple ($T = 9, D = 8$), l'algorithme renvoie une réponse correcte, alors que pour le couple ($T = 8, D = 9$), il renvoie une réponse incorrecte. Donc, pour $T \times D = 72$ étapes de Trotter, on obtient 2 résultats différents : les facteurs T et D jouent bien deux rôles distincts dans notre algorithme. Cette indépendance est d'autant plus marquée pour des graphes plus complexes.

Le test a été généralisé pour les autres graphes, et a été poussé avec 8, 9 et 10 arêtes, et les résultats obtenus sont similaires. Il est nécessaire d'avoir au moins autant d'arêtes que de valeurs de D et T pour obtenir des résultats significatifs, et les valeurs de T ou D inférieures au nombre d'arêtes donnent des résultats faux ou inexploitables.

Ceci étant, cette partie sur simulateur permet de vérifier que **notre Hamiltonien et notre algorithme sont corrects**, en ce que l'exécution (simulée) sur un QPU parfait fonctionne, en choisissant un bon nombre d'étapes de trotter. Il faut désormais se concentrer sur l'optimisation réelle (sur QPU non simulé).

• OPTIMISATION SUR MACHINE RÉELLE

Il est bon de rappeler que le simulateur simule un ordinateur quantique parfait non bruité. Le problème principal du simulateur sera donc en termes de complexité, mais pas en termes d'erreur, de bruit ou de décohérence. Là où le QPU peut être meilleur, et c'est tout l'intérêt, c'est en complexité et en temps pour des graphes bien plus grands.

En pratique, le temps de cohérence des qubits d'IBM nous impose une contrainte qui ne nous permet pas de résoudre le problème avec beaucoup d'étapes de Trotter. Par exemple, sur la figure 21, on constate qu'avec 4 sommets, on ne peut pas imposer $D \geq 1$ sans dépasser le temps de cohérence des qubits.

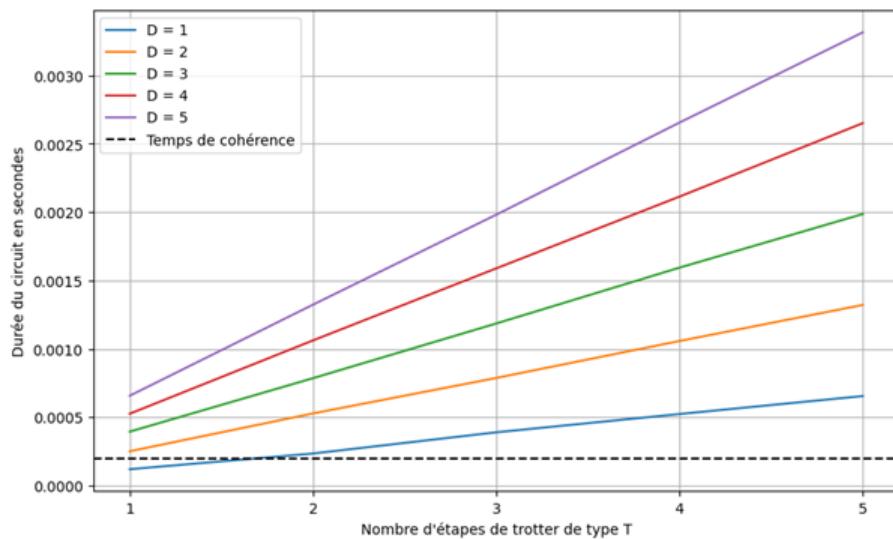


FIGURE 21 – Durée du circuit en fonction du nombre d'étapes de Trotter

Ainsi, on ne peut pas avoir le nombre correct d'étapes de Trotter (5×5) tout en restant sous le temps de cohérence. Pour l'exécution réelle, il faut donc **modifier notre Hamiltonien** (cf 2.3), faire de la **mitigation d'erreur** (cf 2.5), ou encore augmenter le nombre de shots, en espérant que ces deux dernières méthodes compensent le bruit lié à la décohérence (pas complète, on l'espère!).

Concernant le nombre de shots, en dessous de 500, les résultats ne sont pas très pertinents. La figure 22 montre une évolution linéaire du temps d'exécution en fonction du nombre de shots (entre 500 et 10^6 shots et toujours en simulation). La relation linéaire nous pousse donc à prendre un grand nombre de shots, tout en restant sous la limite des 10 minutes par mois sur les machines d'IBM qui nous est imposée (encore une contrainte temporelle).

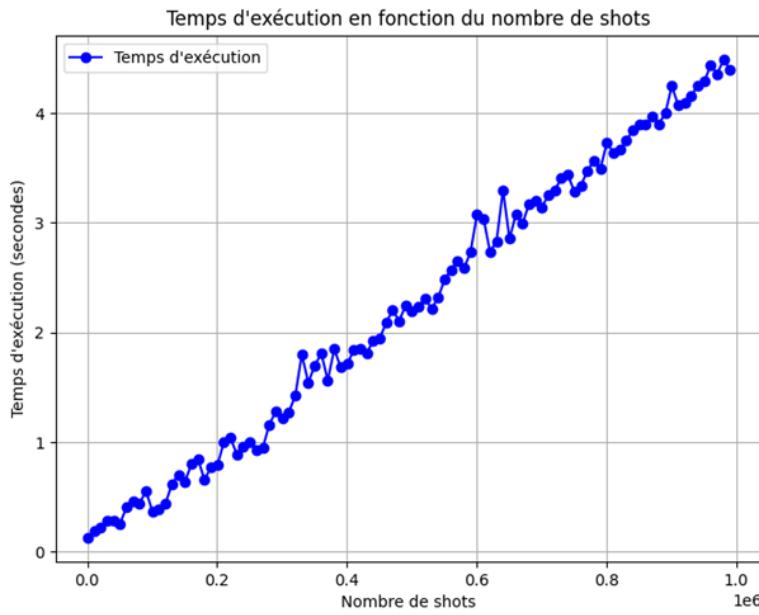


FIGURE 22 – Temps d'exécution en fonction du nombre de shots

En se basant sur toutes ces conclusions, on exécute notre programme sur machine réelle en choisissant $T = D = 2$ pour 2 sommets pour vérifier sa fiabilité lorsque l'on ne dépasse pas le temps de cohérence. Le résultat est correct et a été présenté en section 1.4. Cependant, au-delà de 2 sommets, le bruit est trop important : voici les résultats que l'on a obtenus pour des graphes plus grands.

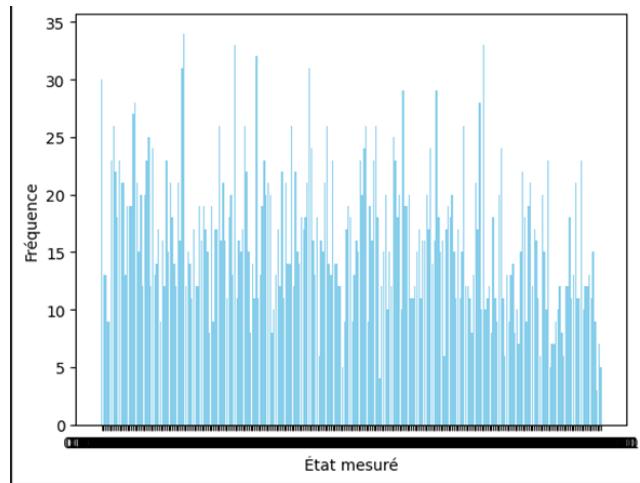


FIGURE 23 – Implémentation réelle pour 4 sommets

Ceci dit, la vérification de l'isomorphisme entre deux graphes (i.e. si on se donne une permutation, déterminer si elle correspond à un isomorphisme entre les deux ou pas) peut être effectuée très rapidement sur un ordinateur classique.

Ainsi, pour chaque shot, on peut extraire l'état final et vérifier s'il correspond à un isomorphisme valide. L'avantage de cette approche est qu'il suffit qu'un seul état final parmi tous les échantillons corresponde à un isomorphisme pour conclure que les deux graphes sont isomorphes. En effet, l'existence d'un unique isomorphisme suffit.

Néanmoins, cela n'a de l'intérêt que si la fréquence d'apparition de cet état final est supérieure à $1/N!$, sans quoi il serait plus coûteux que le test exhaustif de toutes les permutations, qui en compte précisément $N!$. Pour des graphes de taille $N = 3$, on trouve pour 10 000 shots, une fréquence d'apparition de 0,048, trois fois plus grand que $1/N!$. Mais pour des graphes plus grands, ça ne marche plus : pour des graphes de taille $N = 4$, on trouve pour 10 000 shots, une fréquence d'apparition de 0,0039, soit environ $1/2^{N \log_2 N} = N^{-N}$, bien plus petit que $1/N!$.

Cela suggère que les résultats obtenus ne reflètent aucune structure utile : un simple générateur aléatoire aurait donné des performances comparables. On en conclut que les sorties observées **relèvent essentiellement du bruit**.

C'est pourquoi notre attention s'est ensuite portée vers la mitigation d'erreur (cf 2.5), qui désigne un ensemble de méthodes destinées à atténuer les erreurs et donc le bruit, sans modifier le Hamiltonien.

2.3 LONGUEUR DU HAMILTONIEN

Avant de nous intéresser à la mitigation d'erreur, nous venons de voir qu'une possibilité était de modifier le Hamiltonien, de réduire son nombre de termes pour réussir à passer sous le temps de cohérence. C'est le sujet de cette section.

Comme détaillé en annexe 3.3.3, le Hamiltonien peut, *a priori*, comporter environ $2^{\text{nb qubits}}$ termes, chacun de la forme

$$Z_i = I_0 \otimes \cdots \otimes Z_i \otimes \cdots \otimes I_{L-1}.$$

Ce nombre de termes croît également comme n^n si n est le nombre de sommets du graphe. Théoriquement, cela ne pose pas de problème sur un ordinateur quantique parfait (sans bruit ni erreur), qui pourrait manipuler de tels espaces de Hilbert.

Cependant, dans la pratique actuelle, cette complexité constitue un frein majeur. Les ordinateurs quantiques ont un temps de cohérence limité, et chaque porte quantique introduit une certaine erreur. Il devient donc primordial de :

- limiter la profondeur des circuits (c'est-à-dire le nombre de portes successives),
- réduire le nombre total d'opérations pour rester sous la durée de cohérence,
- limiter les erreurs cumulées.

Ce sont ces limitations qui nous ont contraints à travailler avec de petits graphes (n petit). Cela est d'autant plus vrai que la **trotterisation** utilisée pour approximer l'évolution sous le Hamiltonien nécessite un grand nombre d'étapes lorsque ce dernier est complexe.

C'est pourquoi nous avons tenté de réduire sa taille en essayant d'autres approches, comme une nouvelle façon de représenter les permutations (théorique, et détaillé en annexe 3.3.3), mais sans succès.

Un autre obstacle courant en informatique quantique est la matérialisation physique des interactions à plusieurs qubits. Dès que l'on dépasse des interactions à 3 ou 4 qubits, leur implémentation devient extrêmement difficile, voire quasi irréalisable avec les technologies actuelles.

Ainsi, dans des approches comme le QAA, on impose que l'Hamiltonien soit de type **Ising** : c'est-à-dire qu'il ne contient que des interactions entre deux qubits au maximum. Pour respecter cette contrainte, il faut alors décomposer chaque interaction à plusieurs qubits en une somme d'interactions à deux qubits, ce qui augmente encore la taille et la complexité du Hamiltonien, donc du circuit quantique.

2.4 IDLE TIME

Une autre difficulté rencontrée concerne l'attribution des qubits sur les machines d'IBM.

Certaines portes de notre opérateur d'évolution trotterisé nécessite un couplage de plusieurs qubits. Ce couplage nécessite une proximité physique entre les qubits : ils doivent être "l'un à côté de l'autre" : on parle de **connectivité** des qubits.

A priori, les qubits attribués dans la machine d'IBM ne sont pas à proximité. Qiskit admet donc un transpilateur avant de compiler le code (qui optimise le circuit par rapport au hardware qui va être utilisé).

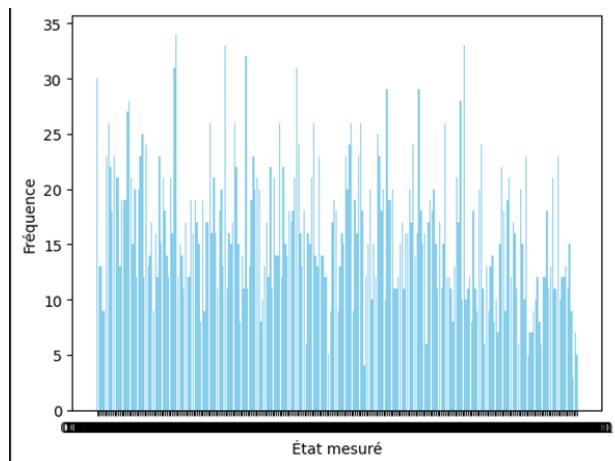
Le rôle du transpilateur est d'inclure des swap gate et d'assurer la bonne connectivité entre les qubits. Les swaps échangent l'état de deux qubits voisins. Ainsi, de proche en proche, le transpilateur permet de "rapprocher" deux qubits qui doivent être couplés.

La connectivité des qubits pose le problème de l"**"idle time"** - le temps nécessaire aux swaps pendant lequel aucune opération n'est effectuée - qui a deux conséquences principales :

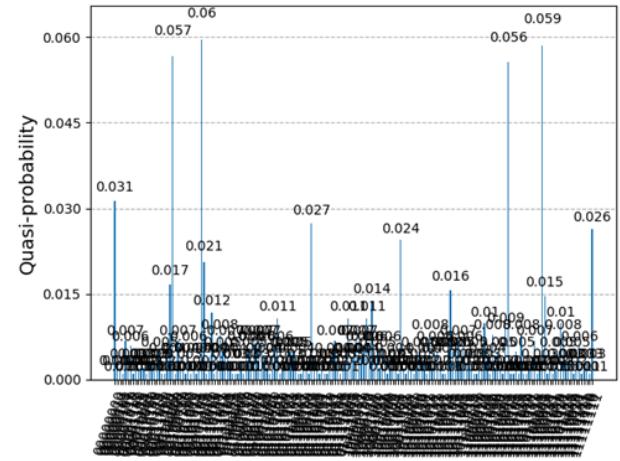
- La première est de **rallonger la durée du code**, ce qui concrètement nous empêche de traiter des graphes à plus de 5 ou 6 sommets ;

- La deuxième est d'influer sur la **cohérence** quantique des qubits et donc de réduire la fiabilité du résultat.

Concrètement, les résultats des programmes lancés sur les machines d'IBM sont très bruités par rapport à ceux de notre simulation numérique (voir figure 24).



(a) Implémentation réelle pour 4 sommets



(b) Implémentation théorique pour 4 sommets

FIGURE 24 – Exécution réelle (bruitée) vs simulée

2.5 MITIGATION D'ERREUR

Nous avons donc cherché à réduire ou supprimer le bruit en implémentant deux méthodes de mitigation d'erreur, domaine essentiel de la recherche actuelle en informatique quantique.

• DYNAMICAL DECOUPLING

Le **dynamical decoupling** consiste à "occuper" les qubits lors de leur temps de repos. Ces qubits passent par une séquence rapide de portes qui valent au final l'identité I_n . Le bruit est filtré et on gagne ainsi en temps

de cohérence sur les qubits.

Cette méthode de mitigation d'erreur a donc surtout un sens pour des opérations réparties localement au sein des qubits.

Le choix de la séquence de portes reste une source active de recherche encore aujourd'hui.

Dans notre cas nous avions le choix entre deux séquences de portes :

- **La séquence X, X, X, X** : en inversant régulièrement le qubit on évite d'accumuler du bruit selon une seule direction (par exemple Z) et on réduit **le déphasage**.
- **La séquence X, Y, X, Y** : en inversant le qubit régulièrement **selon deux directions**, on réduit non seulement **le déphasage** mais également **les erreurs transverses** (amplitude damping, rotation parasite). Cette séquence est donc plus intéressante car elle permet de supprimer une plus large gamme d'erreur.

Malheureusement, les ordinateurs d'IBM à notre disposition ne disposaient pas nativement de porte Y , et nous n'avons pas réussi à l'y implémenter manuellement.

En utilisant la séquence de portes X, X, X, X , nous analysons sur la figure 25 le mapping de ce dynamical decoupling. On voit, à gauche, les qubits sur lesquels on agit (au nombre de $4 \times \log_2(4) = 8$). En violet sont représentées les portes liées au dynamical decoupling. Enfin, en abscisse on retrouve la durée du circuit. Un "system cycle time" ou cycle d'horloge vaut ici 0.222ns donc le circuit dure quelques millisecondes (soit dix fois plus que le temps de cohérence.)

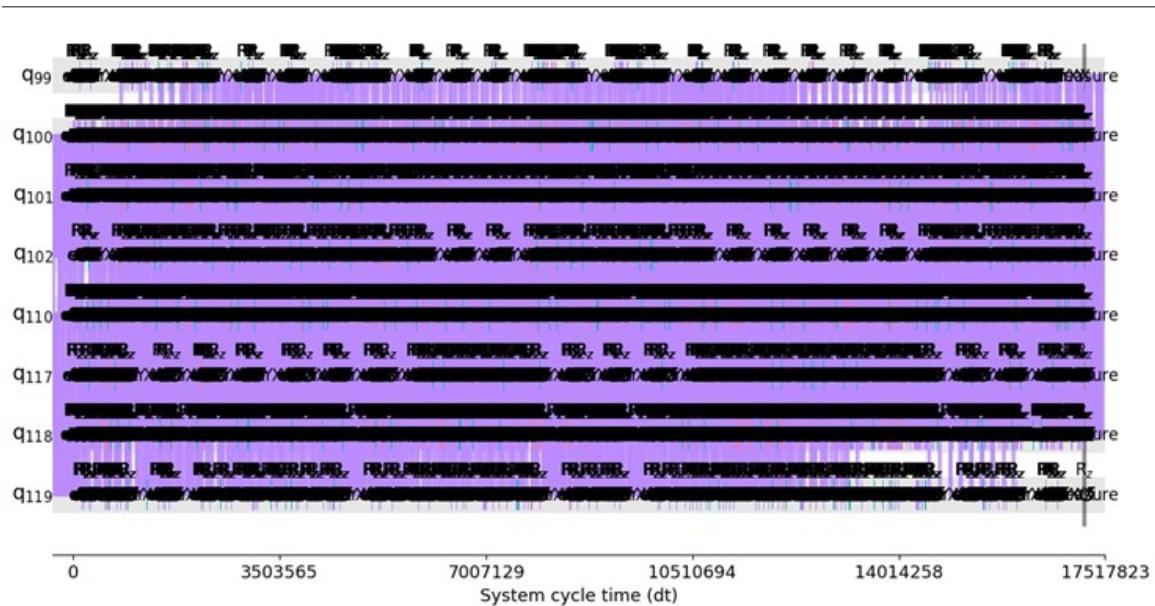


FIGURE 25 – Dynamical decoupling pour 4 sommets

Malgré le gain en temps de cohérence, on voit que le dynamical decoupling rajoute énormément de portes, et cela peut poser un problème car chaque porte est susceptible d'introduire une erreur.

Lorsque l'on fait tourner le programme, on obtient en effet des résultats toujours aussi bruités : le compromis temps de cohérence/erreurs liées à l'ajout de portes n'a pas été efficace (figure 26).

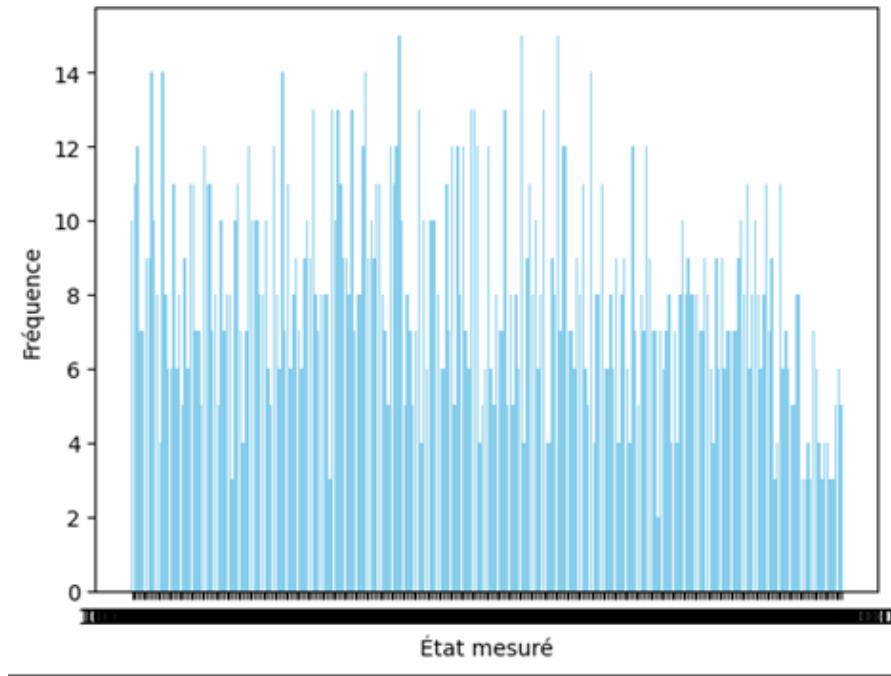


FIGURE 26 – Dynamical decoupling pour 4 sommets

- **ZERO NOISE EXTRAPOLATION**

La méthode de **Zero Noise Extrapolation (ZNE)** est une méthode de mitigation d'erreurs qui estime des observables sans l'influence du bruit. Elle repose sur l'idée d'exécuter un circuit quantique à différents niveaux de bruit amplifié, puis d'extrapoler les résultats pour estimer la valeur attendue en l'absence de bruit.

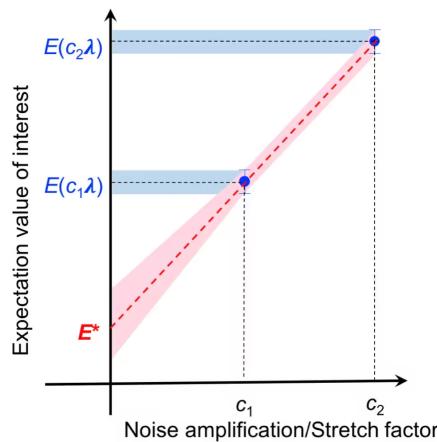
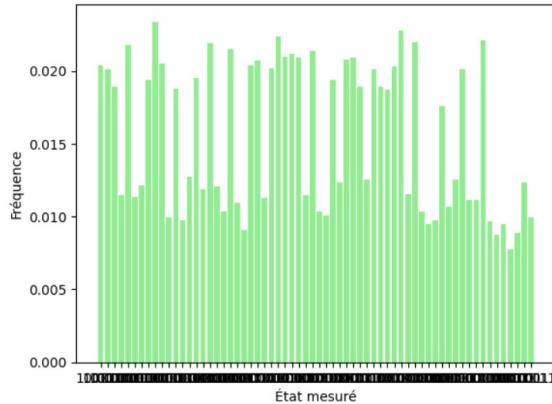


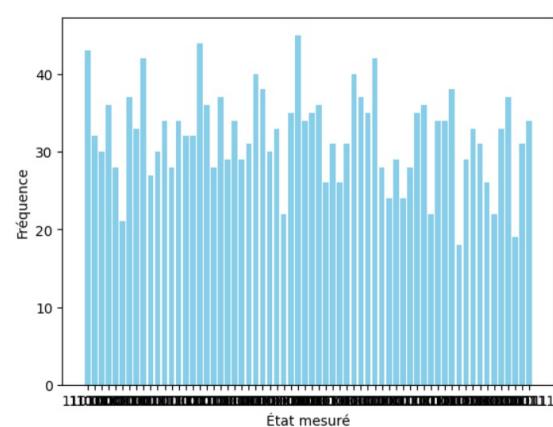
FIGURE 27 – Graphe d'extrapolation du résultat en fonction du bruit [3]

ZNE commence par exécuter le circuit de base avec le niveau de bruit nominal (facteur d'échelle 1). Ensuite, **le bruit est intentionnellement amplifié** en augmentant la profondeur du circuit, par exemple en répétant les portes bruitées comme les portes CNOT, pour des facteurs d'échelle supérieurs (par exemple, 1.5 et 2).

Pour chaque facteur, la valeur d'espérance d'un opérateur (comme le Hamiltonien du problème), c'est à dire l'estimation de l'observable associée, est calculée à partir des mesures. Ces valeurs sont ensuite utilisées pour construire un modèle mathématique, généralement une **extrapolation** linéaire, quadratique ou exponentielle, qui estime la valeur d'espérance au point où le facteur d'échelle est nul, correspondant à un circuit idéal sans bruit. Il est important de noter que cette approche suppose que le bruit varie de manière prévisible avec l'amplification.



(a) Implémentation sur machine réelle avec mitigation ZNE sur un graphe de 3 sommets avec extrapolation linéaire



(b) Implémentation sur machine réelle sans mitigation d'erreur

FIGURE 28 – Effet de la ZNE linéaire, 3 sommets

Dans le projet, nous avons appliqué ZNE sur un graphe à 3 noeuds, les résultats étant déjà satisfaisants pour un graphe à 2 noeuds, où les probabilités convergeaient correctement vers les états optimaux. Plus particulièrement, les résultats présentés ici sont ceux pour deux graphes isomorphes à 3 sommets et 1 arête. Le circuit QAA a été exécuté en utilisant des facteurs d'échelle de bruit de 1, 1.5 et 2 et une extrapolation linéaire. Les résultats ont montré une amélioration partielle des probabilités mesurées par rapport à une exécution sans ZNE, plusieurs états ne correspondant pas à la solution optimale étant beaucoup moins probables, comme le montre la figure 28. Cependant, les distributions restaient tout de même trop uniformes, indiquant que le bruit dominait encore les résultats.

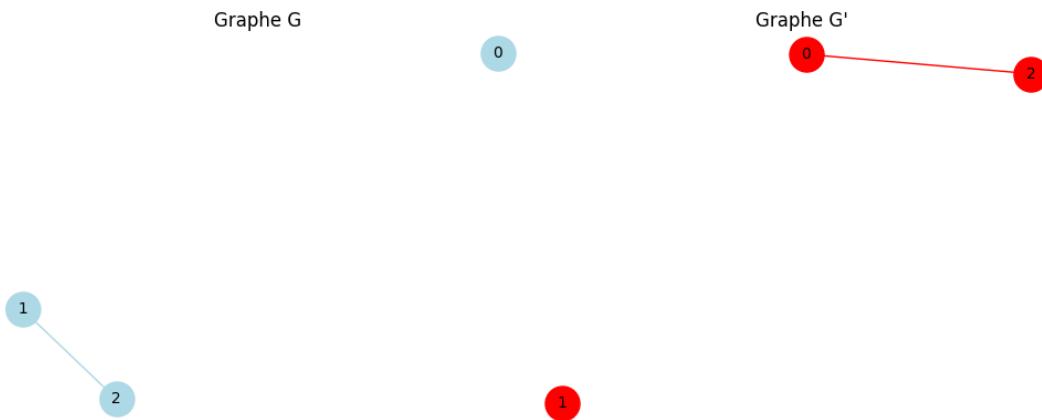


FIGURE 29 – Graphe de 3 sommets isomorphes

L'**extrapolation quadratique** donne des résultats beaucoup plus satisfaisants comme le montre la figure 30 où on constate que les résultats de la simulation (corrects) et de la machine réelle sont semblables. On ob-

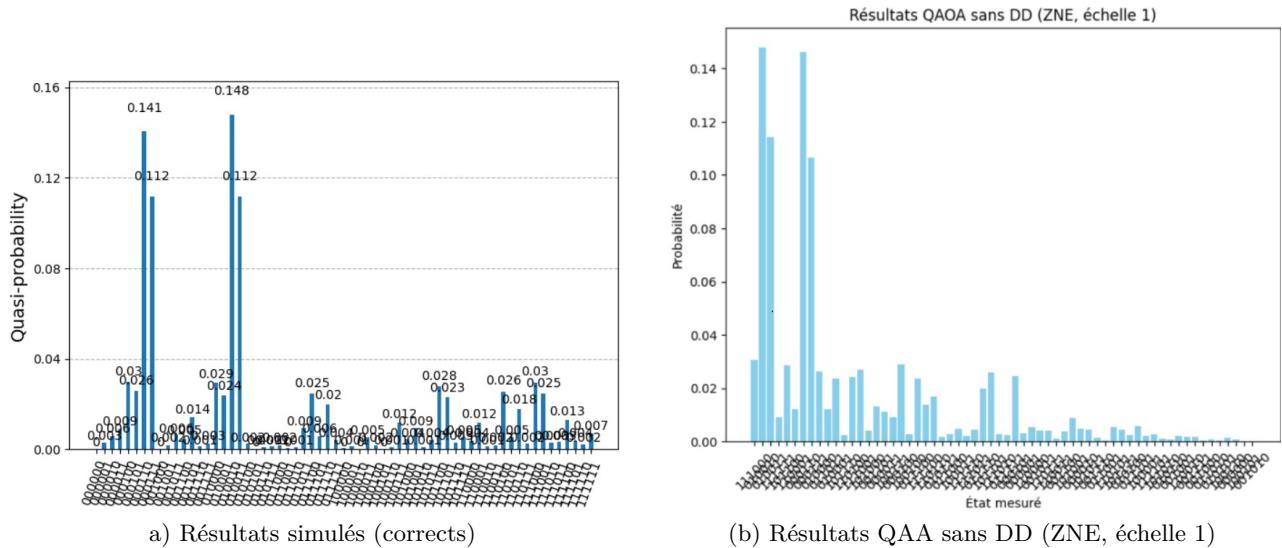


FIGURE 30 – Résultats ZNE avec interpolation quadratique (3 sommets, 1 arête)

tient un graphe **parfaitement exploitable** et des résultats attendus. Il s'agit de la meilleure performance sur ordinateur quantique réel que nous ayons pu obtenir, grâce à la ZNE (3 sommets, 1 arête).

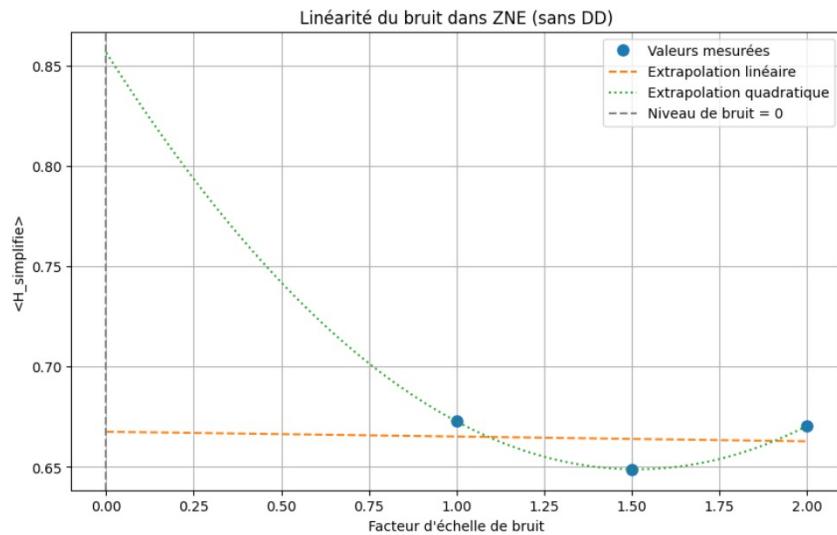


FIGURE 31 – Courbe d'extrapolation linéaire et quadratique

Le graphe de l'espérance du Hamiltonien en fonction du facteur d'échelle de bruit, figure 31, justifie la réussite de l'extrapolation quadratique devant celle linéaire.

Ces améliorations n'ont pas pu être approfondies en raison de la contrainte de temps imposée par IBM Quantum, limitée à 10 minutes par mois. Cela restreint le nombre d'exécutions, celles-ci devenant de plus en plus longues (en moyenne 5 minutes), et nous empêche également d'ajuster les paramètres de manière optimale. En effet, augmenter le nombre de facteurs d'échelle de bruit au-delà de trois ou agrandir le graphe entraînait des temps d'exécution dépassant cette limite.

• TWIRLED READOUT ERROR EXTINCTION (TREX)

La **Twirled Readout Error eXtinction** (TREX) est une méthode de mitigation d'erreur de *readout* (erreur de mesure) contrairement aux autres méthodes que nous avons utilisées, qui s'intéressaient aux erreurs faites avant la mesure. Concrètement, TREX procède en deux étapes :

- **Twirling (randomisation)** : On applique au hasard, avant la mesure, de petits « flips » (portes X) sur les qubits, puis on les ré-applique de manière classique sur les bits mesurés. Ceci uniformise le bruit de mesure, le transformant en un simple « facteur d'erreur » à corriger.
- **Calibration et correction** : On exécute d'abord quelques circuits de test préparés en $|0\rangle \dots |0\rangle$ pour estimer ce facteur d'erreur/biais. Ensuite, on ré-échelonne les résultats de notre programme avec ce facteur, ce qui efface la plupart des déformations dues à la mesure.

Nous avons tenté d'implémenter cette méthode de mitigation d'erreur. Cependant, nous n'avons pas utilisé la version développeur du module de Qiskit qui permet de disposer des bibliothèques `readout_extinction` et `twirl_readout`, mais sommes restés sur la version stable, qui ne propose pas (pour l'instant) d'API TREX. Nous n'avons donc tout simplement pas pu lancer le protocole de twirling/extinction de bruit de lecture en interne.

Nous avons certes envisagé une solution offrant un service de calibration TREX, mais celle-ci ne livre ni la séquence exacte de portes X appliquées, ni les valeurs brutes avant et après correction, ni les facteurs de ré-échelonnement calculés. En l'absence d'un accès direct à ces données, il nous était impossible de quantifier l'amélioration de notre métrique d'isomorphisme (ni même de visualiser localement l'impact sur chaque qubit), si bien que nous avons finalement dû écarter TREX de notre stratégie de mitigation.

3 CONDUITE DU PROJET

3.1 ORGANISATION TEMPORELLE DU PROJET

• OBJECTIFS DE SEPTEMBRE

Au début de notre PSC, en septembre, nous avions établi de respecter le déroulé suivant :

1. **Pseudo-algorithme quantique** : Mi-octobre ;
2. **Traduction en code fonctionnel** : Mi à fin janvier ;
3. **Exécution du code sur machine quantique** : Début mars ;
4. **Finalisation de la rédaction du rapport** : Mi-avril.

Nous avons réussi à respecter ce planning jusqu'en janvier, mais une difficulté majeure se posait à nous : le nombre de termes de notre Hamiltonien augmente très rapidement avec le nombre de sommets (cf 2.3).

De plus, pour pouvoir faire tourner le code sur les machines d'IBM, il devenait nécessaire d'optimiser le code.

• OBJECTIFS DE JANVIER

Au vu des difficultés rencontrées, nous avions choisi en janvier de réviser nos objectifs comme ceci :

1. **Estimer le nombre de portes de la trotterisation pour n sommets** : mi-février ;
2. **Faire tourner le code sur les machines d'IBM pour des graphes plus grands** : début mars ;
3. **Optimiser le code** : fin mars.

Après avoir estimé une complexité qui ne nous permettait pas de résoudre le problème pour de grands graphes, nous avons donc tenté de reformuler notre Hamiltonien sous une autre forme, mais sans succès.

En parallèle, nous nous sommes intéressés à la mitigation d'erreur pour réduire le bruit et essayer à nouveau d'augmenter la taille des graphes.

• CONCLUSION D'AVRIL

Finalement, la complexité du programme au vu du temps de cohérence des qubits actuels était un frein trop important pour résoudre le problème sur machine réelle pour des graphes avec $n=3$. Les limitations des bibliothèques gratuites et du temps d'exécution permis sur les machines d'IBM ne nous ont pas permis de pousser la mitigation d'erreur assez loin pour pouvoir augmenter la taille de nos graphes.

Toutefois, les résultats sur simulateur restent très encourageants et prouvent que la structure de notre code permet bien de résoudre le problème (du moins théoriquement).

3.2 RÉPARTITION DU TRAVAIL ET RELATION AVEC LE TUTEUR

• ORGANISATION DU DÉBUT DU PROJET

En septembre, notre objectif principal était de nous familiariser avec les notions complexes de notre sujet, tant sur le plan technique que théorique.

Pour cela, nous avions commencé à faire des réunions dès cet été avec notre tuteur. Il était aussi important, selon nous, de rencontrer notre tuteur hors écrans à (au moins) un moment dans l'année. Cette rencontre a permis d'échanger sur le sujet, de clarifier certaines notions et de réaxer notre travail dans la bonne direction.

Nous nous sommes donc formés sur le problème de l'isomorphisme de graphes à l'aide de différents articles conseillés par notre tuteur et nous nous sommes renseignés sur l'informatique quantique à l'aide du site d'IBM.

Les mercredis après-midi étaient des moments de rencontre évidents entre membres du PSC. C'est à ce moment-là que nous nous répartissions les différentes tâches de la semaine, afin de travailler efficacement.

• RÉORGANISATION DE JANVIER

En janvier, nous avions rencontré plusieurs problèmes complexes qui nécessitaient chacun un travail plus approfondi.

Afin de gagner en efficacité, nous avons repensé notre répartition des tâches au sein du groupe.

D'abord, chacun a choisi de s'intéresser ou bien à la physique du problème, ou bien à la rédaction du code. De cette manière, chacun a pu s'approprier les notions complexes du problème et les expliquer au reste du groupe afin que tout le monde ait une vision globale du projet.

Les appels avec notre tuteur sont restés réguliers, mais plus espacés car les objectifs que nous nous sommes fixés demandaient un travail plus long.

Enfin, en cas de difficulté, nous demandions conseil à notre tuteur.

Finalement, au vu des nombreux domaines auxquels notre projet touchait, chacun a réussi à y participer aussi bien sur l'aspect physique, que sur l'aspect informatique ou mathématique.

3.3 COMPÉTENCES ACQUISES

3.3.1 • FORTE AUTONOMIE

De par le fait que notre tuteur n'était pas présent physiquement à Polytechnique, nous avons dû travailler majoritairement en autonomie. Cela a été à la fois une difficulté et une opportunité.

Tout d'abord, le projet ne requérant d'autre matériel qu'un ordinateur connecté à Internet, il pouvait se tenir intégralement en distanciel.

Pour autant, il est clair qu'au cours du début du projet cela était parfois compliqué de savoir où aller. Nous nous sommes ainsi nous-mêmes fixés des objectifs. Ceux-ci se trouvèrent être trop ambitieux, entraînant un sentiment de découragement lors du mois de janvier. La réorganisation que nous avons alors opérée permit de faire un point sur les difficultés rencontrées et de réadapter nos objectifs par rapport à ceux-ci. Notre tuteur, Corentin, nous guida vers de possibles solutions. La prise de conscience que les problèmes que nous rencontrions étaient en fait tout à fait classiques, voire inhérents, à l'informatique quantique a entraîné un regain de motivation. Par la suite, des réunions de groupe plus régulières nous ont permis de fixer plus facilement des objectifs atteignables et à conserver une vision d'ensemble sur le projet.

3.3.2 • TRAVAIL DE GROUPE

Nous avons pu tout au long du projet apprendre à travailler en groupe sur un projet de taille aussi conséquente que celui-ci. Ainsi, au début nous avions du mal à nous répartir efficacement le travail. Nous avons alors essayé plusieurs choses. A l'origine, nous tentions de découper les tâches à effectuer en sous-tâches aussi indépendantes les unes des autres que possible afin d'ensuite se les répartir. Mais, cela fut assez vite un frein

à la communication au sein du groupe, chacun avançait de son côté sans avoir une vision claire du travail des autres. De plus, certains rencontraient des difficultés que d'autres avaient résolus. Bref, notre répartition était finalement assez peu efficace.

Après la réunion de cadrage d'octobre, nous primes conscience que nous manquions de communication au sein du groupe. Nous avons alors changé radicalement de tactique et travaillions plutôt tous ensemble sur le projet. Mais cela provoquait un déséquilibre dans la répartition du travail entre les membres et il était finalement compliqué de se motiver quand quatre autres personnes devaient de toutes façons faire la même chose que soi.

Finalement après la restructuration de janvier, nous nous sommes répartis en binômes/trinômes les différents problèmes complexes que nous avions cerné. Nous avons également décidé d'organiser plus régulièrement des réunions de groupe pour présenter l'avancement de chacun. Cette organisation a permis d'être bien plus efficace.

3.3.3 • INITIATION À LA RECHERCHE

Ce projet étant très orienté recherche, il nous a permis de commencer à nous accoutumer à ce milieu. Ainsi, dès nos premières réunions avec notre tuteur, afin de choisir le sujet sur lequel nous allions travailler, nous dûmes naviguer parmi divers articles de recherche afin de se faire une idée de l'état de l'art et afin de choisir un sujet sur lequel des choses ont déjà été faites afin que le projet soit malgré tout réalisable.

De plus, l'informatique quantique étant encore à ses balbutiements, la vaste majorité des ressources existantes sont issues du milieu de la recherche. Ainsi, tout au long du projet nous fûmes confrontés à ce milieu et à ses codes.

Enfin, dans l'avancement même du projet, si la première partie théorique est assez proche du texte de Frank Gaitan et Lane Clark [1], la seconde partie, elle, est intégralement de notre cru. Nous avons ainsi dû :

- étudier les différentes méthodes pour améliorer les résultats d'un algorithme quantique.
- confronter ces méthodes et décider, au vu de la forme de nos résultats, lesquelles nous allions creuser.
- implémenter directement ces méthodes
- évaluer leur impact en fonction de leurs différents paramètres et essayer de trouver ces meilleurs paramètres.
- implémenter plusieurs de ces méthodes simultanément et donc trouver des synergies entre elles.

CONCLUSION ET REMERCIEMENTS

A travers ce projet scientifique commun, nous avons donc pu nous pencher sur la résolution algorithmique d'un problème mathématique complexe : le problème d'isomorphisme de graphe. Ce problème, nous avons tenté de le résoudre via l'informatique quantique. Cela nous a permis d'acquérir plusieurs notions fondamentales dans cette discipline.

Nous avons ainsi développé un algorithme quantique exécutable par les machines d'IBM. Après avoir étudié les résultats de cet algorithme tant sur les machines d'IBM que localement en simulant un ordinateur quantique, nous avons travaillé sur plusieurs méthodes pour améliorer les résultats. Ainsi, nous avons pu :

- Étudier notre Hamiltonien afin de tenter d'en réduire la complexité
- Mettre en place, et évaluer des méthodes pour choisir le pas de trotterisation
- Étudier, mettre en place et évaluer des techniques de mitigation d'erreur : le *Dynamical Decoupling*, ainsi que le *Zero Noise Extrapolation*.

Finalement, nous sommes parvenus à résoudre le problème d'isomorphisme de graphe pour des graphes d'environ 3 sommets sur les machines quantiques d'IBM. En simulant un ordinateur quantique parfait, nous parvenons à pousser cela à des graphes plus gros. Le facteur de limitation majeur étant ici non pas le nombre de qubits mais plutôt le temps de cohérence de ces qubits. Au vu des améliorations continues du temps de cohérence, nous pouvons raisonnablement considérer qu'un tel algorithme pourrait être exploitable dans un futur proche sur des machines quantiques plus performantes.

Ce travail nous a permis de nous essayer à un travail de groupe scientifique, de recherche et donc de commencer à acquérir de nombreuses compétences que nous aurons à mobiliser au cours du reste de notre scolarité à l'X (stage de recherche, 4A, ...).

Nous tenons à remercier notre tuteur Corentin Bertrand qui nous a guidés tout au long de ce projet. Il a su nous orienter efficacement, se rendre disponible malgré son travail et être particulièrement pédagogue.

Nous remercions également IBM pour la mise à disposition gratuite d'ordinateurs quantiques, particulièrement modernes et onéreux.

Enfin, merci à Alistair Rowe, François Ozanam et Guilhem Gallot, nos coordinateurs du département de physique de l'Ecole Polytechnique, pour leurs critiques constructives lors de la réunion de cadrage du 08 octobre, ainsi que pour la gestion administrative du projet.

ANNEXE

HAMILTONIEN DE ISING

Nous notons $L = N \times \log(N)$ le nombre de qubits nécessaires au QAA. Par définition, un qubit est un système quantique à deux niveaux (on peut par exemple se représenter un spin). H_1 agit donc dans un espace de Hilbert de dimension 2^L . Nous avons fait en sorte que les valeurs propres de H_1 soient exactement les $C(s)$ pour les permutations s . Pour ce faire, notant $|0\rangle$ et $|1\rangle$ les états respectivement fondamental et excité de l'opérateur Z (matrice de Pauli-Z), nous construisons aisément la base propre (tensorielle) de l'opérateur $Z_0 \otimes Z_1 \dots \otimes Z_{L-1}$ (où l'indice représente l'indice du qubit). Pour $b = 01\dots0$ une séquence binaire à L caractères associée à la permutation s , nous notons $|s_b\rangle = |0\rangle \otimes |1\rangle \dots \otimes |0\rangle$ l'état propre de la base tensorielle associée. Notre objectif est que

$$H_1 |s_b\rangle = C(s) |s_b\rangle \quad (7)$$

Ainsi, si $C(s) = \sum_{i,j} Q_{ij} z_i z_j + \sum_i b_i z_i$ où $z_i \in \{-1, 1\}$ (d'où l'intérêt de la reformulation de la permutation s en variables binaires!), alors on pose

$$H_1(s) = \sum_{i,j} Q_{ij} Z_i Z_j + \sum_i b_i Z_i$$

où $Z_i = I_0 \otimes \dots \otimes Z_i \otimes \dots \otimes I_{L-1}$, I représentant l'opérateur identité. On vérifie alors aisément (7).

Ce type de Hamiltonien est appelé Hamiltonien "de Ising". Il suppose uniquement des interactions entre deux qubits au maximum, mais toute interaction à n qubits peut s'y ramener.

REPRÉSENTATION DES PERMUTATIONS

Notre fonction de coût est donc une fonction qui prend en entrée une fonction s de $\{1, n\}$. Nous voulons représenter cette fonction par des variables binaires, qui correspondront aux qubits, afin de pouvoir mapper le Hamiltonien. Nous avons essayé deux approches, et c'est la première que nous avons utilisée dans nos résultats.

- PREMIÈRE APPROCHE

Cette manière de représenter une fonction par des variables binaires est celle utilisée par Frank Gaitan et Lane Clark [1]. C'est la plus simple possible. Une fonction s de $\{1, n\}$ dans $\{1, n\}$ est définie par ses images $s(0), s(1), \dots, s(n)$. On va décomposer chaque $s(i)$ en binaire, ce qui nous donne $\log_2(n)$ bits par $s(i)$, soit $n \times \log_2(n)$ qubits en tout. Par ailleurs, à toute séquence binaire de cette longueur, on associe bien une unique fonction de $\{1, n\}$. On a donc une bijection entre séquences binaires de longueur $n \times \log_2(n)$ et fonctions de $\{1, n\}$ dans $\{1, n\}$.

- DEUXIÈME APPROCHE

Cette seconde approche nous a été suggérée par notre tuteur, Corentin Bertrand.

Son objectif est de mettre en bijection l'espace des permutations de $\{1, \dots, n\}$, et non plus des fonctions quelconques de $\{1, \dots, n\}$ dans $\{1, \dots, n\}$.

En effet, jusqu'à présent, les parties C_1 et C_2 de la fonction de coût avaient pour unique but de vérifier que la fonction s en entrée était bien une permutation. Cela ajoutait beaucoup de termes à la fonction de coût, et donc au Hamiltonien.

C'est pourquoi nous avons souhaité essayer une autre approche, afin de réduire le nombre de termes du Hamiltonien.

Nous cherchons donc cette fois une bijection entre des **séquences binaires** (d'une certaine taille), et les permutations de $\{1, \dots, n\}$.

L'idée est la suivante : Prenons une permutation s de $\{1, \dots, n\}$. Nous appliquons à cette permutation la bijection :

$$u : k \mapsto (k + 1) \bmod n$$

appelée **up**. On obtient alors une nouvelle permutation dont toutes les images ont été **augmentées de 1** modulo n .

On répète ce processus jusqu'à obtenir une permutation, notée \tilde{s} , telle que l'image de n est égale à n . Autrement dit, on cherche l'entier u_n tel que :

$$\tilde{s}(n) = (u^{u_n} \circ s)(n) = n$$

Cette permutation \tilde{s} peut alors être vue comme une permutation de $\{1, \dots, n - 1\}$. On recommence alors le processus, et on trouve u_{n-1} tel que :

$$(u^{u_{n-1}} \circ \tilde{s})(n - 1) = n - 1$$

Et ainsi de suite, jusqu'à u_1 .

Ainsi, le vecteur $(u_1, \dots, u_{n-1}, u_n)$ constitue une **représentation de la permutation** s dans cette nouvelle approche.

Voici un exemple pour une permutation de taille 4. Considérons la permutation $(4, 3, 1, 2)$.

On applique u de la manière suivante :

$$\begin{aligned} (4, 3, 1, 2) &\xrightarrow{u} (1, 4, 2, 3) \\ &\xrightarrow{u} (2, 1, 3, 4) \end{aligned}$$

Le 4 est maintenant en dernière position. On a donc dû appliquer deux fois u pour cela, ce qui donne :

$$u_4 = 2$$

On retire ensuite le 4 de la dernière permutation obtenue et on recommence le processus avec $(2, 1, 3)$:

$$(2, 1, 3)$$

Le 3 est déjà en dernière position, donc :

$$u_3 = 0$$

On continue avec $(2, 1)$:

$$(2, 1) \xrightarrow{u} (1, 2)$$

Le 2 est en dernière position après une seule application, donc :

$$u_2 = 1$$

Enfin, on termine avec (1) , qui est déjà en dernière position, donc :

$$u_1 = 0$$

Cela nous donne $(0, 1, 0, 2)$ comme représentation de cette permutation.

En remontant dans les étapes, on se convainc aisément qu'il y a bien une bijection entre $\{0, \dots, n-1\} \times \{0, \dots, n-2\} \times \dots \times \{0\}$ et permutations de $\{1, n\}$ (ce qui est évident puisqu'ils ont même cardinal, mais il est important de trouver la bijection pour la suite).

La décomposition en binaire nous donne donc ensuite la bijection avec les séquences binaires de taille $\log_2(n!)$.

En plus de s'être débarassés de beaucoup de termes de la fonction de coût, on a donc aussi réduit le nombre de qubits de $n \times \log_2(n)$ à $\log_2(n!)$, ce qui n'a de l'intérêt que pour des n petits (asymptotiquement c'est équivalent).

Mais alors, pourquoi n'a t-on pas plutôt utilisé cette approche si elle n'a que des avantages ? En fait, la traduction de la condition d'isomorphisme dans la fonction de coût est bien plus difficile à faire en fonction des variables (u_1, \dots, u_n) qu'en fonction de $(s(1), \dots, s(n))$. (voir Annexe 3.3.3).

Alors, certes on supprime les termes de C_1 et C_2 de la fonction de coût, mais en fait le nombre de termes de la fonction C_3 explose.

Nous mentionnons ici cette approche en ce qu'elle est assez riche dans la compréhension de la difficulté du mapping d'un problème mathématique en un Hamiltonien, et en ce qu'elle peut tout de même être une piste d'amélioration.

DÉTAIL DE LA FONCTION DE COÛT

Nous détaillons ici les différentes contributions à la fonction de coût, ainsi que leur réécriture sous forme de Hamiltonien de type Ising.

Rappelons d'abord la forme générale des trois termes :

$$\begin{aligned} C_1(s) &= \sum_{i=0}^{N-1} \sum_{\alpha=N}^M \delta_{s_i, \alpha}, \\ C_2(s) &= \sum_{i=0}^{N-2} \sum_{j=i+1}^{N-1} \delta_{s_i, s_j}, \\ C_3(s) &= \|\sigma(s)A\sigma^T(s) - A'\|_2. \end{aligned}$$

Termes C_1 et C_2 : vérification de la validité d'une permutation. Le terme $C_1(s)$ est non nul si, et seulement si, il existe une image de s , notée $s(i)$, telle que $s(i) \in [N, M]$, où M est la plus petite puissance de 2 supérieure à N . En effet, puisque l'on décompose chaque entier en binaire, on teste des fonctions allant de $[1, N]$ vers $[1, M]$. Le terme C_1 s'assure donc que l'image de la fonction s est bien contenue dans $[1, N]$, ce qui est nécessaire pour qu'elle soit une permutation.

Le terme $C_2(s)$, quant à lui, est non nul si, et seulement si, s n'est pas injective. Ainsi, la condition :

$$C_1(s) + C_2(s) = 0$$

est équivalente au fait que s soit une permutation.

Le terme C_3 : vérification de l'isomorphisme. Toute la difficulté réside dans le terme C_3 , qui teste si s est un isomorphisme entre les graphes G et G' . Par définition des matrices d'adjacence, on a :

$$\begin{aligned}\sigma(s)A\sigma^T(s) &= \sum_{i,j=0}^{N-1} \sigma_{li}(s) A_{ij} \sigma_{mj}(s) \\ &= \sum_{i,j=0}^{N-1} \left(\delta_{l,s_i} \prod_{\alpha=N}^M (1 - \delta_{s_i, \alpha}) \right) A_{ij} \left(\delta_{m,s_j} \prod_{\beta=N}^M (1 - \delta_{s_j, \beta}) \right).\end{aligned}$$

Vers une réécriture sous forme de modèle d'Ising. Nous souhaitons traduire cette fonction de coût sous la forme d'un Hamiltonien de type Ising :

$$C(s) = \sum_{i,j} Q_{ij} z_i z_j + \sum_i b_i z_i, \quad \text{avec } z_i \in \{-1, 1\}.$$

Pour cela, il suffit de réécrire le symbole de Kronecker $\delta_{a,b}$ en fonction des variables binaires associées. En notant a_i, b_i les bits binaires, et $U = \lfloor \log_2 N \rfloor$, on a :

$$\begin{aligned}\delta_{a,b} &= \prod_{i=0}^{U-1} \delta_{a_i, b_i} \\ &= \prod_{i=0}^{U-1} (a_i + b_i - 1)^2 \\ &= \begin{cases} 1 & \text{si tous les } a_i = b_i, \\ 0 & \text{sinon.} \end{cases}\end{aligned}$$

Cette écriture permet, en principe, de développer chaque terme comme une combinaison quadratique de variables binaires. Ceci étant, on peut voir cette fonction de coût comme une somme et produit de fonctions de coûts beaucoup plus simples. Le Hamiltonien total est donc la somme et la composée des Hamiltoniens des sous fonctions de coûts, et on a une méthode en Qiskit pour le calculer. Ceci nous a permis de nous départir du problème qui nous avait bloqué au moment du rapport intermédiaire, à savoir l'incapacité d'exprimer **explicitement** la fonction de coût sous la forme de Ising.

RÉFÉRENCES

- [1] Frank Gaitan and Lane Clark. Graph isomorphism and adiabatic quantum computing. *Physical Review A*, 89(2), February 2014.
- [2] Michael R. Garey and David S. Johnson. *Computers and Intractability : A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., USA, 1979.
- [3] IBM. Error mitigation and suppression techniques, 2025.
- [4] Josh Schneider & Ian Smalley (IBM). Qu'est-ce que l'informatique quantique ?, 2024.
- [5] Richard M. Karp. *Reducibility among Combinatorial Problems*, pages 85–103. Springer US, Boston, MA, 1972.