Projet : résolution quantique de MAX XOR-SAT

Wagui Diallo, Théo Gounine, Simon Darnault, Simon Bélier

18 Janvier 2026

1 Description du problème et question préliminaire

Q1) Ecrire une classe MaxXorSat qui représente une entrée du problème.

Le code est disponible dans `code/basic_solver.py`**.**

```
class MaxXorSat:

def __init__(self, n  , m , A, b):
    try:
        # A taille nxm
        if (len(A) != n):
            raise ValueError("A n'a pas n lignes")
        if (len(A[0]) != m):
            raise ValueError("A n'a pas m colonnes")


        # b taille nx1
        if(len(b) != n):
            raise ValueError("b n'a pas n lignes")

        self.n = n
        self.m = m
        self.A = A
        self.b = b
    except Exception as e:
        print("Erreur dans la création de l'instance MaxXorSat:", e)
        exit(0)
```

Q2) Ecrire une fonction qui prend en entrée une instance du problème MaxXorSat et résout l'instance. La réponse de la fonction doit être la solution et son utilité.

Le code est disponible dans `code/basic_solver.py`**.**

```python
    #retourne la solution qui maximise et utilité = nombre d'égalité vérifié
def solve(entry: MaxXorSat):
    """
    Solveur Basique naif par énumération

    Args:
        entry: Instance de MaxXorSat

    Returns:
        (best_solution, max_utilite)

    """
    A = np.array(entry.A)
    b = np.array(entry.b)
    n = entry.n
    m = entry.m
    max_utilite = 0
    best_solution = None

    for bits in itertools.product([0, 1], repeat=m):
        bits = np.array(bits)

        # print(bits)
        res = np.dot(A, bits) % 2
        utilite = np.sum(res == b)
        if utilite > max_utilite:
            max_utilite = utilite
            best_solution = bits

    return (best_solution, max_utilite)


# Test
entry = MaxXorSat(2,2,[[1,1],[0,1]], [0,1])
solution = solve(entry)
print("solution: " , solution[0])
print("utlilité : " , solution[1])
```

Q3) Soit $\bigoplus$ l'opérateur XOR. Trouver un polynome de degré 2, utilisant les variables x1 et x2 égal à x1 $\oplus$ x2 .
On a :

$$x_1 \oplus x_2 = x_1 + x_2 - 2x_1x_2$$

Q4) Généralisez à $\bigoplus_{i=1}^{n} x_i$. Il faudra utiliser un polynome de degré n.

Pour construire le polynôme correspondant à $\bigoplus_{i=1}^{n} x_i$, on va représenter chaque x_i comme un bit égal à 0 ou 1.

On a doc un bit b de taille n correspondant à ceci :

- b = $x_1 x_2 ... x_n$

Soit $|b|$ le nombre de x_i valant 1.
Chaque bit b peut être codé par le produit suivant:

- $b = \prod_{si\ i\ dans\ b=1} x_i \prod_{si\ dans\ b=0} (1 - x_i)$

Ansi le polynôme p va être l'association de chaque bit b où —b— est impair (pour que le XOR vale 1, sinon le XOR vaut 0).

Ainsi on a : $P = \sum_{|b|\ impair} \prod_{i \in b} x_i \prod_{i \notin b} (1 - x_i)$

2 Résolution avec QAOA

Q5) Reformuler le problème de Max XOR SAT sous forme HOBO. Aidez vous de la question 4. Attention au cas où $b_i = 1$.

$$A = \begin{pmatrix} a_{1,1} & a_{1,2} & ... & a_{1,n} \\ ... & ... & ... & ... \\ a_{n,1} & a_{n,2} & ... & a_{m,n} \end{pmatrix} \text{ et } X = \begin{pmatrix} x_1 \\ ... \\ ... \\ x_n \end{pmatrix} \text{ et } b = \begin{pmatrix} b_1 \\ ... \\ ... \\ b_m \end{pmatrix}$$

$$\text{On a } AX = \begin{pmatrix} \sum_{i=1}^{n} a_{1,i} x_i \ mod(2) \\ ... \\ ... \\ \sum_{i=1}^{n} a_{1,i} x_i \ mod(2) \end{pmatrix} = \begin{pmatrix} \bigoplus_{i=1}^{n} a_{1,i} x_i \\ ... \\ ... \\ \bigoplus_{i=1}^{n} a_{m,i} x_i \end{pmatrix} = \begin{pmatrix} b_1 \\ ... \\ ... \\ b_m \end{pmatrix}$$

Donc $\forall l \in [1, m]$

- Si $b_l = 0$, on veut minimiser le polynôme
  $\bigoplus_{i=1}^{n} a_{l,i} x_i = \sum_{|b|\ impair} \prod_{i \in b} a_{l,i} x_i \prod_{i \notin b} (1 - a_{l,i} x_i)$
- Si $b_i = 1$, on veut minimiser le polynôme
  $1 - \bigoplus_{i=1}^{n} a_{l,i} x_i = 1 - \sum_{|b|\ impair} \prod_{i \in b} a_{l,i} x_i \prod_{i \notin b} (1 - a_{l,i} x_i)$

On veut donc au final minimiser le polynôme :

$$\sum_{i=1}^{m} \alpha_i \text{ avec } \begin{cases} \alpha_i = \sum\limits_{|b| \, impair} \prod\limits_{i \in b} a_{l,i} x_i \prod\limits_{i \notin b} (1 - a_{l,i} x_i) & si \, b_i = 0 \\ \alpha_i = 1 - \sum\limits_{|b| \, impair} \prod\limits_{i \in b} a_{l,i} x_i \prod\limits_{i \notin b} (1 - a_{l,i} x_i) & si \, b_i = 1 \end{cases}$$

La somme des α_i va nous donner exactement le nombre d'égalités non satisfaites.

Q6) Effectuer un changement de variable pour remplacer les variables binaires par des variables $-1, 1$.
On souhaite avoir :

- $x_i = 1 \Leftrightarrow z_i = -1$
- $x_i = 0 \Leftrightarrow z_i = 1$

On pose $z_i = 1 - 2x_i \Leftrightarrow x_i = \frac{1 - z_i}{2}$

On a donc :

$$P = \sum_{|b| \, impair} \prod_{i \in b} \frac{1 - z_i}{2} \prod_{i \notin b} (1 - \frac{1 - z_i}{2})$$
$$P = \frac{1}{2^n} \sum_{|b| \, impair} \prod_{i \in b} (1 - z_i) \prod_{i \notin b} (1 + z_i)$$

La somme se simplifie (somme télescopique)

$$P = \frac{1}{2}(1 - \prod_{i=1}^{n} z_i) \text{ et } 1 - P = 1 - \frac{1}{2}(1 - \prod_{i=1}^{n} z_i) = 1 + \prod_{i=1}^{n} z_i$$

Le problème devient donc de minimiser le polynôme :

$$\frac{1}{2} \sum_{i=1}^{n} \alpha_i \text{ où } \begin{cases} \alpha_i = 1 - \prod\limits_{i=1}^{n} z_i & si \, b_i = 0 \\ \alpha_i = 1 + \prod\limits_{i=1}^{n} z_i & si \, b_i = 1 \end{cases}$$

Q7) Écrire l'Hamiltonien H_C associé, montrer sur un exemple avec 3 variables et 2 contraintes de votre choix qu'une solution est bien associée à un vecteur propre de valeur propre γ égale à l'opposé du nombre de contraintes non satisfaites.
On a donc l'Hamiltonien

$$H_c = \frac{1}{2} \sum_{i=1}^{n} (1 + (-1)^{1 - b_i} \bigotimes_{k=1}^{n} Z_k)$$

Exemple avec 3 variables et 2 contraintes:

$$Hc = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \end{pmatrix}, et \, b = \begin{pmatrix} 1 \\ 0 \end{pmatrix}$$

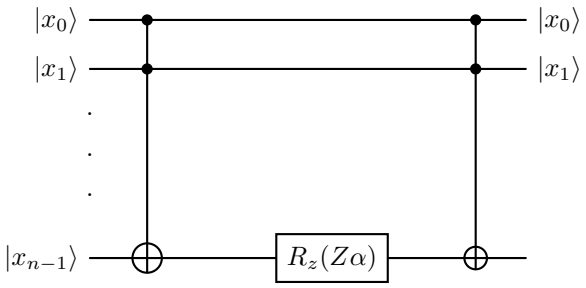
Ici on a 1 solution qui ne peut pas être satisfaite.
On a alors $H_c = \frac{1}{2} \sum_{i=1}^{n} (1 + (-1)^{1 - b_i} \bigotimes_{k=1}^{n} Z_k)$

$$H_c = \frac{1}{2}(I_4 + Z_1 \bigotimes Z_2 + I4 - Z_1 \bigotimes Z_2) \, et \, Z_1 \bigotimes Z_2 = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & -1 & 0 & 0 \\ 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

$$H_c = \frac{1}{2}(\begin{pmatrix} 2 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 2 \end{pmatrix} + \begin{pmatrix} 0 & 0 & 0 & 0 \\ 0 & 2 & 0 & 0 \\ 0 & 0 & 2 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix}) = I_4$$

Q8) Décrire le circuit quantique de la matrice unitaire $\exp i\gamma H_C$**.**

Les termes de H_c commutent entre eux car diagonaux. D'après ce qui a pu être observé en cours, le circuit associé à $\exp i\gamma H_C$ est le suivant:



Avec $R_Z(Z\alpha) = \begin{pmatrix} exp(i\alpha) & 0 \\ 0 & exp(i\alpha) \end{pmatrix}$

3 Résolution avec l'algorithme de Grover

3.1 Résolution du problème de décision associé

Q10) Écrire le problème de décision associé au problème Max XOR SAT.

Le problème de décision associé à MAX XOR SAT est définit comme ceci :

- Entrée : matrice 1, vecteur b et entier k (seuil)
- Question : Existe-t-il $x \in \{0,1\}^n$ tel que le nombre d

Le nombre d'égalité dans Ax = b est supérieur ou égal à k.

Q11) Proposer schématiquement la construction de l'oracle du problème de décision, en précisant le rôle de chaque partie de l'oracle.

Schéma O_f : Calcul (1) → Test du seuil (2) → Uncompute (3)

(1) : Pour chaque ligne de A, on calcule la parité des variables. On compare la parité à b_j

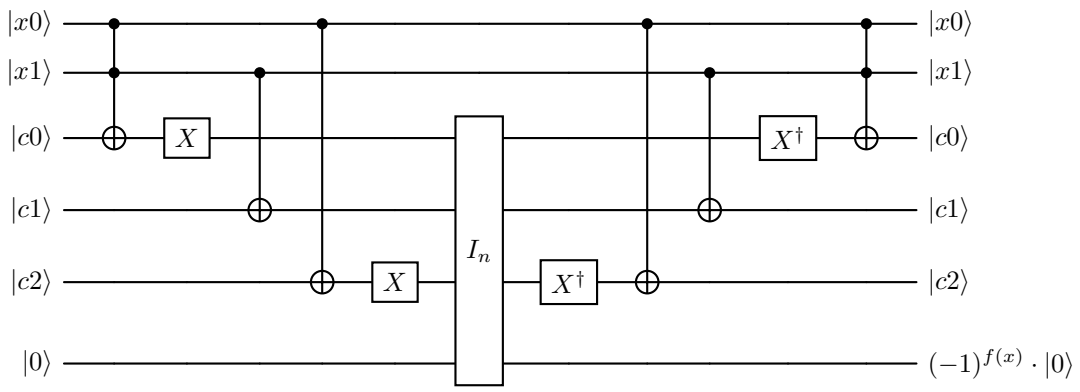
On stocke dans un qbit ancillaire c_j. Si $c_j = 1$ équation j est satisfaite, 0 sinon.

(2) : Test du seuil, on prend en entrée les c_j et avec une addition on teste si $\sum c_j \geq k$

(3) : Miroir partie 1

Q12) Proposer un circuit pour l'instance suivante et la valeur k = 2. Vous pouvez supposer que vous disposer d'une porte In qui indique si au moins k qbits parmi ses entrées sont égales à 1 (le nombre d'entrées, et la description des sorties de cette porte peuvent être indiquées librement dans votre réponse).

$$A = \begin{pmatrix} 1 & 1 \\ 0 & 1 \\ 1 & 0 \end{pmatrix}, \; b = \begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix}$$



Nous considérons que la porte I_n prend en entrée 3 qbits $|c0\rangle$, $|c1\rangle$, $|0\rangle$ et change la phase si $\sum_j c_j \geq k$. Ainsi en sortie de l'oracle, la phase est changée si le problème de décision est vérifié. Nous considérons donc f tel que:

$$f\colon \{0,1\}^n \to \{0,1\}$$
$$x \mapsto \begin{cases} 1 & \text{si } \textit{au moins k qbits parmis ses entrées sont égales à 1} \\ 0 & \text{sinon} \end{cases}$$

Q13) Pour ne pas exploser le nombre de qbits avec les qbits de travail dans qiskit en construisant la porte I_n, on va tricher un peu. Soit une liste L de p nombres entre 0 et $2n-1$, expliquer comment construire un circuit de profondeur $O(|L|)$ qui, connaissant un qbit de base $|\underline{x}\rangle$ renvoie $-|\underline{x}\rangle$ si $x \in L$ et renvoie $|\underline{x}\rangle$ sinon.

Pour obtenir un tel circuit, on peut poser f la fonction suivante:

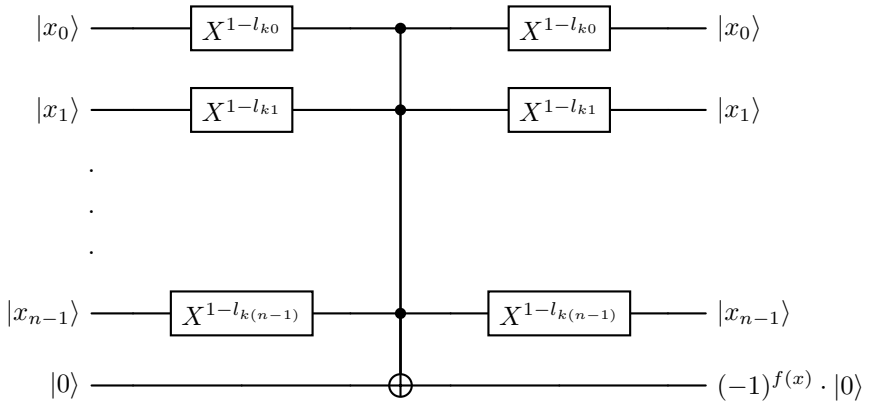
$$f\colon \{0,1\}^n \to \{0,1\}$$
$$x \mapsto \begin{cases} 0 & \text{si } x \in L \\ 1 & \text{si } x \notin L \end{cases}$$

On effectue une hypothèse: les éléments de L sont 2 à 2 distincts !.

Pour chaque élément de la liste L, on va réaliser ce circuit:
Soit l_k le k-ème élément de L, il peut s'écrire sous la forme binaire $\left|\underline{l_k}\right\rangle = |l_{k1}\rangle\,|l_{k2}\rangle\,...\,|l_{kn}\rangle$.

Pour le circuit on va utiliser la porte : $\boxed{X^{1-l_{ki}}}$ qui va agir comme une porte X si le qbit $|l_k i\rangle$ vaut $|0\rangle$ et ne va rien faire si le qbit $|l_k i\rangle$ vaut $|1\rangle$.

Pour chaque $\left|\underline{l_k}\right\rangle$, on va appliquer le circuit suivant à $|\underline{x}\rangle$:



L'idée est d'appliquer une porte X sur chaque bit de la à 0 et de faire une CC...CNOT sur $|\underline{x}\rangle$ à la fin.
Ce circuit est de profondeur 2.
En l'appliquant sur chaque élément de la liste L on a un circuit de profondeur en $\bigcirc(Z|L|) = \bigcirc(|L|)$.

Q14) Implémenter ensuite une fonction qui commence par construire L la liste de toutes les solutions réalisables dont le poids est supérieur à k. Utiliser ensuite cette fonction pour construire un oracle.

Le code est disponible dans `code/grover_solver.py`.

```
def build_realizable_solutions(entry: MaxXorSat, k: int):
n = entry.n
m = entry.m
A = entry.A
b = entry.b

realizable_solutions = []
for bits in itertools.product([0, 1], repeat=m):
    bits = np.array(bits)
    res = np.dot(A, bits) % 2
    utilite = np.sum(res == b)
    if utilite >= k:
        realizable_solutions.append(bits)
return realizable_solutions
```

Q15) Expliquer pourquoi concevoir ce circuit est en contradiction avec l'utilisation de l'algorithme de Grover.

Normalement, l'algorithme de Grover est utilisé pour trouver ces solutions sans les connaître. Ici, on utilise Grover pour amplifier des solutions qu'on a déjà trouvées classiquement (juste pour l'exercice d'implémentation de l'algorithme).

Q16) Codez (malgré la contradiction) avec qiskit un circuit qui implante l'algorithme de Grover pour résoudre le problème de décision associé au problème max XOR SAT.

3.2 Résolution du problème d'optimisation

Q17) Proposer de façon schématique le fonctionnement de l'algorithme d'optimisation.

L'algorithme d'optimisation a pour objectif de renvoyer une liste de x_i qui satisfassent un maximum de clauses tandis que le problème de décision renvoie seulement si le nombre de clauses satisfaites est supérieure où égale à une constante K donné.

On pourrait donc obtenir l'optimisation en cherchant le K maximum en utilisant le problème de décision de manière dichotomique. Puis une fois K trouvé, on applique Grover pour obtenir cette solution:

Schéma:

$$K_0 = \lfloor \tfrac{n}{2} \rfloor \begin{cases} 0 & \to K_1 = \lfloor \tfrac{n}{2} \rfloor - \lfloor \tfrac{n}{4} \rfloor \begin{cases} 0 & \to K_2 = \lfloor \tfrac{n}{2} \rfloor - \lfloor \tfrac{n}{4} \rfloor - \lfloor \tfrac{n}{8} \rfloor \begin{cases} 0 & \dots \\ 1 & \dots \end{cases} \\ & K_2 \neq K_1 ? \\ 1 & \to K_2 = \lfloor \tfrac{n}{2} \rfloor + \lfloor \tfrac{n}{4} \rfloor + \lfloor \tfrac{n}{8} \rfloor \begin{cases} 0 & \dots \\ 1 & \dots \end{cases} \\ & K_2 \neq K_1 ? \end{cases} \\ & K_1 \neq K_0 ? \\ 1 & \to K_1 = \lfloor \tfrac{n}{2} \rfloor - \lfloor \tfrac{n}{4} \rfloor \begin{cases} 0 & \dots \\ 1 & \dots \end{cases} \\ & K_1 \neq K_0 ? \end{cases}$$

Une fois K trouvé, on applique Grover pour le k trouvé et on renvoie le vecteur x associé.

Q18) Implémenter cet algorithme avec Qiskit.
Le code est disponible dans `code/grover_solver.py`.

```python
def determine_opti_k(entry: MaxXorSat):
    """
    Détermine une valeur k optimale pour Grover par dichotomie.
    """
    # Dichotomie sur k entre 0 et n (nb max de contraintes satisfaisables)
    low = 0
    high = entry.n
    best_k = 0

    while low <= high:
        mid = (low + high) // 2
        # On vérifie s'il existe au moins une solution avec utilité >= mid
        solutions = build_realizable_solutions(entry, mid)
        if len(solutions) > 0:
            best_k = mid
            low = mid + 1
        else:
            high = mid - 1

    return best_k
```

4 Evaluation

Proposer une évaluation de la qualité des 2 algorithmes, QAOA et Grover. Cet algorithme évaluera:

- le temps de calcul
- la qualité des solutions renvoyées, utilisez pour cela l'algorithme exact codé au début du projet.
- la complexité des algorithmes (nombre de porte, profondeur des circuits, nombre de qbits, nombre de répétition des algorithmes quantiques).

Le code est disponible dans `code/evaluate.py`.

4.1 Génération des instances

L'algorithme génère un jeu de données synthétique composé de N instances aléatoires. Chaque instance est définie par :

- n : le nombre de clauses
- m : le nombre de variables booléennes.

Les matrices A et les vecteurs b sont générés uniformément de manière aléatoire binaire. Les dimensions n et m varient dans une plage définie.

4.2 Exécution des algorithmes

Pour chaque instance générée, trois solveurs sont exécutés séquentiellement :

- **Solveur Exact (Référence) :** Une approche par force brute énumère toutes les 2^m combinaisons possibles pour identifier la solution optimale globale. Cette étape sert de base de comparaison.
- **QAOA :** Le circuit est construit avec une profondeur p (nombre de répétitions) paramétrable. Une boucle d'optimisation classique (utilisant l'optimiseur COBYLA) . Le circuit final est ensuite mesuré pour extraire l'état le plus probable.
- **Algorithme de Grover :** L'algorithme détermine d'abord un seuil d'utilité k optimal (via une recherche dichotomique classique sur l'oracle simulé). Le circuit est ensuite instancié avec un nombre d'itérations déterminé plus tôt, puis mesuré.

4.3 Métriques de comparaison

Pour chaque exécution, les indicateurs suivants sont collectés :

- **Qualité de la solution (Utility Ratio) :** Il s'agit du rapport entre l'utilité de la solution trouvée par l'algorithme quantique et l'utilité optimale trouvée par le solveur exact. Un ratio de 1.0 indique une solution optimale.
$$\text{Ratio} = \frac{\text{Utilité}_{\text{Quantique}}}{\text{Utilité}_{\text{Exacte}}}$$
- **Temps de calcul (Time-to-Solution) :** Le temps d'exécution total, incluant pour le cas quantique la construction du circuit, la compilation (transpilation) et la simulation (*sampling*).
- **Complexité du Circuit (Ressources Quantiques) :**
  - **Profondeur (Depth) :** Le nombre de couches de portes quantiques, indicateur critique pour la faisabilité sur des machines NISQ (*Noisy Intermediate-Scale Quantum*) à temps de cohérence limité.
  - **Volume de portes (Gate Count) :** Le nombre total d'opérations quantiques nécessaires.
  - **Nombre de Qubits :** La largeur du circuit requise.

4.4 Analyse des résultats

Les données brutes sont stockées dans un tableau (*DataFrame*) et exportées au format CSV. Des visualisations graphiques sont générées automatiquement pour illustrer l'évolution du temps de calcul et des ratios de performance en fonction de la taille du problème (n et m), permettant d'identifier les avantages et les limites de chaque approche.

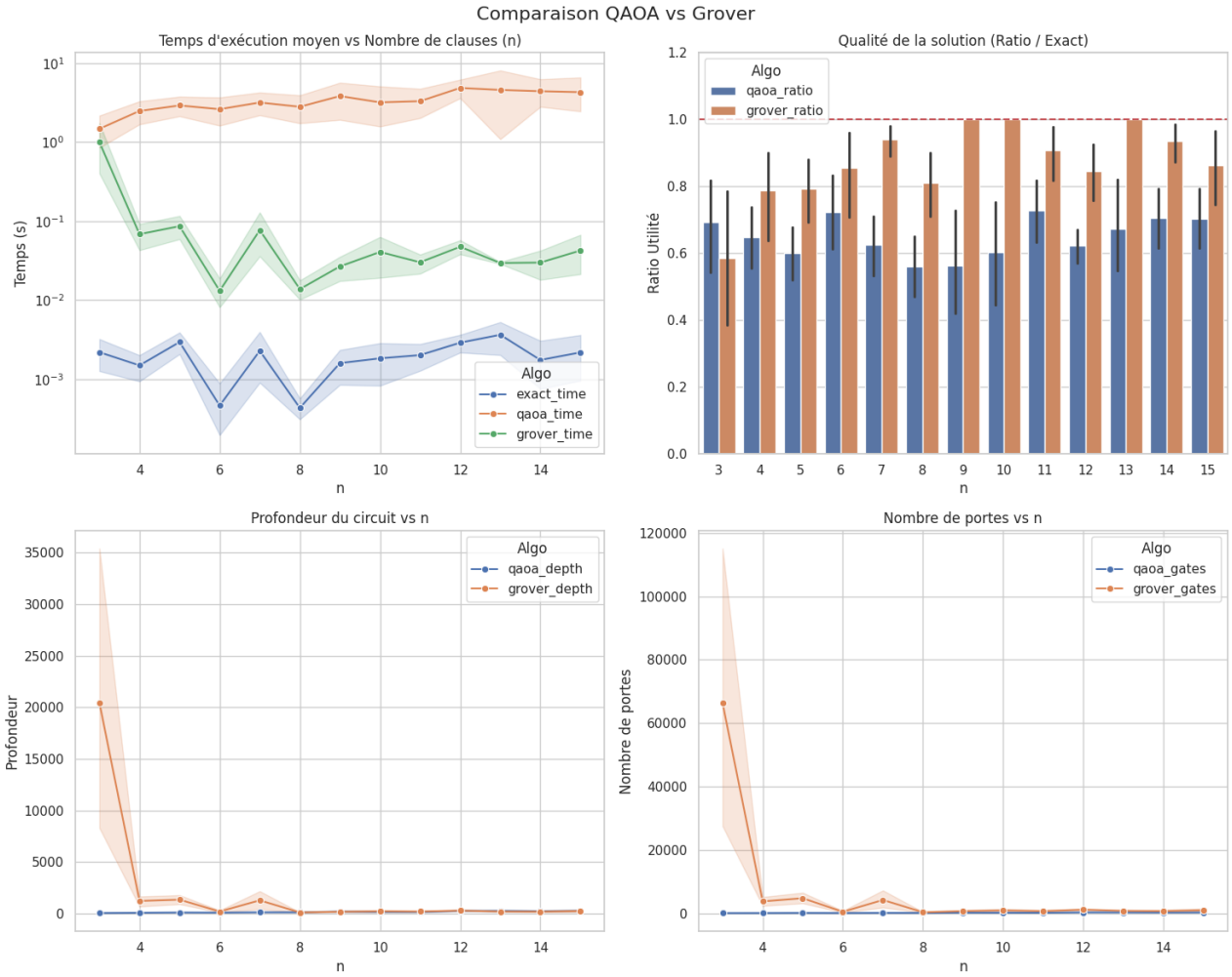


Figure 1: Résultat du benchmark pour 50 instances, $n \in [\![3, 15]\!]$, $m \in [\![3, 10]\!]$, 50 instances, $reps\_qaoa \in [\![1, 5]\!]$