Projet : Optimisation de problèmes pseudo-booléen quadratique (QUBO)

Résolution de problèmes d'optimisation

Tableau des versions

Version	Date	Auteur
1.0	02-04-2025	Fellah Mohammed Nassim
2.0	20-04-2025	Fellah Mohammed Nassim

Table des matières

1. Introduction **1**

2. Objectifs du projet et hypothèses initiales **1**

3. Méthodologie et Conception des Algorithmes **2**
3.1. Vue d'ensemble du Framework 2
3.2. Mécanisme d'Évaluation Incrémentale 2
3.2.1. Dérivation de la Formule Delta Fitness 3
3.2.2. Comparaison des Complexités 3
3.2.3. Preuve de la formule pour $\Delta^{(2)}_{u,v} f(x)$ 4

4. Algorithmes implémentés **7**
4.1. Recherche Aléatoire 7
4.2. Recherche Locale Itérée (ILS) 7
4.3. Recherche Tabou (TS) 8
4.4. Recuit Simulé (SA) 8
4.5. Algorithme Génétique (GA) 9
4.6. Descente de Gradient (GD) 9
4.7. HybridSAGDOptimiser (Hybride SA et GD) 10

5. Méthodologie de test **10**

6. Configuration expérimentale **10**
6.1. Scope des tests 10
6.2. Analyse comparative 11

7. Démonstrations et validations des temps d'exécution **11**
7.1. Comportement de convergence 11
7.2. Variation du temps d'exécution 11
7.3. Analyse des résultats (voir rapports) 12

8. Améliorations futures **12**

1. Introduction

Les problèmes de Quadratic Unconstrained Binary Optimization (QUBO) visent à minimiser une fonction quadratique sur des variables binaires ($x_i \in \{0, 1\}$), formulée comme suit :

$$f(x) = \sum_{i=1}^{n} h_i(-1)^{x_i} + \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} q_{ij}(-1)^{x_i}(-1)^{x_j}$$

Ici, h_i sont des coefficients linéaires, q_{ij} sont des termes d'interaction quadratique, et $(-1)^{x_i}$ associe $x_i = 0$ à 1 et $x_i = 1$ à -1. En tant que problèmes NP-difficiles, les instances QUBO représentent un défi pour les méthodes computationnelles, mais leur capacité à modéliser des applications comme l'optimisation de graphes, l'apprentissage automatique et les entrées en informatique quantique les rend particulièrement pertinentes.

Ce projet a été motivé par le besoin de comprendre et de comparer les techniques d'optimisation classiques pour QUBO, en fournissant une base pour l'évaluation par rapport aux solveurs quantiques émergents et en explorant les approches hybrides classiques-numériques. L'objectif était de développer des solveurs pratiques et efficaces et de comprendre leurs caractéristiques de performance.

2. Objectifs du projet et hypothèses initiales

L'objectif principal était de développer et d'évaluer une suite d'algorithmes d'optimisation (recherche aléatoire, recherche locale itérée (ILS), recherche tabou, descente de gradient (GD), recuit simulé (SA) et une combinaison de SA et de GD (HybridSAGD)) pour résoudre des problèmes QUBO dans un délai calibré de 30 secondes par exécution. Les objectifs spécifiques comprenaient :

- Obtenir des valeurs de fitness proches ou supérieures aux meilleures solutions connues (par exemple, -70459 pour l'instance 990).
- Évaluer l'évolutivité, la cohérence et l'efficacité de chaque algorithme sur les instances 990 à 999.
- Comparer l'efficacité des méthodes combinatoires et numériques.
- Étudier si une approche hybride pourrait surpasser les techniques autonomes.

Hypothèses initiales :

- **Recherche Aléatoire** : On s'attend à ce qu'elle soit la moins performante en raison de son manque de structure.
- **ILS** : On s'attend une amélioration par rapport à la Recherche Aléatoire grâce à une optimisation locale et une perturbation.

- **TS** : On s'attend à ce qu'elle surpasse ILS grâce à sa diversification guidée par la mémoire.
- **SA** : On suppose qu'elle excelle, potentiellement la meilleure, grâce à ses échappées probabilistes des optima locaux.
- **GD** : On s'attend à une convergence rapide mais à des solutions binaires sous-optimales en raison de la projection.
- **HybridSAGD** : On suppose qu'elle pourrait surpasser toutes les autres en combinant la recherche globale de SA avec l'affinage de GD.

J'ai supposé que les méthodes combinatoires excelleraient généralement dans le domaine QUBO discret, mais les approches numériques et hybrides pourraient offrir des avantages uniques si elles sont correctement intégrées.

3. Méthodologie et Conception des Algorithmes

3.1. Vue d'ensemble du Framework

Un framework modulaire en Java a été développé pour permettre le prototypage rapide et le benchmarking :

- **QUBOEval** : Contient les coefficients h_i et q_{ij}, avec les méthodes `evaluate(x)` pour l'évaluation binaire et `fnumerical(z)` pour l'évaluation continue.
- **Optimiser** : Classe abstraite avec la méthode `optimize()` retournant un `Result` (solution x, fitness), étendue par tous les algorithmes.
- **Calibration** : Imposait une limite de 30 secondes calbirée (`TIME_LIMIT_MILLIS = 36790`).
- **Main** : Exécutait les optimiseurs sur les instances 990–999, enregistrait les résultats dans des fichiers CSV.

3.2. Mécanisme d'Évaluation Incrémentale

Pour améliorer l'efficacité dans les algorithmes combinatoires, une mise à jour incrémentale du score a été implémentée via la classe LocalSearchHelper:

- **Delta Fitness** :

$$\Delta_u f(x) = -2(-1)^{x_u} \left(h_u + \sum_{i<u} q_{iu}(-1)^{x_i} + \sum_{j>u} q_{uj}(-1)^{x_j} \right)$$

  calcule le changement de score lors du flip du bit x_u.

- **Delta de Second Ordre** :

$$\Delta^{(2)}_{u,v} f(x) = 4q_{uv}(-1)^{x_u}(-1)^{x_v} \quad (u \neq v)$$

permet de mettre à jour les deltas des voisins après un flip, réduisant la complexité de $O(n)$ à $O(d_u)$, où d_u est le degré du nœud u.

- **Utilitaire** : Cette logique a été encapsulée dans une classe utilitaire `LocalSearchHelper`, qui a considérablement accéléré les algorithmes ILS, Tabou, SA, et l'hybride.

3.2.1. Dérivation de la Formule Delta Fitness

Étant donné :

$$f(x) = \sum_{i=1}^{n} h_i(-1)^{x_i} + \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} q_{ij}(-1)^{x_i}(-1)^{x_j}$$

Lorsque l'on effectue un flip de x_u, on définit $x' = x \oplus u$, où $x'_i = x_i$ si $i \neq u$, et $x'_u = 1 - x_u$.

- **Termes linéaires** :

$$Ancien : \sum_{i=1}^{n} h_i(-1)^{x_i}, \quad Nouveau : \sum_{i \neq u} h_i(-1)^{x_i} + h_u(-1)^{1-x_u}$$

Comme $(-1)^{1-x_u} = -(-1)^{x_u}$, le changement est :

$$-2h_u(-1)^{x_u}$$

- **Termes quadratiques** : Les termes où ni i ni j ne sont égaux à u restent inchangés.
  - Pour $i = u, j > u$:

$$q_{uj}(-1)^{1-x_u}(-1)^{x_j} = -q_{uj}(-1)^{x_u}(-1)^{x_j} \quad \Rightarrow Changement = -2q_{uj}(-1)^{x_u}(-1)^{x_j}$$

  - Pour $j = u, i < u$:

$$q_{iu}(-1)^{x_i}(-1)^{1-x_u} = -q_{iu}(-1)^{x_i}(-1)^{x_u} \quad \Rightarrow Changement = -2q_{iu}(-1)^{x_i}(-1)^{x_u}$$

- **Changement total** :

$$\Delta_u f(x) = -2(-1)^{x_u} \left(h_u + \sum_{i=1}^{u-1} q_{iu}(-1)^{x_i} + \sum_{j=u+1}^{n} q_{uj}(-1)^{x_j} \right)$$

3.2.2. Comparaison des Complexités

- **Évaluation directe** de $f(x \oplus u)$ via `evaluate` : Complexité $O(n + m)$, avec m le nombre de q_{ij} non nuls (jusqu'à $O(n^2)$ si dense).
- **Delta Fitness** via `computeDeltaFitness` : Complexité $O(d_u)$, où d_u est le degré de u (nombre de q_{ij} non nuls impliquant u).

Conclusion : L'évaluation incrémentale est beaucoup plus efficace, surtout dans le cas d'un matrice Q creuse, passant de $O(n^2)$ à $O(d_u)$.

3.2.3. Preuve de la formule pour $\Delta^{(2)}_{u,v} f(x)$

La fonction de fitness QUBO est donnée par :

$$f(x) = \sum_{i=1}^{n} h_i (-1)^{x_i} + \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} q_{ij} (-1)^{x_i} (-1)^{x_j}$$

où $x_i \in \{0, 1\}$, et $(-1)^{x_i} = 1$ si $x_i = 0$, ou -1 si $x_i = 1$.
D'après la section 2.2.a, la variation du premier ordre $\Delta_u f(x) = f(x \oplus u) - f(x)$, où $x \oplus u$ inverse le u-ième bit de x, est :

$$\Delta_u f(x) = -2(-1)^{x_u} \left(\sum_{i=1}^{u-1} q_{iu} (-1)^{x_i} + h_u + \sum_{j=u+1}^{n} q_{uj} (-1)^{x_j} \right)$$

La variation du second ordre est définie comme :

$$\Delta^{(2)}_{u,v} f(x) = \Delta_u f(x \oplus v) - \Delta_u f(x)$$

Cela représente la différence de la variation du premier ordre $\Delta_u f$ lorsque le v-ième bit est inversé. Nous devons calculer $\Delta_u f(x \oplus v)$, qui est la variation de f en inversant le u-ième bit dans la solution $x \oplus v$, et soustraire $\Delta_u f(x)$.

Étape 1 : Calcul de $\Delta_u f(x \oplus v)$ La solution $x \oplus v$ est identique à x, sauf que le v-ième bit est inversé, donc :

- $(x \oplus v)_i = x_i$ pour tout $i \neq v$,
- $(x \oplus v)_v = 1 - x_v$, donc $(-1)^{(x \oplus v)_v} = (-1)^{1 - x_v} = -(-1)^{x_v}$.

En utilisant la formule de $\Delta_u f$, nous calculons :

$$\Delta_u f(x \oplus v) = -2(-1)^{(x \oplus v)_u} \left(\sum_{i=1}^{u-1} q_{iu} (-1)^{(x \oplus v)_i} + h_u + \sum_{j=u+1}^{n} q_{uj} (-1)^{(x \oplus v)_j} \right)$$

Nous devons ajuster les termes en fonction de $u = v$, $u < v$, ou $v < u$.

Cas 1 : $u = v$ Si $u = v$, alors $x \oplus v$ a le u-ième bit inversé, donc $(x \oplus v)_u = 1 - x_u$, et :

$$(-1)^{(x \oplus v)_u} = (-1)^{1 - x_u} = -(-1)^{x_u}$$

Pour la somme, les indices $i < u$ et $j > u$ n'incluent pas $v = u$, donc $(x \oplus v)_i = x_i$ et $(x \oplus v)_j = x_j$. Ainsi, la somme interne est identique à celle de $\Delta_u f(x)$:

$$\sum_{i=1}^{u-1} q_{iu} (-1)^{(x \oplus v)_i} + h_u + \sum_{j=u+1}^{n} q_{uj} (-1)^{(x \oplus v)_j} = \sum_{i=1}^{u-1} q_{iu} (-1)^{x_i} + h_u + \sum_{j=u+1}^{n} q_{uj} (-1)^{x_j}$$

Désignons cette somme par :

$$S_u = \sum_{i=1}^{u-1} q_{iu}(-1)^{x_i} + h_u + \sum_{j=u+1}^{n} q_{uj}(-1)^{x_j}$$

Donc :

$$\Delta_u f(x \oplus v) = -2(-(-1)^{x_u})S_u = 2(-1)^{x_u} S_u$$

Maintenant, calculons $\Delta_u f(x)$:

$$\Delta_u f(x) = -2(-1)^{x_u} S_u$$

Ainsi :

$$\Delta^{(2)}_{u,u} f(x) = \Delta_u f(x \oplus u) - \Delta_u f(x) = 2(-1)^{x_u} S_u - (-2(-1)^{x_u} S_u) = 2(-1)^{x_u} S_u + 2(-1)^{x_u} S_u = 4(-1)^{x_u} S_u$$

Puisque $\Delta_u f(x) = -2(-1)^{x_u} S_u$, nous avons :

$$4(-1)^{x_u} S_u = 2 \cdot (-2(-1)^{x_u} S_u) = 2\Delta_u f(x)$$

Donc, pour $u = v$:

$$\Delta^{(2)}_{u,u} f(x) = 2\Delta_u f(x)$$

Cela correspond à la formule pour $u = v$.

Cas 2 : $u < v$ Si $u < v$, alors $(x \oplus v)_u = x_u$, donc $(-1)^{(x \oplus v)_u} = (-1)^{x_u}$. La somme interne nécessite un ajustement uniquement pour les termes impliquant v. Les termes sont :

$$\sum_{i=1}^{u-1} q_{iu}(-1)^{(x \oplus v)_i} + h_u + \sum_{j=u+1}^{n} q_{uj}(-1)^{(x \oplus v)_j}$$

- Pour $i < u$, comme $u < v$, $i \neq v$, donc $(x \oplus v)_i = x_i$, et $\sum_{i=1}^{u-1} q_{iu}(-1)^{(x \oplus v)_i} = \sum_{i=1}^{u-1} q_{iu}(-1)^{x_i}$.
- Le terme h_u reste inchangé.
- Pour $j > u$, nous devons considérer $j = v$. La somme est :

$$\sum_{j=u+1}^{n} q_{uj}(-1)^{(x \oplus v)_j} = q_{uv}(-1)^{(x \oplus v)_v} + \sum_{j=u+1, j \neq v}^{n} q_{uj}(-1)^{(x \oplus v)_j}$$

Puisque $(x \oplus v)_v = 1 - x_v$, nous avons $(-1)^{(x \oplus v)_v} = -(-1)^{x_v}$. Pour $j \neq v$, $(x \oplus v)_j = x_j$. Donc :

$$\sum_{j=u+1}^{n} q_{uj}(-1)^{(x \oplus v)_j} = q_{uv}\left(-(-1)^{x_v}\right) + \sum_{j=u+1, j \neq v}^{n} q_{uj}(-1)^{x_j} = -q_{uv}(-1)^{x_v} + \sum_{j=u+1, j \neq v}^{n} q_{uj}(-1)^{x_j}$$

Comparons cela à la somme originale dans $\Delta_u f(x)$:

$$\sum_{j=u+1}^{n} q_{uj}(-1)^{x_j} = q_{uv}(-1)^{x_v} + \sum_{j=u+1, j\neq v}^{n} q_{uj}(-1)^{x_j}$$

La différence est :

$$\left(-q_{uv}(-1)^{x_v} + \sum_{j=u+1, j\neq v}^{n} q_{uj}(-1)^{x_j}\right) - \sum_{j=u+1}^{n} q_{uj}(-1)^{x_j} = -q_{uv}(-1)^{x_v} - q_{uv}(-1)^{x_v} = -2q_{uv}(-1)^{x_v}$$

Ainsi, la somme interne pour $\Delta_u f(x \oplus v)$:

$$\sum_{i=1}^{u-1} q_{iu}(-1)^{x_i} + h_u + \sum_{j=u+1}^{n} q_{uj}(-1)^{(x\oplus v)_j} = \left(\sum_{i=1}^{u-1} q_{iu}(-1)^{x_i} + h_u + \sum_{j=u+1}^{n} q_{uj}(-1)^{x_j}\right) - 2q_{uv}(-1)^{x_v}$$

Donc :

$$\Delta_u f(x \oplus v) = -2(-1)^{x_u}\left(S_u - 2q_{uv}(-1)^{x_v}\right)$$

Maintenant, calculons :

$$\Delta^{(2)}_{u,v} f(x) = \Delta_u f(x \oplus v) - \Delta_u f(x) = -2(-1)^{x_u}\left(S_u - 2q_{uv}(-1)^{x_v}\right) - \left(-2(-1)^{x_u} S_u\right)$$

$$= -2(-1)^{x_u} S_u + 4(-1)^{x_u} q_{uv}(-1)^{x_v} + 2(-1)^{x_u} S_u = 4(-1)^{x_u} q_{uv}(-1)^{x_v}$$

Ainsi, pour $u < v$:

$$\Delta^{(2)}_{u,v} f(x) = 4q_{uv}(-1)^{x_u}(-1)^{x_v}$$

Cela correspond à la formule.

Cas 3 : $v < u$ Si $v < u$, alors $(x \oplus v)_u = x_u$, donc $(-1)^{(x\oplus v)_u} = (-1)^{x_u}$. Maintenant, v apparaît dans la première somme $i < u$:

$$\sum_{i=1}^{u-1} q_{iu}(-1)^{(x\oplus v)_i} = q_{vu}(-1)^{(x\oplus v)_v} + \sum_{i=1, i\neq v}^{u-1} q_{iu}(-1)^{(x\oplus v)_i}$$

Puisque $(x \oplus v)_v = 1 - x_v$, $(-1)^{(x\oplus v)_v} = -(-1)^{x_v}$, et pour $i \neq v$, $(x \oplus v)_i = x_i$. Donc :

$$\sum_{i=1}^{u-1} q_{iu}(-1)^{(x\oplus v)_i} = q_{vu}\left(-(-1)^{x_v}\right) + \sum_{i=1, i\neq v}^{u-1} q_{iu}(-1)^{x_i} = -q_{vu}(-1)^{x_v} + \sum_{i=1, i\neq v}^{u-1} q_{iu}(-1)^{x_i}$$

La somme originale est :

$$\sum_{i=1}^{u-1} q_{iu}(-1)^{x_i} = q_{vu}(-1)^{x_v} + \sum_{i=1, i\neq v}^{u-1} q_{iu}(-1)^{x_i}$$

La différence est :

$$\left(-q_{vu}(-1)^{x_v} + \sum_{i=1, i\neq v}^{u-1} q_{iu}(-1)^{x_i}\right) - \sum_{i=1}^{u-1} q_{iu}(-1)^{x_i} = -q_{vu}(-1)^{x_v} - q_{vu}(-1)^{x_v} = -2q_{vu}(-1)^{x_v}$$

Pour $j > u$, comme $v < u$, $j \neq v$, donc la seconde somme reste inchangée. Ainsi :

$$\Delta_u f(x \oplus v) = -2(-1)^{x_u} \left(\left(\sum_{i=1}^{u-1} q_{iu}(-1)^{x_i} + h_u + \sum_{j=u+1}^{n} q_{uj}(-1)^{x_j} \right) - 2q_{vu}(-1)^{x_v} \right)$$
$$= -2(-1)^{x_u} \left(S_u - 2q_{vu}(-1)^{x_v} \right)$$

Donc :

$$\Delta^{(2)}_{u,v} f(x) = -2(-1)^{x_u} \left(S_u - 2q_{vu}(-1)^{x_v} \right) - \left(-2(-1)^{x_u} S_u \right) = 4(-1)^{x_u} q_{vu}(-1)^{x_v}$$

Ainsi, pour $v < u$:

$$\Delta^{(2)}_{u,v} f(x) = 4q_{vu}(-1)^{x_u}(-1)^{x_v}$$

Cela correspond à la formule.

4. Algorithmes implémentés

4.1. Recherche Aléatoire

Logique de conception : Génère et évalue des solutions binaires aléatoires, en conservant la meilleure. C'est une base simple.

- **Attentes** : Mauvaise performance — haute aptitude et variance — en raison de l'absence de guidance.
- **Implémentation** : Solutions échantillonnées de manière modulaire et évaluées, avec enregistrement au format CSV.
- **Observations** : Des mauvaises valeurs, c'est un baseline, pour explorer l'espace de recherche et les solutions.
- **Forces et faiblesses** : Simple et rapide par itération, mais inefficace pour la complexité de QUBO.
- **Complexité** : $O(k \cdot (n+m))$, où k est le nombre d'itérations (proportionnel au temps imparti), n est le nombre de variables et m est le nombre de termes q_{ij} non nuls.

4.2. Recherche Locale Itérée (ILS)

Logique de conception : Utilise la recherche locale pour atteindre un optimum local, perturbe (par exemple 5 inversions) et répète. Cela équilibre l'exploitation et l'exploration.

- **Attentes** : Amélioration modérée par rapport à la Recherche Aléatoire, limitée par la perturbation.
- **Implémentation** : Utilisation d'un assistant de recherche locale avec mises à jour incrémentielles, perturbation fixée à 5.

- **Observations** : Plus bonne fitness que la Recherche Aléatoire, mais souvent bloqué dans des optima locaux, avec une consistance modérée.
- **Forces et faiblesses** : Trouve rapidement des solutions décentes, mais la perturbation fixe limite la portée globale.
- **Complexité** : $O(k \cdot n \cdot d_{avg})$, où k est le nombre d'itérations, d_{avg} est le degré moyen. Chaque étape de recherche locale est $O(n \cdot d_{avg})$ pour trouver et mettre à jour la meilleure inversion.

4.3. Recherche Tabou (TS)

Logique de conception : Sélectionne le meilleur mouvement autorisé, en utilisant une liste tabou (tenure 10) pour éviter les cycles, acceptant des mouvements moins bons pour diversifier.

- **Attentes** : Meilleure ou égale que ILS grâce à la mémoire, avec une bonne fitness et une meilleure consistance.
- **Implémentation** : Utilisation de l'assistant de recherche locale, suivi de l'état tabou pour chaque variable.
- **Observations** : égal à ILS, parfois ralentie par les contraintes tabou.
- **Forces et faiblesses** : Robuste et consistant, mais la tenure fixe peut ne pas convenir à toutes les instances.
- **Complexité** : $O(k \cdot n \cdot d_{avg})$, similaire à ILS, avec des vérifications tabou $O(1)$ par mouvement.

4.4. Recuit Simulé (SA)

Logique de conception : Démarre avec une température élevée, accepte des mouvements moins bons de manière probabiliste, refroidit pour affiner les solutions (par exemple, température initiale 100, taux de refroidissement 0.95).

- **Attentes** : Fort, probablement le meilleur, grâce au mécanisme d'échappée.
- **Implémentation** : Démarrage aléatoire, acceptation de Metropolis, intégré avec l'assistant de recherche locale.
- **Observations** : Fitness la plus basse, convergence robuste, variance modérée.
- **Forces et faiblesses** : Excellente optimisation globale, mais intensive et sensible au calendrier.
- **Complexité** : $O(k \cdot d_{avg})$, avec k itérations, chaque évaluation et mise à jour d'une inversion dans $O(d_{avg})$.

4.5. Algorithme Génétique (GA)

Logique de conception : Fait évoluer une population (taille = 50) de solutions binaires à l'aide de la sélection par tournoi (taille = 3), du croisement à un point et de la mutation par inversion de bit (taux = 0,01), remplaçant les individus les moins adaptés à chaque itération jusqu'à la limite de temps.

- **Attentes** : On s'attend à ce qu'il surpasse la Recherche Aléatoire et rivalise avec ILS/TS, en exploitant la diversité de la population pour une recherche globale, bien qu'il soit plus lent par itération.
- **Implémentation** : Sous-classe de `Optimiser`, utilisant `QUBOEval` pour l'évaluation de la fitness, avec une sélection, un croisement et une mutation paramétrables pour plus de flexibilité.
- **Observations** : Les résultats étaient meilleurs que la Recherche Aléatoire, mais pas aussi bons que ceux obtenus avec ILS/TS. La convergence était plus lente, et les solutions étaient plutôt médiocres,
- **Forces et faiblesses** : Bien qu'il offre une certaine diversité d'exploration via la population, il présente un coût computationnel élevé et est très sensible aux paramètres.
- **Complexité** : $O(k \cdot (n + m))$, où k est le nombre d'itérations, n est le nombre de variables et m est le nombre de termes non nuls q_{ij}.

4.6. Descente de Gradient (GD)

Logique de conception : Relaxation de QUBO à $z_i \in [-1, 1]$, minimisation de $f(z)$ par mises à jour de gradient, projection vers x. Utilisation de la parcimonie pour plus d'efficacité.

- **Attentes** : Rapide mais sous-optimal en raison des pertes de projection.
- **Implémentation** : Initialisation de z aléatoirement, mise à jour avec un pas de 0.01, projection et évaluation.
- **Observations** : Convergence rapide mais fitness moins bonne (plus élevée) que les méthodes combinatoires.
- **Forces et faiblesses** : Rapide, mais le décalage entre continu et discret limite la qualité.
- **Complexité** : $O(k \cdot m)$, où k est le nombre d'itérations, m est le nombre de termes non nuls, car le calcul du gradient utilise la parcimonie.

4.7. HybridSAGDOptimiser (Hybride SA et GD)

Pour l'approche hybride avec Gradient Descent (GD), j'ai décidé de prendre les meilleurs résultats des algorithmes discrets, qui étaient le recuit simulé (SA).
Logique de conception : Alterne SA pour l'exploration discrète avec GD pour l'affinage continu, projection entre les phases.

- **Attentes** : Potentiellement supérieure à toutes par la synergie.
- **Implémentation** : SA avec refroidissement, GD avec un pas de 0.01 et 100 itérations par cycle, en utilisant l'assistant de recherche locale.
- **Observations** : Fitness trés bonne, proche de SA, mais avec une variance plus élevée due à la projection de GD.
- **Forces et faiblesses** : Polyvalent, mais complexe et dépendant des réglages.
- **Complexité** : $O(k \cdot (d_{avg} + m))$, combinant $O(d_{avg})$ par inversion de SA avec $O(m)$ par cycle de GD.

5. Méthodologie de test

J'ai testé tous les algorithmes sur les instances QUBO 990 à 999 du benchmark PUBO, chaque algorithme étant exécuté pendant 30 secondes (ajusté par un ratio d'étalonnage). En raison de contraintes d'exécution, l'analyse détaillée s'est concentrée sur l'instance 990 (ce qui a influencé le choix de certaines valeurs pour les algorithmes, qui pourraient ne pas être optimales pour d'autres instances, voir les résultats hybrides), avec 30 exécutions par algorithme afin de capturer le comportement statistique. Cependant, j'ai exécuté tous les algorithmes sur les instances 990 à 999 afin de confirmer les tendances générales. Les résultats ont été enregistrés dans des fichiers CSV, incluant l'identifiant de l'instance, le numéro d'exécution, fitness et les valeurs best-known.

6. Configuration expérimentale

6.1. Scope des tests

- **Instances** : Problèmes QUBO 990–999, principalement centrés sur 990 en raison de contraintes d'exécution, avec des exécutions sur les instances restantes pour validation et pour confirmer les tendances.
- **Exécutions** : 30 par optimiseur et par instance, limite de 30 secondes (calibrée).
- **Métriques** : Best fitness, average fitness, gap to best-known, runtime consistency.

6.2. Analyse comparative

- **Sortie** : Fichiers CSV par optimiseur (par exemple, `simulated_annealing_results.csv`), résumés de la console (meilleur, moyen).
- **Matériel** : desktop standard, exécution Java.

7. Démonstrations et validations des temps d'exécution

7.1. Comportement de convergence

- **Simulated Annealing (SA)** : Fitness a chuté rapidement au début (par exemple, de -50000 à -69000 en 5 secondes), puis a été affinée (par exemple, -70787), validée par les journaux d'exécution.
- **Gradient Descent (GD)** : Comportement erratique—souvent positif au début, avec une descente lente jusqu'à -62501, confirmant un local trapping.
- **Hybrid** : La phase SA a suivi un comportement similaire à celui de SA (-70000), tandis que la phase GD a montré des sauts (par exemple, -71021), mais a varié (par exemple, -69289).
- **Random Restarts** : ILS et Hybrid ont été testés avec des redémarrages—ILS a légèrement amélioré ses résultats, tandis que Hybrid a largement dépendu du démarrage avec SA.

7.2. Variation du temps d'exécution

- **Runtime Variation** : SA et Hybrid étaient cohérents (30s), tandis que GD était parfois plus rapide mais inefficace, et Random Search a pleinement utilisé les 30 secondes avec de mauvais résultats.
- **Random Restarts** : ILS a été testé avec des redémarrages au lieu de perturbations ; les résultats se sont légèrement améliorés mais étaient moins cohérents que ceux du Simulated Annealing.
- **Incremental Updates** : La réduction des temps de mouvement est passée de $O(n)$ à $O(d_u)$, avec SA et TS bénéficiant le plus des inversions fréquentes (par exemple, environ 10 fois plus rapides sur des instances peu denses).

7.3. Analyse des résultats (voir rapports)

Note : Certains algorithmes ont été testés plusieurs fois avec différentes valeurs de paramètres, que j'ai perdus de vue, mais les résultats restent disponibles et les paramètres conservés dans le code ont donné les meilleurs résultats.

- **Ce qui a fonctionné**
  - **Simulated Annealing (SA)** : Meilleur performeur globalement, robuste sur tous les instances, surpassant souvent les meilleures valeurs connues (par exemple, -70787).
- **Ce qui n'a pas fonctionné**
  - **Gradient Descent (GD)** : A échoué en standalone en raison d'un décalage d'espace et de pièges locaux (-62501).
  - **Hybrid** : Incohérent - a excellé sur 990, mais a sous-performé ailleurs.
  - **ILS / Tabu** : Ont été à la traîne derrière SA, manquant de son équilibre.

8. Améliorations futures

- Tester d'autres répartitions temporelles adaptatives SA-GD (par exemple, 80%-20%).
- Pénaliser z_i dans GD pour garantir un alignement binaire.
- Étendre les exécutions de 995 à 999 pour une validation plus large.