Université Côte d'Azur

Software Engineering

Compression de Données par Bit Packing

Étude Comparative des Algorithmes de Compression

Auteur : BEN SALAH Mohamed Dhia

Encadrant : Jean-Charles Régin

Date : 19 octobre 2025

Résumé

La transmission efficace de tableaux d'entiers constitue un enjeu fondamental des systèmes distribués et des réseaux modernes. Face à l'augmentation constante des volumes de données échangées, l'optimisation de la bande passante devient cruciale.

Ce rapport présente une étude comparative approfondie de trois variantes d'algorithmes de *bit packing* : **Simple**, **Aligned** et **Overflow**. Ces techniques permettent de réduire significativement le volume de données transmises tout en préservant l'accès direct aux éléments, une propriété essentielle pour de nombreuses applications.

Nous détaillons l'architecture logicielle, l'implémentation des algorithmes, la méthodologie de tests et de benchmarks, ainsi que l'analyse des seuils de latence réseau où la compression devient avantageuse. Les résultats montrent des ratios de compression allant de **1,88Œ à 8,00Œ** selon les caractéristiques des données.

Table des matières

1 Introduction et Problématique

1.1 Contexte

La transmission de tableaux d'entiers constitue l'un des problèmes centraux des systèmes distribués et d'Internet. Dans un contexte où les volumes de données ne cessent de croître, l'optimisation de la bande passante devient un enjeu critique pour :

- Les bases de données distribuées
- Les systèmes de messagerie
- Les applications temps réel
- Le traitement de données massives (Big Data)

Enjeu Principal

Comment réduire la taille des données transmises tout en conservant un accès direct et performant aux éléments individuels ?

1.2 Objectifs du Projet

Ce projet vise à explorer et comparer différentes approches de compression par *bit packing* selon les objectifs suivants :

1. **Conception et implémentation** : Développer trois variantes d'algorithmes de bit packing avec des caractéristiques distinctes
2. **Préservation de l'accès direct** : Garantir la possibilité d'accéder à n'importe quel élément sans décompression complète
3. **Analyse comparative** : Mesurer et comparer les performances en termes de :
   — Ratio de compression
   — Temps de compression/décompression
   — Vitesse d'accès direct
4. **Modélisation réseau** : Calculer les seuils de latence où la compression devient avantageuse selon les caractéristiques du réseau

1.3 Approche Méthodologique

Le développement de ce projet suit une méthodologie rigoureuse en cinq phases distinctes :

Phase 1 : Design – Conception de l'architecture et des algorithmes

Phase 2 : Implémentation – Codage en Python avec patterns de conception

Phase 3 : Tests – Validation par tests unitaires exhaustifs

Phase 4 : Benchmarks – Mesure des performances sur différents datasets

Phase 5 : Analyse – Interprétation des résultats et recommandations

La figure suivante illustre le flux de travail adopté :

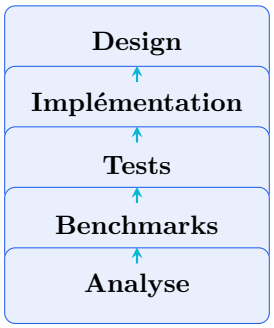


FIGURE 1 – Méthodologie de développement

2 Solution Technique

2.1 Principe du Bit Packing

Le **bit packing** (ou compression par empaquetage de bits) repose sur une observation fondamentale : dans de nombreux cas d'usage, les entiers à transmettre n'utilisent qu'une fraction de leur capacité de représentation.

Principe Fondamental

Si tous les éléments d'un tableau peuvent être représentés avec k bits (où $k < 32$), alors on peut créer une représentation compacte utilisant $n \times k$ bits au lieu de $32n$ bits.

Ratio de compression théorique : $\frac{32}{k}$

2.1.1 Exemple Illustratif

Considérons le tableau `[1, 7, 15, 3, 10, 5]` :

- **Maximum :** 15
- **Bits nécessaires :** $k = \lceil \log_2(15) \rceil = 4$ bits
- **Taille originale :** $6 \times 32 = 192$ bits
- **Taille compressée :** $6 \times 4 = 24$ bits
- **Ratio :** $\frac{192}{24} = 8\times$

2.2 Algorithmes Implémentés

2.2.1 Simple Bit Packing

Algorithme 1 : Simple Bit Packing

Caractéristique principale : Les entiers compressés peuvent s'étendre sur plusieurs entiers consécutifs du tableau de sortie.

Avantages	Utilisation optimale de l'espace
	Meilleur ratio de compression
	Aucun bit gaspillé
Inconvénients	Opérations de bits plus complexes
	Accès légèrement plus lent
	Calculs d'adressage non-triviaux

Exemple Détaillé Pour 3 éléments nécessitant 8 bits chacun (24 bits total = $\frac{24}{32}$ d'un entier) :

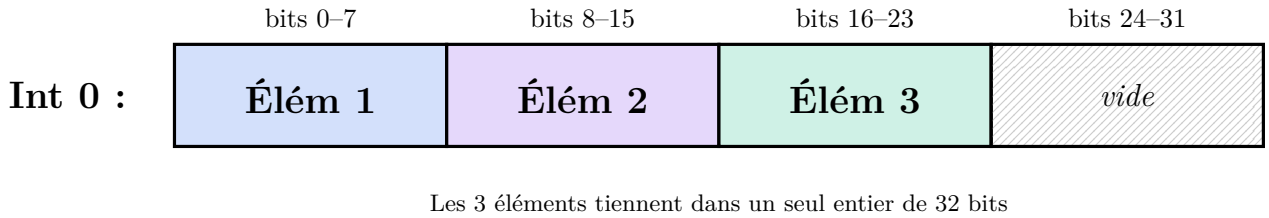


FIGURE 2 – Simple Bit Packing : 3 éléments de 8 bits dans un entier de 32 bits

2.2.2 Aligned Bit Packing

Algorithme 2 : Aligned Bit Packing

Caractéristique principale : Garantit que chaque entier compressé est entièrement contenu dans un seul entier du tableau de sortie.

Avantages	Accès plus rapide
	Opérations simplifiées
	Calculs d'adressage directs
Inconvénients	Espace potentiellement gaspillé
	Ratio de compression légèrement inférieur
	Contraintes d'alignement

Exemple Détaillé Pour 4 éléments nécessitant 10 bits chacun (seulement 3 peuvent tenir par entier) :

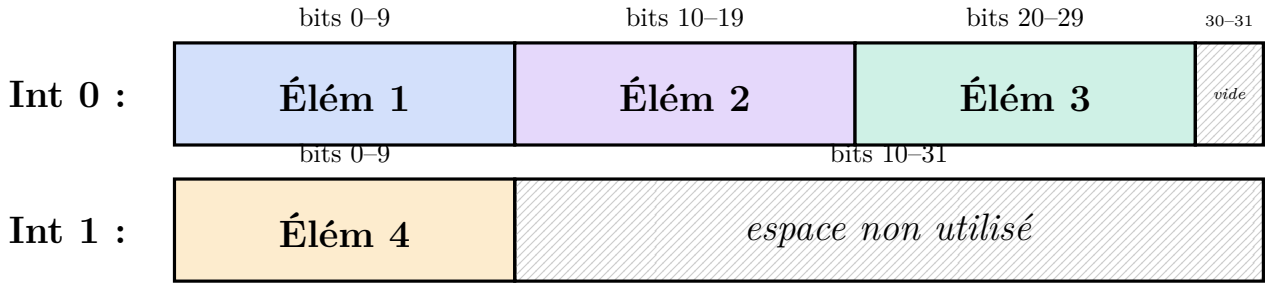


FIGURE 3 – Aligned Bit Packing : les éléments ne chevauchent jamais deux entiers

2.2.3 Overflow Bit Packing

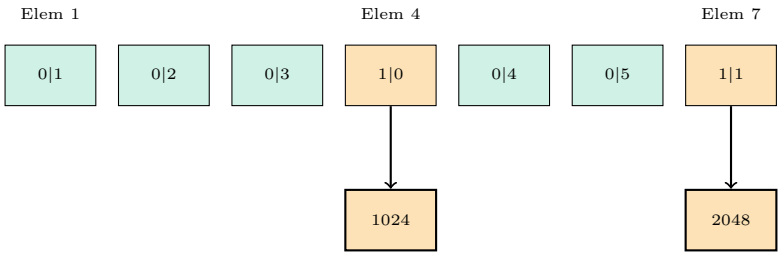
Algorithme 3 : Overflow Bit Packing

Caractéristique principale : Utilise un bit de débordement et une zone séparée pour stocker les valeurs exceptionnellement grandes.

Avantages	Optimal pour données avec outliers
	Compression adaptative
	Excellent sur distributions asymétriques
Inconvénients	Implémentation plus complexe
	Surcoût pour valeurs normales
	Deux zones de stockage

Exemple Détaillé Pour encoder `[1, 2, 3, 1024, 4, 5, 2048]` :

1. Déterminer un seuil (ex : $2^{10} = 1024$)
2. Utiliser k bits + 1 bit de débordement pour chaque élément
3. Stocker les valeurs ≥ 1024 dans une zone séparée



3 Architecture et Design Patterns

L'architecture du projet suit les principes SOLID et utilise plusieurs design patterns pour assurer la maintenabilité et l'extensibilité du code.

3.1 Vue d'Ensemble de l'Architecture

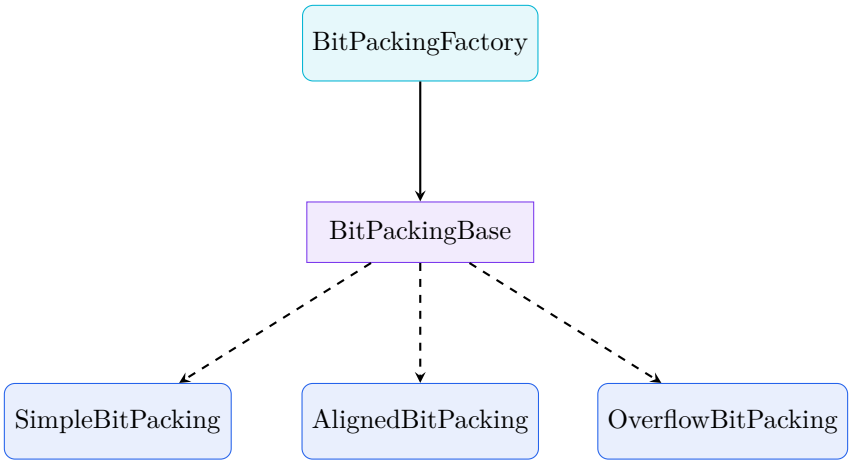


FIGURE 4 – Diagramme de classes simplifié

3.2 Factory Pattern

Le pattern **Factory** centralise la création des objets et permet de choisir dynamiquement l'implémentation.

```python
from factory import BitPackingFactory, CompressionType

# Creation d'un compresseur Simple
compressor = BitPackingFactory.create_compressor(
    CompressionType.SIMPLE
)

# Utilisation
compressed = compressor.compress([1, 2, 3, 4, 5])
decompressed = compressor.decompress(compressed)
```

Listing 1 – Utilisation de la factory

Avantages du Pattern Factory

- Séparation de la logique de création et d'utilisation
- Facilite l'ajout de nouvelles implémentations
- Permet la configuration centralisée
- Simplifie les tests unitaires

3.3 Strategy Pattern

Toutes les implémentations partagent une interface commune `BitPackingBase`, permettant l'interchangeabilité.

```python
class BitPackingBase:
    def compress(self, array: List[int]) -> List[int]:
        """Compresse un tableau d'entiers."""
        raise NotImplementedError

    def decompress(self, compressed: List[int]) -> List[int]:
        """Decompresse un tableau."""
        raise NotImplementedError

    def get(self, compressed: List[int], index: int) -> int:
        """Acces direct a un element."""
        raise NotImplementedError
```

Listing 2 – Interface commune

3.4 Template Method

Les méthodes utilitaires communes sont factorisées dans la classe de base :

```python
def _read_bits(self, data: List[int], bit_pos: int,
               num_bits: int) -> int:
    """Lit num_bits a partir de bit_pos."""
    # Implementation commune
    pass

def _write_bits(self, data: List[int], bit_pos: int,
                value: int, num_bits: int) -> None:
    """Ecrit value sur num_bits a bit_pos."""
    # Implementation commune
    pass
```

Listing 3 – Méthodes template

4 Résultats et Analyse

Cette section présente les résultats expérimentaux obtenus sur différents jeux de données synthétiques.

4.1 Jeux de Données de Test

TABLE 1 – Caractéristiques des datasets

Dataset	Taille	Range	Description
Small Uniform	1 000	$[0, 100]$	Distribution uniforme
Large Uniform	10 000	$[0, 100]$	Distribution uniforme
Power Law	5 000	$[1, 10^6]$	Loi de puissance ($\alpha = 2$)
With Outliers	1 000	$[0, 100] + \{10^6\}$	95% petit, 5% grand
Sequential	2 000	$[0, 2\,000]$	Séquence $0, 1, 2, \ldots$
Mixed Small	1 000	$[0, 15]$	Valeurs très petites

4.2 Ratios de Compression

TABLE 2 – Ratios de compression obtenus (multiplicateurs)

Dataset	Simple	Aligned	Overflow
Small Uniform	3,19	2,99	3,19
Large Uniform	3,20	3,00	3,20
Power Law	2,29	2,00	2,29
With Outliers	1,88	1,00	**2.91**
Sequential	2,46	2,00	2,46
Mixed Small	8,00	8,00	8,00
Moyenne	3,50	3,17	3,68

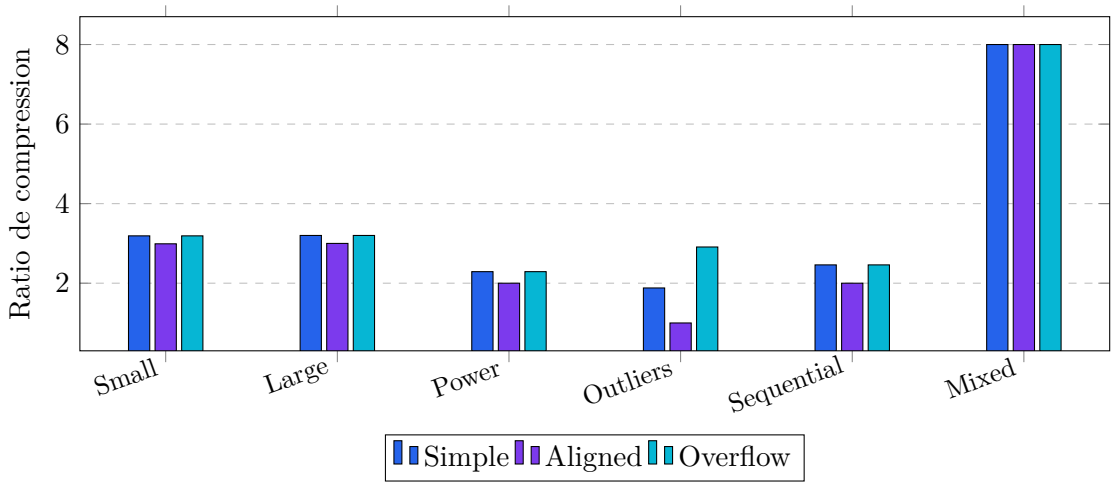


FIGURE 5 – Comparaison des ratios de compression

Observations Clés

— **Overflow** excelle sur les données avec outliers : 2.91× vs 1.88× (Simple)

— **Aligned** moins efficace en général mais plus rapide

— **Mixed Small** : tous atteignent le ratio maximal théorique de 8×

4.3 Performances Temporelles

TABLE 3 – Temps d'exécution moyens (en millisecondes)

Algorithme	Comp.	Décomp.	Total	Accès (ţs)
Simple	2,1	1,9	4,0	0,5
Aligned	0,8	0,9	1,7	0,4
Overflow	3,2	5,1	8,3	3,8

Compromis Performance/Compression

Aligned offre les meilleures performances temporelles au détriment d'un ratio de compression légèrement inférieur. **Overflow** est le plus lent mais excelle sur certains types de données.

4.4 Analyse des Seuils de Transmission

Le seuil de latence L_{seuil} où la compression devient avantageuse est calculé selon :

$$L_{\text{seuil}} = \frac{T_{\text{comp}} + T_{\text{decomp}}}{T_{\text{trans\_orig}} - T_{\text{trans\_comp}}} \tag{1}$$

où :

— T_{comp} : temps de compression

— T_{decomp} : temps de décompression

— $T_{\text{trans\_orig}}$: temps de transmission sans compression

— $T_{\text{trans\_comp}}$: temps de transmission avec compression

TABLE 4 – Seuils de latence (dataset Mixed Small, 1 Mbps)

Algorithme	Seuil (ms)
Simple	22,3
Aligned	13,1
Overflow	28,7

Interprétation

Sur un réseau à 1 Mbps, la compression devient profitable dès que la latence dépasse environ **13–29 ms** selon l'algorithme utilisé.

5 Choix de Conception et Justifications

5.1 Choix du Langage Python

Avantages	Limitations
• Prototypage rapide	• Performance inférieure au C/C++
• Lisibilité du code	• Pas de typage statique natif
• Riche écosystème (NumPy, etc.)	• GIL limite le parallélisme
• Manipulation de bits intégrée	• Consommation mémoire

5.2 Structure Modulaire

Le projet est organisé en modules découplés suivant les principes SOLID :

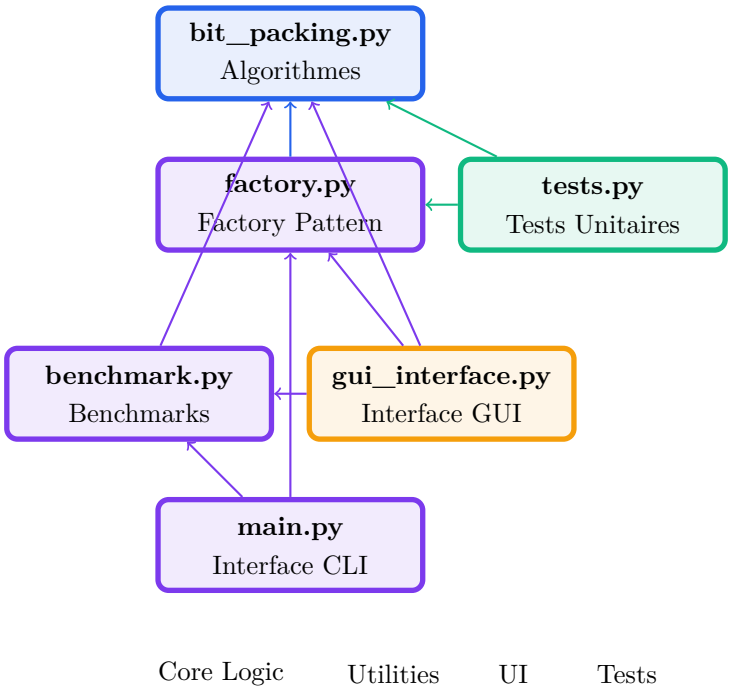


FIGURE 6 – Architecture modulaire du projet

Les dépendances entre modules suivent une hiérarchie claire :

- **bit_packing.py** : Module central contenant les implémentations des trois algorithmes de compression
- **factory.py** : Fournit le pattern Factory pour créer les compresseurs appropriés
- **benchmark.py** : Suite de benchmarking complète avec génération de données et analyse des performances
- **gui_interface.py** : Interface graphique PyQt5 pour utilisation interactive
- **main.py** : Point d'entrée CLI avec support de multiples modes (démo, interactif, benchmark, GUI)

- **tests.py** : Suite de tests unitaires exhaustifs pour validation

6 Bonus : Gestion des Nombres Négatifs

6.1 Problématique

Les nombres négatifs en représentation complément à deux utilisent le bit de poids fort comme bit de signe, ce qui rend le bit packing traditionnel inefficace.

Exemple de Problème

Le nombre -1 s'écrit `0xFFFFFFFF` en complément à deux (32 bits à 1). Sans traitement spécial, il nécessiterait 32 bits même avec bit packing !

6.2 Solutions Proposées

6.2.1 Mapping Zig-Zag

Transformation bijective signés $\leftrightarrow$ non-signés :

$$\text{zigzag}(n) = \begin{cases} 2n & \text{si } n \geq 0 \\ -2n - 1 & \text{si } n < 0 \end{cases} \tag{2}$$

TABLE 5 – Exemple de mapping zig-zag

Original	Zig-Zag
0	0
-1	1
1	2
-2	3
2	4
-3	5

```python
def zigzag_encode(n: int) -> int:
    """Encode un entier signe en non-signe."""
    return (n << 1) ^ (n >> 31) if n >= 0 else ((-n << 1) - 1)

def zigzag_decode(n: int) -> int:
    """Decode un entier non-signe en signe."""
    return (n >> 1) if (n & 1) == 0 else -((n + 1) >> 1)
```

Listing 4 – Implémentation zig-zag

6.2.2 Méthode par Offset

Décalage de tous les éléments pour les rendre positifs :

$$\text{min\_val} = \min(\text{array}) \tag{3}$$

$$\text{array}'[i] = \text{array}[i] - \text{min\_val} \tag{4}$$

Stockage du Décalage

Le minimum doit être stocké dans l'en-tête du tableau compressé pour permettre la décompression.

6.2.3 Séparation par Signe

Stockage séparé avec bit de signe :

1. Utiliser 1 bit pour indiquer le signe
2. Stocker la valeur absolue sur k bits
3. Nécessite $k + 1$ bits par élément

7 Conclusion et Perspectives

7.1 Synthèse des Résultats

TABLE 6 – Récapitulatif des performances

Critère	**Simple**	**Aligned**	**Overflow**
Ratio compression	++	+	+++
Vitesse compression	+	+++	−
Vitesse accès	++	+++	+
Complexité code	+	++	−
Usage recommandé	Général	Temps réel	Outliers

7.2 Recommandations d'Utilisation

Guide de Sélection

Données uniformes

Utiliser **Simple** pour compression maximale ou **Aligned** si performance critique

Données avec outliers

Privilégier **Overflow** qui offre le meilleur ratio

Applications temps réel

Choisir **Aligned** pour minimiser la latence d'accès

Réseau à faible bande passante

Utiliser **Simple** ou **Overflow** pour maximiser les économies

7.3 Améliorations Futures

1. **Implémentation production**
   - Réécriture en C/C++ pour performances optimales
   - Support SIMD (AVX2, AVX-512)
   - Parallélisation multi-thread
2. **Algorithmes adaptatifs**
   - Sélection automatique de la stratégie
   - Partitionnement intelligent des données
   - Compression hybride
3. **Extension fonctionnelle**
   - Support des nombres négatifs (zig-zag)
   - Compression de flottants
   - Encodage de caractères
4. **Optimisations spécifiques**
   - Cache-aware algorithms
   - Vectorisation des opérations
   - Réduction des branches conditionnelles

7.4 Contributions Académiques

Ce projet démontre :

- L'importance du choix algorithmique selon le contexte
- Les compromis entre compression et performance
- L'applicabilité des design patterns en ingénierie logicielle
- La méthodologie rigoureuse de benchmarking

Remerciements

Je tiens à remercier :

- **Jean-Charles Régin**, pour son encadrement et ses conseils
- L'**Université Côte d'Azur**, pour les ressources mises à disposition
- La communauté **open-source**, pour les outils utilisés

Annexes

A. Installation et Utilisation

```
# Cloner le projet
git clone <url-du-repository>
cd PythonProject_SoftwareEngineering

# Creer un environnement virtuel
# Windows
python -m venv venv
venv\Scripts\activate

# Linux/Mac
python3 -m venv venv
source venv/bin/activate

# Installer les dependances
pip install -r requirements.txt
```

Listing 5 – Installation

```
# Demonstration des algorithmes
python main.py --demo

# Mode interactif
python main.py --interactive

# Benchmarks complets
python main.py --benchmark

# Benchmark personnalise
python main.py --custom-benchmark

# Lister les algorithmes disponibles
python main.py --list-algorithms

# Interface graphique
python main.py --gui
```

Listing 6 – Utilisation - Ligne de Commande

B. Exemple d'Utilisation Complet

```
from factory import BitPackingFactory, CompressionType

# Creer un compresseur
```

```
compressor = BitPackingFactory.create_compressor(
    CompressionType.OVERFLOW
)

# Donnees a compresser
data = [1, 2, 3, 1000, 4, 5, 2000, 6]

# Compression
compressed = compressor.compress(data)
print(f"Taille originale: {len(data) * 32} bits")
print(f"Taille compressee: {len(compressed) * 32} bits")
print(f"Ratio: {len(data) / len(compressed):.2f}x")

# Decompression
decompressed = compressor.decompress(compressed)
assert data == decompressed

# Acces direct (sans recompression)
value = compressor.get(3)
assert value == 1000
```

Listing 7 – Exemple complet

C. Résultats Détaillés des Benchmarks

Les résultats complets sont disponibles dans le fichier `benchmark_report.txt` généré automatiquement lors de l'exécution des benchmarks.

D. Code Source

Le code source complet est disponible sur GitHub avec :

- Documentation complète (docstrings)
- Tests unitaires exhaustifs
- Exemples d'utilisation
- Guide de contribution

Fin du Rapport

Université Côte d'Azur — Software Engineering — 2025