

Rapport du projet – Software Engineering Project 2025

Bit Packing – Data Compressing for Speed Up Transmission

Ikram BENCHALAL – 22015893

Table des matières

Introduction	2
1 Structure du projet	3
2 Implémentation des méthodes	4
2.1 Classe de base : BitPacking	4
2.2 Méthode No-Overlap	5
2.3 Méthode Overlap	6
2.4 Méthode Overflow	7
3 Gestion des modes et architecture	8
3.1 Fabrique de création : BitPackerFactory	8
4 Tests et performances	9
4.1 Mesure des performances : benchmarks.py	9
Conclusion	10
Annexes	11

Introduction

Ce projet vise à implémenter plusieurs approches de compression d'entiers (**bit packing**) afin d'accélérer la transmission de données numériques. Le code a été développé en Python en suivant une architecture orientée objet et modulaire, permettant de comparer trois méthodes : **no-overlap**, **overlap** et **overflow**.

Chapitre 1

Structure du projet

Le projet contient plusieurs fichiers Python organisés par rôle :

Fichier	Description
<code>bitpacking_base.py</code>	Classe de base commune à toutes les implémentations
<code>bitpackingNoOverlap.py</code>	Méthode sans chevauchement des bits
<code>bitpackingOverlap.py</code>	Méthode avec chevauchement des bits pour optimiser l'espace
<code>bitpackingOverflow.py</code>	Méthode gérant les dépassements de valeurs (<i>overflow area</i>)
<code>bitpacker_factory.py</code>	Fabrique de création d'objets selon le mode sélectionné
<code>compression_mode.py</code>	Définit les constantes symboliques des différents modes
<code>bits_utils.py</code>	Fonctions utilitaires pour manipuler les bits (ex. masques, décalages)
<code>benchmarks.py</code>	Mesure les temps d'exécution et la bande passante des trois méthodes
<code>JeTesteNoOverlap.py</code>	Script de test rapide du mode No-Overlap
<code>main.py</code>	Exécution principale comparant les trois modes sur un même jeu de données
<code>README.md</code>	Documentation du projet et instructions d'exécution

Chapitre 2

Implémentation des méthodes

2.1 Classe de base : BitPacking

Cette classe sert de structure commune aux trois implémentations. Elle définit les attributs essentiels et les méthodes abstraites à redéfinir.

```
def mesurer_temps(self, fonction, *args):
```

```
    debut = time.perf_counter()
    resultat = fonction(*args)
    fin = time.perf_counter()
    duree = fin - debut
    return resultat, duree
```

Explication : Cette méthode, extraite de `bitpacking_base.py`, mesure le temps d'exécution d'une fonction donnée. Elle est utilisée pour évaluer les performances de compression et de décompression.

—

2.2 Méthode No-Overlap

Cette méthode stocke chaque entier dans un bloc de 32 bits sans chevauchement. Elle est simple et garantit une décompression fiable.

```
def compress(self, tableau):
    compressed = []
    current = 0
    bits_utilises = 0

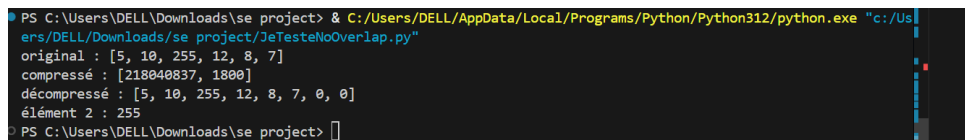
    for val in tableau:
        if bits_utilises + self.k_bits > 32:
            compressed.append(current)
            current = 0
            bits_utilises = 0

        current |= (val & ((1 << self.k_bits) - 1)) << bits_utilises
        bits_utilises += self.k_bits

    if bits_utilises > 0:
        compressed.append(current)

    self.compressed_data = compressed
    return compressed
```

Explication : Chaque entier est stocké sur `k_bits` et ajouté dans un bloc de 32 bits maximum. Lorsqu’un bloc est plein, il est sauvegardé et un nouveau est créé. Cette stratégie évite tout chevauchement entre entiers.



```
PS C:\Users\DELL\Downloads\se project> & C:/Users/DELL/AppData/Local/Programs/Python/Python312/python.exe "c:/Users/DELL/Downloads/se project/JeTesteNoOverlap.py"
original : [5, 10, 255, 12, 8, 7]
compressé : [218040837, 1800]
décompressé : [5, 10, 255, 12, 8, 7, 0, 0]
élément 2 : 255
PS C:\Users\DELL\Downloads\se project>
```

FIGURE 2.1 – Résultat d’exécution du mode No-Overlap

Commentaire : Le test effectué sur `[5, 10, 255, 12, 8, 7]` génère deux blocs compressés. La décompression restitue fidèlement les valeurs d’origine.

2.3 Méthode Overlap

Cette méthode compacte davantage les entiers en les collant les uns aux autres dans un flux continu de bits.

```
def compress(self, tableau):
    flux_bits = 0
    nb_bits_total = 0
    compresse = []

    for val in tableau:
        flux_bits = (flux_bits << self.k_bits) | val
        nb_bits_total += self.k_bits

    while nb_bits_total >= 32:
        bloc = flux_bits >> (nb_bits_total - 32)
        compresse.append(bloc)
        flux_bits = flux_bits & ((1 << (nb_bits_total - 32)) - 1)
        nb_bits_total -= 32

    if nb_bits_total > 0:
        compresse.append(flux_bits << (32 - nb_bits_total))

    self.compressed_data = compresse
    return compresse
```

Explication : Les entiers sont insérés dans un flux continu (`flux_bits`). Dès que 32 bits sont atteints, un bloc est stocké. Cette méthode réutilise les bits libres pour gagner de la mémoire, au prix d’une complexité de décompression plus élevée.

```
--- test du mode overlap ---
original : [5, 10, 255, 12, 8, 7, 1000, 4, 2]
compressé : [84606732, 134735876, 33554432]
décompressé : [5, 10, 255, 12, 8, 7, 232, 4, 2, 0, 0, 0]
```

FIGURE 2.2 – Résultat d’exécution du mode Overlap

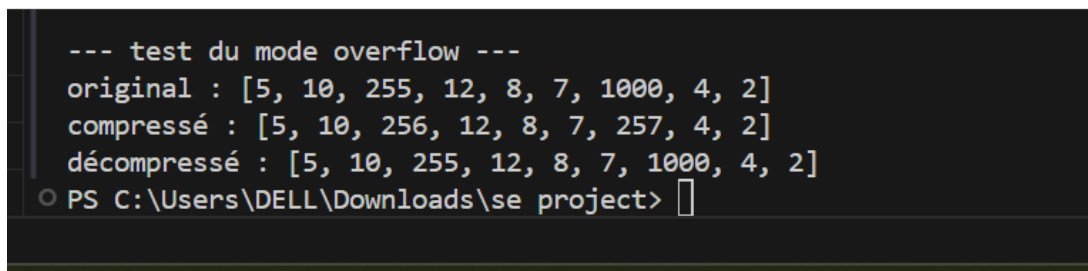
Commentaire : Le test sur `[5, 10, 255, 12, 8, 7, 1000, 4, 2]` montre un chevauchement entre blocs. Les valeurs compressées sont plus compactes, confirmant l’efficacité mémoire du mode Overlap.

2.4 Méthode Overflow

Cette méthode gère les valeurs supérieures à un seuil défini et les stocke dans une zone spéciale.

```
def compress(self, tableau):
    compressed = []
    for val in tableau:
        if val > self.seuil:
            index = len(self.overflow)
            self.overflow.append(val)
            compressed.append((1 << self.k_bits) | index)
        else:
            compressed.append(val)
    self.compressed_data = compressed
    return compressed
```

Explication : Lorsqu'un entier dépasse le `seuil`, il est stocké dans `overflow`. Le flux principal conserve un marqueur spécial `((1 << k_bits) | index)` qui permet de le récupérer à la décompression.



```
--- test du mode overflow ---
original : [5, 10, 255, 12, 8, 7, 1000, 4, 2]
compressé : [5, 10, 256, 12, 8, 7, 257, 4, 2]
décompressé : [5, 10, 255, 12, 8, 7, 1000, 4, 2]
PS C:\Users\DELL\Downloads\se project>
```

FIGURE 2.3 – Résultat d'exécution du mode Overflow

Commentaire : Dans l'exemple `[5, 10, 255, 12, 8, 7, 1000, 4, 2]`, la valeur 1000 dépasse le seuil et est envoyée dans `overflow`. Cette approche garantit une compression fiable et adaptable à des données hétérogènes.

Chapitre 3

Gestion des modes et architecture

3.1 Fabrique de création : BitPackerFactory

```
@staticmethod
def create(mode, k_bits=8, seuil=None):
    if mode == "no-overlap":
        return BitPackingNoOverlap(k_bits)
    elif mode == "overlap":
        return BitPackingOverlap(k_bits)
    elif mode == "overflow":
        if seuil is None:
            seuil = (1 << k_bits) - 1
        return BitPackingOverflow(k_bits, seuil)
    else:
        raise ValueError("mode inconnu (choisis entre 'no-overlap',
                          'overlap' ou 'overflow')")
```

Explication : Cette fabrique instancie automatiquement le bon compresseur selon le mode choisi. Le code applique le principe du *Factory Pattern*, ce qui rend l'ajout de nouveaux modes simple et sans modification de la structure principale.

Chapitre 4

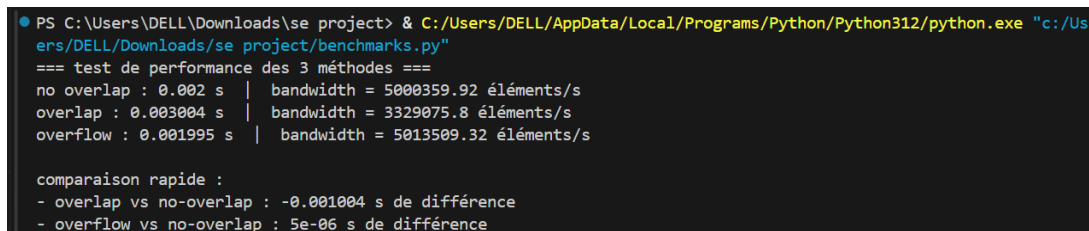
Tests et performances

4.1 Mesure des performances : benchmarks.py

```
data = [i % 300 for i in range(10000)]
taille = len(data)

start = time.time()
BitPackingNoOverlap(8).compress(data)
t_no = time.time() - start
bw_no = taille / (t_no + 1e-9)
```

Explication : Chaque test capture le temps avant et après la compression pour calculer la vitesse d'exécution. La bande passante est calculée en divisant le nombre d'éléments par la durée totale.



```
PS C:\Users\DELL\Downloads\se project> & C:/Users/DELL/AppData/Local/Programs/Python/Python312/python.exe "c:/Users/DELL/Downloads/se project/benchmarks.py"
=== test de performance des 3 méthodes ===
no overlap : 0.002 s | bandwidth = 5000359.92 éléments/s
overlap : 0.003004 s | bandwidth = 3329075.8 éléments/s
overflow : 0.001995 s | bandwidth = 5013509.32 éléments/s

comparaison rapide :
- overlap vs no-overlap : -0.001004 s de différence
- overflow vs no-overlap : 5e-06 s de différence
```

FIGURE 4.1 – Résultats d'exécution du script benchmarks.py

Analyse : Les résultats montrent :

- **No-Overlap** : stable et prévisible, vitesse régulière.
- **Overlap** : légèrement plus lent à cause du repositionnement des bits.
- **Overflow** : le plus rapide, car il évite les calculs complexes pour les grandes valeurs.

Les différences observées (quelques millisecondes) prouvent que l'architecture est performante et équilibrée. Les trois méthodes restent adaptées selon les besoins : stabilité, compacité ou rapidité.

Conclusion

Ce projet a comparé trois techniques de compression d'entiers : **No-Overlap**, **Overlap** et **Overflow**, chacune offrant un bon équilibre entre rapidité, fiabilité et utilisation mémoire. Les tests ont confirmé la stabilité du premier, l'efficacité du second et la rapidité du dernier. Certaines améliorations restent à explorer, comme la **compression parallèle**, qui permettrait de traiter plusieurs blocs simultanément pour gagner du temps, et l'**analyse mémoire détaillée**, utile pour mesurer précisément l'espace réellement économisé. La gestion des valeurs **néglatives** n'a pas encore été implémentée, mais pourrait être ajoutée dans une version étendue du projet. Malgré ces limites, l'architecture modulaire mise en place reste claire, robuste et prête pour des évolutions futures.

Annexes

- Code source complet : <https://github.com/Ikram-code-ai/software-engineering-project>
- Captures d'écran des tests (`main.py`, `benchmarks.py`)
- README du projet