

Projet : Compression d'entiers par BitPacking – Software Engineering Project 2025

Étudiante : RANIA TAHA
Encadrant : Jean-Charles RÉGIN
Année universitaire : 2025 – 2026

Fais-le : 02/11/2025
Université Côte d'Azur
Master 1 Informatique parcours IA
2025 – 2026

Table des matières

1. Introduction

2. Problématique et objectifs

- 2.1. Problématique
 - 2.2. Objectifs du projet
-

3. Description des méthodes de compression

- 3.1. Principe général
-

4. Variantes implémentées

- 4.1. Mode *Crossing*
 - 4.2. Mode *Non-crossing*
 - 4.3. Mode *Overflow*
-

5. Contraintes du projet

6. Méthodologie de développement

7. Structure du projet et rôle des fichiers

8. Analyse expérimentale et résultats

- 8.1. Exemple concret
 - 8.2. Résultats du benchmark
 - 8.3. Interprétation détaillée des résultats
 - 8.4. Analyse de la latence
 - 8.5. Fonctionnement des opérations principales
 - 8.6. Types de données testées
-

9. Discussion et limites

10. Conclusion et perspectives

1. Introduction

Le volume des données numériques générées à travers le monde croît de manière exponentielle.

Cette situation impose la recherche de solutions visant à **optimiser l'espace de stockage et la vitesse de traitement**.

La compression s'impose ainsi comme un outil essentiel pour répondre à ces besoins.

Dans ce contexte, la **compression d'entiers** occupe une place importante dans plusieurs domaines tels que :

- les **bases de données**,
- les **systèmes embarqués**,
- les **protocoles réseau**,
- et les **applications d'intelligence artificielle** manipulant de grandes quantités de valeurs numériques.

L'objectif du projet *BitPacking 2025* est de proposer une **implémentation modulaire et performante** de cette technique, tout en permettant une **évaluation précise de ses performances**.

2. Problématique et objectifs

2.1. Problématique

Chaque entier est généralement stocké sur 32 bits, même si sa valeur réelle ne nécessite que quelques bits.

Cette inefficacité conduit à une **surconsommation mémoire** et à des **temps de traitement plus élevés**.

Le problème consiste donc à trouver **une méthode efficace pour compresser une suite d'entiers sans perte**, tout en permettant un **accès direct à chaque valeur** sans devoir tout décompresser.

2.2. Objectifs du projet

- Implémenter trois variantes de BitPacking : *Crossing*, *Non-crossing* et *Overflow*.
- Comparer leurs performances (taux de compression, vitesse, latence).
- Créer une **interface CLI** intuitive pour exécuter les fonctions de compression et de décompression.
- Mettre en place un **système de tests automatiques et de benchmark** pour évaluer les performances.

3. Description des méthodes de compression

3.1. Principe général

L'idée du BitPacking est de **minimiser le nombre de bits** utilisés pour représenter chaque entier, selon la formule :

$$k = \lceil \log_2(\max(A) + 1) \rceil$$

où A représente l'ensemble des entiers à compresser.

Exemple :

Pour les valeurs : 1, 2, 3, 1024, 4, 5, 2048

→ $\max(A) = 2048$, donc $k = 12$.

Chaque entier sera donc stocké sur **12 bits au lieu de 32**.

Ainsi, 7 entiers nécessitent seulement **84 bits** (au lieu de 224), soit un **gain de 62,5 %**.

4. Variantes implémentées

4.1. Mode *Crossing*

Les entiers peuvent **chevaucher plusieurs mots binaires**.

Chaque bit disponible est exploité, ce qui maximise la compression, au prix d'une gestion plus complexe des décalages.

4.2. Mode *Non-crossing*

Chaque entier est **aligné** sur un mot mémoire.

Ce mode est plus simple à gérer, facilite la **lecture directe**, mais offre un taux de compression légèrement inférieur.

4.3. Mode *Overflow*

Lorsqu'une valeur dépasse la capacité $2^k - 1$, elle est **déportée dans une zone de débordement** (*overflow area*).

Cette méthode garantit la **robustesse** et la **compatibilité universelle** du système.

5. Contraintes du projet

Le développement a été soumis à plusieurs contraintes techniques et organisationnelles :

- Langage imposé : **Python 3.11**
 - Architecture modulaire (cli, bitpacking, tests)
 - Compatibilité entre les trois modes
 - Gestion des erreurs et exceptions
 - Optimisation du code pour un **temps de compression < 5 ms** sur 1000 entiers
-

6. Méthodologie de développement

Le projet a suivi une **approche incrémentale et testée** :

1. Analyse du besoin et définition des variantes.
2. Implémentation des modules bitpacking/.
3. Création des CLI avec la bibliothèque *Click*.
4. Tests unitaires automatisés (*Pytest*).
5. Mesure des performances via *benchmark.py*.
6. Validation et comparaison des résultats.

Chaque étape a été accompagnée d'une phase de **test et validation** afin d'assurer la stabilité du code avant intégration finale.

7. Structure du projet et rôle des fichiers

```
bitpackingproject/
|
└── bitpacking/          # Cœur du projet : logique de compression
    ├── __init__.py        # Rend le dossier importable en module Python
    ├── core.py            # Fonctions binaires communes (pack/unpack bits)
    ├── crossing.py        # Mode Crossing → compression maximale (bits partagés)
    ├── noncrossing.py     # Mode Non-crossing → alignement sur mots binaires
    ├── overflow.py        # Mode Overflow → gestion des débordements
    └── factory.py         # Fabrique : crée dynamiquement le bon compresseur
```

```
|  
|   └── cli/          # Interface utilisateur (Command Line Interface)  
|       |   └── __init__.py      # Rend le package exécutable (obligatoire)  
|       |   └── bitpacking_cli.py    # CLI principale : crossing / non_crossing  
|       └── overflow_cli.py      # CLI spécifique au mode overflow  
|  
|  
|   └── tests/         # Tests unitaires (Pytest)  
|       |   └── __init__.py      # Package d'import pour les tests  
|       |   └── test_core_bits.py    # Teste la manipulation des bits (core)  
|       |   └── test_crossing.py     # Teste la logique crossing  
|       |   └── test_overflow.py     # Teste la logique overflow  
|       └── test_smoke.py        # Test global : vérifie le bon import des modules  
|  
|  
|   └── benchmark.py      # Mesure les performances et la latence  
|  
|  
|   └── data.txt          # Jeu de données d'exemple (entiers à compresser)  
|  
|  
|   └── requirements.txt    # Dépendances Python (click, pytest, packaging...)  
|  
|  
|   └── README.md          # Documentation du projet (exécution, modes, exemples)  
|  
|  
|   └── .gitignore          # exclusion des fichiers inutiles (.pyc, __pycache__, .bin...)  
|  
|  
└── Rania_Taha_BitPacking_Report.pdf # Rapport du projet
```

8. Analyse expérimentale et résultats

8.1. Exemple concret

Fichier d'entrée utilisé :

data.txt → 1 2 3 1024 4 5 2048

Commandes exécutées :

```
python -m cli.bitpacking_cli compress -i data.txt -o data.cross.bin -m crossing
```

```
python -m cli.bitpacking_cli decompress -i data.cross.bin -o data_out.txt
```

```
cmd /c fc data.txt data_out.txt
```

Résultat :

OK: 7 integers -> 3 words (k=12, mode=crossing)

OK: decompressed 7 integers (reconstructed mode: k=12, mode=crossing)

Comparaison des fichiers data.txt et DATA_OUT.TXT

FC : aucune différence trouvée

Le fichier décompressé est **identique à l'original**.

Le système fonctionne donc sans perte et la compression atteint un **gain de 62,5 %**.

8.2. Résultats du benchmark

Mode	Type de données	Gain (%)	Tcomp (ms)	Tdecomp (ms)	Tget (ms)
Crossing	Aléatoire	62.5	1.66	1.05	0.0076
Non-crossing	Aléatoire	62.5	1.64	0.89	0.0071
Overflow	Aléatoire	0.0	2.32	1.61	0.0706
Crossing	Petites valeurs	87.5	0.9881	0.6676	0.0032

8.3. Interprétation détaillée des résultats

Mode *Crossing*

- Meilleur taux de compression (**62–87 %**) grâce à l'exploitation maximale des bits disponibles.
- Très faible temps de traitement (**< 1 ms pour 1000 entiers** sur petites valeurs).
- Idéal pour les séries de valeurs proches (données homogènes).
- Gestion des décalages plus complexe mais performante.

Mode *Non-crossing*

- Performances constantes (**62,5 % de gain**) et structure simple.
- Lecture directe rapide, bonne stabilité.
- Recommandé pour les systèmes nécessitant clarté et rapidité de lecture.

Mode Overflow

- Utile pour gérer des valeurs extrêmes (hors borne).
- Aucun gain sur données aléatoires mais garantit la **robustesse du système**.
- Temps légèrement supérieur (2,32 ms) à cause de la gestion du débordement.

Analyse globale

Tous les temps de compression et décompression restent **inférieurs à 3 ms pour 1000 entiers**, ce qui montre la **haute performance du système BitPacking**.

8.4. Analyse de la latence

La **latence seuil** (t_{seuil}) indique le moment où la compression devient avantageuse :

$$t_{seuil} = \frac{T_{comp} + T_{decomp}}{(\text{Taille brute} - \text{Taille compressée})}$$

Pour le mode *Crossing* sur 1000 petites valeurs :

$$t_{seuil} = 0.00005913 \text{ ms/bit}$$

Si la latence réseau dépasse cette valeur (cas des réseaux Wi-Fi, 4G, fibre), alors la **compression devient systématiquement bénéfique**.

Cela confirme que le **BitPacking** optimise les performances même en environnement rapide.

8.5. Fonctionnement des opérations principales

Fonction	Rôle	Complexité	Particularité
compress(array)	Encodage du flux binaire	O(n)	Réduit la taille mémoire

Fonction	Rôle	Complexité Particularité	
decompress(array)	Reconstruction complète	O(n)	Sans perte
get(i)	Lecture ciblée	O(1)	Accès rapide sans décompression

Exemple d'accès direct :

```
python -m cli.overflow_cli get --input data.ovf --index 3
```

Résultat : 3

Cette fonctionnalité permet une **lecture rapide d'un élément précis**, sans décompresser l'ensemble du fichier, ce qui est idéal pour les applications Big Data ou les systèmes embarqués.

8.6. Types de données testées

Type de données	Structure	Objectif du test	Gain moyen observé	Mode optimal
Aléatoires	Dispersée	Mesure de stabilité	62,5 %	<i>Crossing / Non-crossing</i>
Croissantes	Ordonnée	Vérifier cohérence	62,5 %	<i>Crossing</i>
Mélangées	Inhomogène	Tester débordements	0–40 %	<i>Overflow</i>
Petites valeurs	Homogène	Gain maximal	87,5 %	<i>Crossing</i>

Les résultats démontrent que :

- Le BitPacking est **très performant sur les données homogènes**.
 - Le mode *Overflow* assure la **robustesse** sur des ensembles hétérogènes.
 - Le **mode Crossing** offre le meilleur compromis global entre compacité et vitesse.
-

9. Discussion et limites

Points forts :

- Architecture modulaire claire.

- Interface CLI conviviale.
- Tests automatisés validés (14/14).
- Benchmarks cohérents et reproductibles.

Limites :

- Entiers négatifs non pris en charge.
 - Optimisation CPU (SIMD) non encore implémentée.
 - Pas encore d'interface graphique utilisateur.
-

10. Conclusion et perspectives

Le projet *BitPacking 2025* démontre l'efficacité de la **compression d'entiers** par BitPacking.

Les trois variantes implémentées offrent un équilibre entre **compacité, rapidité** et **robustesse**.

Les résultats confirment la validité théorique du modèle, avec un **gain de 87,5 %** sur de petites valeurs

et des **temps d'exécution inférieurs à la milliseconde**.

Perspectives :

- Prise en charge des entiers signés.
- Optimisation vectorielle (SIMD) et parallélisation.
- Ajout d'une interface graphique.
- Extension à d'autres formats (images, matrices).