Universite cote d'azur

l'annee scolaire 2025 - 2026

Data compressing for speed up transmission

Rapport

Module : genie logiciel

l'etudiant :

KADRI Dia-eddine

Encadré par :

Prof Jean-charles Regin

Contenu

Rapport Technique : Compression et Décompression de Tableaux d'Entiers par Bit-Packing

1. Introduction

Le projet « **Compression et décompression des tableaux d'entiers par Bit-Packing** » a pour objectif d'implémenter plusieurs méthodes de compression de tableaux d'entiers en mémoire. L'idée principale est de **réduire la quantité de mémoire nécessaire** pour stocker ces tableaux tout en conservant la possibilité d'accéder rapidement à un élément précis, sans devoir décompresser l'ensemble du tableau.
Trois variantes ont été développées pour répondre à différents besoins : **Classic**, **Strict**, et **Overflow**. Chaque mode se distingue par son algorithme de compression et son approche du stockage des bits.

2. Structure du projet

Le projet est organisé en modules clairs :

- **src/** : contient le code source des algorithmes (bitpacking.py, bitpacking_strict.py, overflow.py, factory.py).
- **scripts/** : scripts pour exécuter le projet en console (run_demo.py) ou via une interface graphique (run_gui.py).
- **benchmarks/** : scripts pour mesurer les performances et enregistrer les résultats (benchmark_bitpacking.py, utils.py, protocol.md).
- **tests/** : tests unitaires pour chaque module et pour la gestion des cas particuliers (négatifs, overflow).
- **Makefile** : automatisation des tâches (exécution, tests, nettoyage).
- **requirements.txt** : dépendances Python.

Cette organisation permet une **séparation claire des responsabilités**, facilitant la maintenance, l'extension et la validation des algorithmes.

3. Algorithmes de compression et décompression

Principe général du Bit-Packing

Le **Bit-Packing** consiste à stocker plusieurs entiers dans un même mot de 32 bits, en utilisant uniquement le nombre de bits nécessaires pour représenter chaque entier.
Le calcul des bits nécessaires se fait par la fonction _compute_bits_per_int, qui prend en compte le maximum et le minimum du tableau pour gérer les entiers négatifs via un offset.
Le but est de réduire la mémoire utilisée tout en permettant un **accès direct rapide** (getint) à n'importe quel élément du tableau compressé.

4. Modes de compression

4.1 BitPackingClassic

Le mode **Classic** est la version la plus simple de Bit-Packing.

Principe :

- Chaque entier est stocké dans un bloc de 32 bits sans chevauchement avec les entiers suivants.
- Les valeurs négatives sont gérées avec un offset, qui décale tous les éléments pour qu'ils soient positifs.
- La décompression consiste à lire successivement chaque entier depuis les blocs en appliquant l'offset inverse.

Algorithme :

- Parcourir le tableau et appliquer l'offset si nécessaire.
- Ajouter chaque entier dans un accumulateur de 32 bits.
- Lorsque le bloc est rempli, le stocker dans le tableau compressé et recommencer.
- À la décompression, lire bloc par bloc et récupérer chaque entier par un simple décalage et un masque de bits.

Pourquoi ce choix :

- Simplicité et rapidité d'exécution.
- Le compromis vitesse/taux de compression est acceptable pour des tableaux uniformes ou petits.
- Limité par le fait que des blocs partiellement remplis entraînent une perte d'espace mémoire.

4.2 BitPackingStrict

Le mode **Strict** optimise le taux de compression grâce à l'**overlap**, c'est-à-dire que les entiers peuvent chevaucher plusieurs blocs de 32 bits si nécessaire.

Principe :

- Les entiers sont stockés dans un flux continu de bits.
- Si un entier ne tient pas entièrement dans le bloc courant, il est partiellement stocké dans le bloc suivant.
- Les entiers négatifs sont également gérés par un offset.

Algorithme :

- Calculer le nombre de bits nécessaires pour chaque entier avec _compute_bits_per_int.
- Stocker chaque entier dans un accumulateur en continu, en gérant le chevauchement si bits_in_accu >= 32.
- À la décompression, reconstruire les entiers en lisant les bits successifs et en appliquant l'offset inverse.

Pourquoi ce choix :

- Permet un **taux de compression maximal**, surtout pour les tableaux denses.
- Plus complexe que Classic, mais nécessaire pour minimiser l'espace mémoire.
- La gestion du chevauchement est la principale difficulté lors de la lecture et de l'accès direct.

4.3 BitPackingOverflow

Le mode **Overflow** est conçu pour gérer des tableaux contenant des valeurs très grandes ou hétérogènes.

Principe :

- Un **seuil automatique (threshold)** est calculé pour identifier les valeurs "grandes" qui ne peuvent pas tenir dans le nombre de bits standard.
- Les valeurs normales sont stockées dans packed.
- Les valeurs trop grandes sont stockées dans un tableau overflow.
    - Un affichage x-y indique si un élément est normal (0-x) ou dans l'overflow (1-index).

Algorithme :

- Parcourir le tableau : si abs(x) <= threshold, stocker normalement ; sinon, placer dans overflow et stocker la valeur seuil dans packed.
- À la décompression, reconstruire les valeurs depuis packed, en remplaçant les éléments de seuil par ceux de overflow.

Pourquoi ce choix :

- Permet de **gérer des tableaux hétérogènes** sans perte de données.
- Pratique pour les applications où certaines valeurs peuvent être exceptionnellement grandes.
- Complexité légèrement supérieure en raison de la double structure packed + overflow.

5. Gestion des valeurs négatives

Toutes les variantes (Classic et Strict) utilisent un **offset automatique** :

- Calculer le minimum du tableau.
- Ajouter l'opposé du minimum à tous les éléments pour les rendre positifs avant compression.
- Appliquer l'inverse lors de la décompression.

Difficulté rencontrée :

- Assurer que l'offset est correctement appliqué pour tous les accès directs (getint).
- Garantir que le nombre de bits calculé couvre aussi bien les valeurs négatives que positives.

6. Tests unitaires

Des tests ont été réalisés pour **chaque mode** :

- Vérification que la **décompression restitue exactement le tableau original**.
- Tests d'accès direct via getint pour chaque index.
- Tests pour le cas des **négatifs** (avec offset).
- Pour Overflow : vérification que les éléments stockés dans overflow sont correctement récupérés.

Les tests permettent de valider la **fidélité** et la **robustesse** des algorithmes.

7. Interface graphique et console

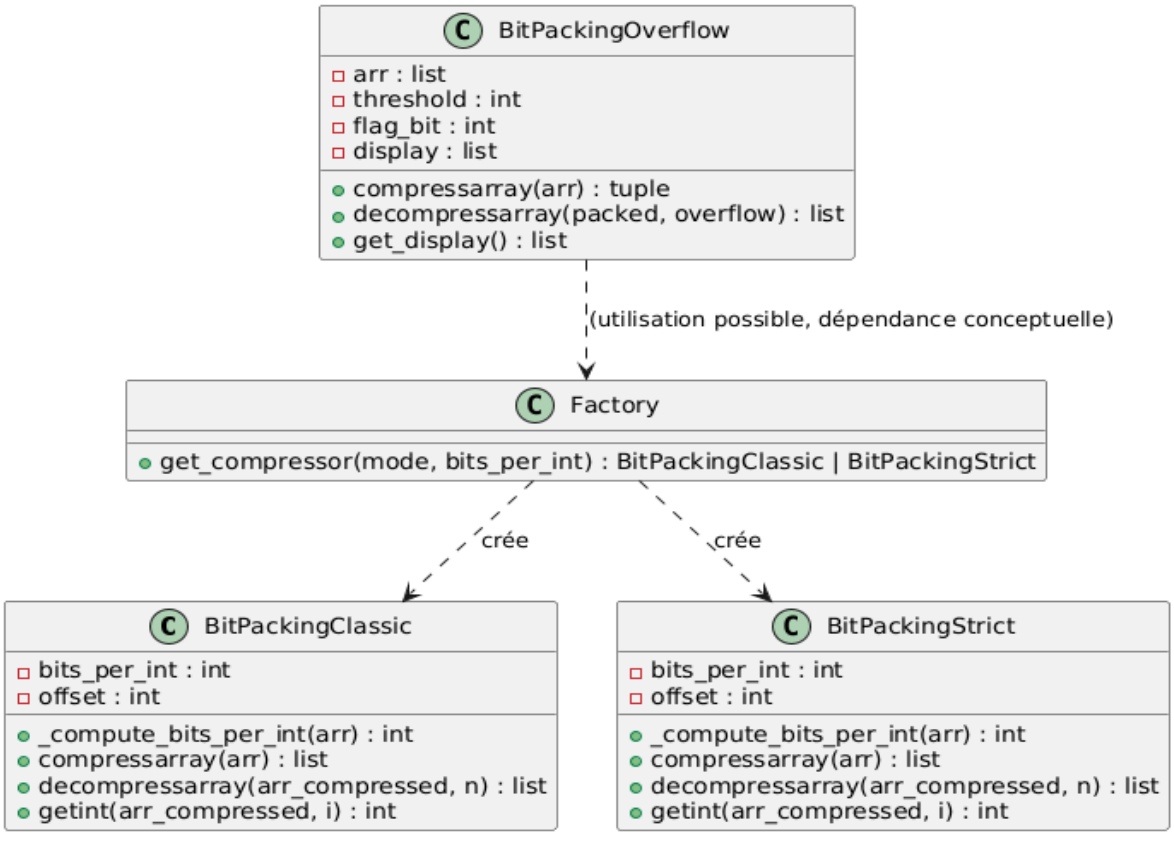
7.1 Interface console (run_demo.py)

- Entrée manuelle d'un tableau d'entiers.
- Choix du mode de compression.
- Affichage du tableau compressé, du temps de compression/décompression et du ratio de compression.

7.2 Interface graphique (run_gui.py)

- Réalisée avec **Tkinter**.
- Permet la saisie interactive et la visualisation des résultats.
- Inclut un accès interactif à un élément spécifique du tableau compressé.

8. Diagramme UML



9. Conclusion

Le projet illustre plusieurs concepts essentiels :

- Optimisation mémoire via le Bit-Packing.
- Différents compromis entre compression, vitesse et complexité (Classic vs Strict vs Overflow).
- Gestion des valeurs négatives et des débordements.
- Tests unitaires rigoureux assurant la fidélité et la robustesse.
- Interfaces console et graphique pour expérimenter facilement avec différents tableaux.

Ce projet fournit ainsi un cadre modulaire, performant et pédagogique pour la compression de tableaux d'entiers.

Remarque : pour l'exécution et l'utilisation du projet, se référer au fichier README.md.