Logiciel de compression de données en Java – Bit Packing

Samuel Premel
Master 1 Informatique – Université de Nice
Année universitaire 2025–2026

Table des matières

Introduction

Le bit packing est une méthode de compression sans perte consistant à utiliser efficacement les 32 bits disponibles dans un int afin de stocker plusieurs valeurs dans un même mot mémoire. Cette technique repose sur un ensemble d'opérations binaires fondamentales : décalages (shift), masquage (mask), et combinaison (or). En choisissant un nombre optimal de bits k pour représenter les valeurs, on peut réduire considérablement la taille totale du tableau stocké.

L'algorithme se deroule en shiftant l'int a sa bonne place en combinant les different int avec or pusi en reduisant leurs taille avec un mask notons qu'il existe des version trés optimisé et populaire de cette algorithm capables d'atteindre plusieurs milliards d'opérations par seconde.

Dans ce projet, nous avons réalisé une implémentation non vectorisée et modulaire du bit packing en Java, comprenant trois variantes : NoCrossBitPacking, CrossBitPacking et OverflowBitPacking.

Environnement et dépendances

Le projet a été développé sous Windows 11 à l'aide de Visual Studio Code, du système de build Maven, et de l'environnement OpenJDK 17. La structure Maven standard a été respectée, avec séparation entre les sources et les tests. L'exécution du projet s'effectue par :

mvn clean package
java -jar target/software_project-1.0-SNAPSHOT.jar benchmark 16bit

L'application supporte plusieurs modes : benchmark, compress et decompress

Architecture logicielle

Les module:

Le projet contient 4 module:

1. **Le module de compression (`bitpacking`)**, qui contient le cœur du projet et implémente les différentes stratégies de bit packing.
2. **Le module d'utilitaires (`utils`)**, dédié aux opérations binaires de bas niveau et aux fonctions d'aide.
3. **Le module de tests et de mesure (`benchmark`)**, qui évalue les performances et la validité des algorithmes.

4. **Le module principal (`main`)**, chargé de l'interprétation des commandes et de l'exécution globale du programme.

L'organisation du code suit une logique hiérarchique : les couches les plus basses (opérations binaires) servent de fondation aux couches intermédiaires (algorithmes de compression), elles-mêmes contrôlées par des modules de plus haut niveau (benchmark, interface principale).

Le module **bitpacking** constitue la couche centrale. Il définit l'interface commune à tous les algorithmes, ainsi qu'une classe abstraite qui en formalise la structure générale. Trois implémentations concrètes héritent de cette classe :

- **NoCrossBitPacking**, la version la plus simple et la plus rapide, qui compresse sans franchir les frontières des mots de 32 bits ;
- **CrossBitPacking**, qui autorise les valeurs à déborder d'un mot à l'autre, augmentant ainsi la densité de compression mais au prix d'un surcoût de calcul ;
- **OverflowBitPacking**, qui combine une compression standard avec une gestion dynamique des valeurs exceptionnelles dépassant la taille prévue, inspirée des algorithmes Pfor.

Le module **utils**, quant à lui, regroupe des fonctions génériques de manipulation de bits

Le module **benchmark** joue un rôle de supervision et d'évaluation. Il fournit une structure expérimentale permettant de comparer les trois stratégies selon plusieurs critères : temps moyen de compression et de décompression, vitesse d'accès aux valeurs (via la méthode `get`), et ratio de compression.

Enfin, le module **principal**, qui contient la classe `Main`, gère la communication entre l'utilisateur et le programme. Il interprète les arguments passés en ligne de commande, sélectionne le mode d'exécution (benchmark, compression ou décompression), et fait appel à la fabrique pour créer la bonne stratégie de compression. Ce module assure donc le lien entre la logique applicative et l'utilisateur final.

Les patterns de conception

Trois **design patterns** structurent la logique interne du projet, chacun répondant à un besoin de flexibilité différent.

Le premier est le **pattern Template Method**. Il est definit comme un pattern qui permet de coder un squelette général que different algorithme simillaire peuvent implementer Il est donc parfait pour definie le corps du BitPackign que les different algorithme pourront implemnter il permet entre autre d'avoir la meme racine pour les different algorithme

evitant la duplication de code et permettant d'en ajouter de nouveau plus facilement. Il gére le corps de l'algoirthme il est de niveau bas

Le deuxième est le **pattern Strategy**. Il repose sur une interface commune à toutes les implémentations et permet de sélectionner dynamiquement l'algorithme de compression à utiliser. Il permet d'implementer facilement de nouveau algorithme dans la factory il gére les different algorithme (il est de niveau intermediaire)

Enfin, le **pattern Factory Method** complète les deux précédents en automatisant la création des objets. Il centralise le choix et l'instanciation des différentes stratégies de compression. Il permet au client de ne pas avoir besoin de savoir l'implementation du bitpacking et de tout gerer avec un seul arguemnt, il gére l'interaction entre le code client et le code modele ainsi il est de haut niveau

Explication détaillée de l'implémentation

L'implémentation repose principalement sur le module **bitpacking**, qui constitue le cœur du projet. C'est dans ce module que sont définis l'interface commune, le squelette général de l'algorithme et les trois variantes concrètes. L'objectif général a été de séparer ce qui est commun à tous les algorithmes (lecture/écriture séquentielle, choix de la largeur de bit, stockage du nombre d'éléments) de ce qui diffère (gestion du dépassement de mot, gestion d'un tableau d'overflow, calcul du masque, etc.).

1. Interface commune

La première étape a été de définir une **interface unique** que toutes les implémentations doivent respecter. Cette interface impose les trois opérations demandées dans le sujet : Cela permet d'imposer la meme implementation pour tout les algorithm

2. Squelette de l'algorithme : `AbstractBitPacking`

Plutôt que de réécrire la même logique de compression dans chaque classe, une classe abstraite a été écrite pour **définir le déroulement général**. Cette classe fait trois choses importantes :

1. **Choix de la taille en bits (**k**)** : cette implementation sera detaillédans BitUtils
2. **Initialisation de l'état** : avant d'entrer dans la boucle de compression, certaines variables sont calculées une seule fois (masque, nombre de valeurs par mot, position courante en bits, etc.). L'idée est d'**éviter de refaire le même calcul dans la boucle chaude**, parce que c'est là qu'on passe le plus de temps.

3. **Boucle de compression** : La boucle chaude elle correspond a la fonction writeOne et Readone elle doiventetre particuliérement optimisez

La décompression suit exactement la même idée. Notons que pour l'optimisez sont implementation est sequentielle ce qui fait qu'un autre algorithme est utilisé pour get(int I)

4. Les trois variantes

a. NoCrossBitPacking

Cette variante est la plus directe : on découpe le mot de 32 bits en cases de taille `k`, et **on garantit qu'une valeur ne traverse jamais deux mots**. C'est ce qui la rend rapide : il y a très peu de cas particuliers à gérer.

b. CrossBitPacking

Cette version est plus lente, : à chaque écriture, il faut **vérifier s'il y a débordement** sur le mot suivant. Si oui, il faut écrire une partie de la valeur dans le mot courant et le reste dans le mot suivant. Ce simple test ajoute une branche conditionnelle dans la boucle chaude, ce qui suffit à faire baisser les performances. Cependant elle est bien plus simple a comprendre au niveau de l'algorithme

c. OverflowBitPacking

Clairement la partie la plus complex du projet . Pour cette raison j'ai choisie de l'implementer en cross même si celle si est plus lente Ici, le problème est que certaines valeurs ne tiennent pas dans `k` bits. Plutôt que d'augmenter `k` pour tout le monde (ce qui annule la compression), **on garde** `k` **pour la majorité**, et **on met les valeurs trop grandes dans une zone à part**.

Cela complexfie l'algorithme nous somme obliger de:

- **faire un premier passage** pour compter combien il y aura de valeurs en overflow ;
- **gérer un tableau séparé** (nous choissions `ArrayList<Integer>` parce que c'est ce qu'il y a de plus simple en Java pour grandir dynamiquement) ;
- **fusionner** à la fin ces données d'overflow avec le reste des données compressées (c'est là qu'intervient un éventuel `finalizeCompress`).

à cause de cette complexité, je n'ais pas pu faire une **lecture séquentielle optimisée** aussi simple que dans les autres cas. Du coup, la version overflow **recalcule plus souvent** et donc **décompresse un peu moins vite** .

5. La factory et le main

La partie "haut niveau" est volontairement restée simple. La **factory** ne fait qu'une chose : en fonction d'un mot-clé passé en paramètre (`"cross"`, `"overflow"`, `"nocross"`), elle

renvoie l'objet qui implémente la bonne stratégie.
Le **main**l lit les arguments, il affiche une erreur si jamais l'utilisateur ne donne pas le bon nombre de paramètres, et il appelle la bonne méthode. Il est donc particulierement ininteressant.

6. Les utilitaires et le choix de k

`BitUtils` est la seule vraie classe utilitaire du projet. Elle sert surtout à deux choses :

1. **compter le nombre de bits nécessaires** pour stocker un entier donné ;
2. **décider d'une valeur de** `k` à partir des données d'entrée.

pour la version sans overflow, le choix est basique : on prend le nombre de bits du maximum.
Pour la version overflow, L'heuristique choisie est la suivant : prendre une taille qui couvre environ **95 % des valeurs**. C'est unbe approche qui a été implementé dans les principaux algorithme de pfor (nom academique de l'overflow) notons que les implementation moderne ont une heuristique plus complex

7. Benchmark

Enfin, le module de **benchmark** n'est pas là que pour la performance, mais aussi pour **valider** que les trois variantes produisent bien le même tableau après décompression. Nous utilisons `System.nanoTime()` qui est recommandé en Java pour ce genre de mesure.
Nous le startons avant le compress et fermons directement aprés ce protocole permet d'avoir des données précise pour toute les implementation.
Nous teston **15 bits** et **16 bits** pour montrer la différence entre une taille qui ne divise pas 32 (15) et une qui le divise (16). Aussi nous avons implementer une version avec overflow.

Résultats expérimentaux

Les benchmarks ont été réalisés sur un tableau de 100 000 entiers aléatoires. Les valeurs ci-dessous sont des moyennes sur 10 exécutions. Avec ces donné il est difficile de dire quand est ce que l'utilisation du bitpacking devient néssécaire de plus la mauvaise optimisation de Overflow traduit mal la réalité de ces algorithme, nous pouvons quand meme dire que nous devrions toujous utilisé Nocross (sauf en overflow ou l'utilisation depend de ce que nous preferons entre le temps de compression et decompression entre cross et overflow)

l'implementation deviens rentable quand le temps de compression + temps de compression est superieur a ration de compression*debit(en 100000 int par nanoseconde)

Mode de test	Méthode	Temps de compression (ns)	Temps de décompression (ns)	Temps d'accès complet (`get`) (ns)	Taille compressée (ints)	Ratio de compression
Overflow	**NoCross**	748 410	535 260	969 520	100 002	100,00 %
	Cross	886 840	990 960	3 057 380	65 627	65,63 %
	Overflow	2 249 170	679 210	2 340 980	67 030	67,03 %
15 bits	**NoCross**	763 140	594 020	967 840	50 002	50,00 %
	Cross	749 709	966 900	2 936 340	50 002	50,00 %
	Overflow	2 285 730	454 010	2 312 210	53 128	53,13 %
16 bits	**NoCross**	726 780	563 900	1 010 110	50 002	50,00 %
	Cross	794 449	991 629	2 996 610	50 002	50,00 %
	Overflow	2 587 049	435 150	2 126 860	53 128	53,13 %

Discussion critique

Le projet fonctionne correctement et respecte les contraintes du sujet. Cependant, plusieurs points sont perfectibles, notamment l'optimisation de OverflowBitPacking (ArrayList<Integer> coûteux), en plus que l'implementation d'un readOne sequentielle, aussi une vectorisation du bitpacking rendrait l'operation vraimment rapide et intéressante , Mon module de benchmarking et mon main sont peu satisfaisant meme si il permettent d'effectuer des teste sur le terminal il faudrait surement permettre d'implementer des benchmarking sans autant toucher au code Aussi il n'est pas possible de mettre le tableau de sortie dans un fichier texte avec > car des ligne indiquand le temps ce mettent on est obligé de supprimer manuellement ces ligne enfin uncompress depend de l'algorithme utilisé et je n'ai pas de moyen de savoir ça il aurait fallue ajouter un flag. Aussi les int négatifs ("bonus") sont pas implementé dans le sujet au début j'ai essayé de les implementé (utilisation de >>>) mais je n'ai pas reussie cela demande une comprehénsion des bits encors plus fine

Références

- Lemire, D. (2012). Fast Integer Compression .https://arxiv.org/pdf/1209.2137
- Cornell University. Bit Packing Lecture Notes.
https://www.cs.cornell.edu/courses/cs3410/2024fa/notes/bitpack.html
- Refactoring Guru. Design Patterns Reference. https://refactoring.guru/design-patterns

-What is BitMaskign https://www.geeksforgeeks.org/c/c-bitmasking/

-How fast is bitpacking https://lemire.me/blog/2012/03/06/how-fast-is-bit-packing/

-Loop Optimization https://osmyasal.medium.com/loop-optimizations-38f9338ef246

-Pfor implementation https://ir.cwi.nl/pub/15564/15564B.pdf

- Oracle. Java 17 Documentation. https://docs.oracle.com/en/java/javase/17/