Software Engineering Compression Project - Nathan THIERY

Nathan Thiery

November 2025

Contents

1 Introduction

Ce rapport a pour but principal de détailler les **différents problèmes, solutions et stratégies employées** lors du développement de ce projet individuel de compression d'entiers.

Le langage de programmation utilisé est **Java**, et **aucune dépendance** n'est à inclure, mis à part les imports compris dans java.*.
Pour plus d'informations sur la compilation et l'utilisation du programme, se référer au fichier README.md disponible sur le GitHub.

2 Planification & organisation

Avant même de commencer à coder quoi que ce soit, la première étape a été de réfléchir à une structure adaptée aux contraintes des différents algorithmes de compression.
L'idée générale était d'utiliser des design patterns connus et fiables pour assurer les différentes fonctionnalités citées ci-dessous.
Chaque design pattern apporte indéniablement des avantages, mais en combiner un trop grand nombre dans ce genre de projet relativement petit peut parfois aussi complexifier inutilement la structure du programme si l'on part dans l'optique d'utiliser un maximum de design patterns sans regarder la cohérence de ceux-ci avec le projet.

2.1 Choix des Design Patterns en fonction d'une stratégie

Dans cet état d'esprit, j'ai essayé de me représenter des cas d'utilisation réels d'algorithmes de compression ainsi que les spécificités qu'ils devraient comporter dans un réel cas d'usage pour être le plus efficace possible.
Voici, au final, les points majeurs que j'ai pu imaginer :

2.1.1 Réflexions sur la portabilité de la donnée

Dans un cas réel, la compression de la donnée (ici, des entiers) se déroule majoritairement côté serveur (exemple : services de streaming ou de live-streaming comme Netflix, Twitch ou encore des plateformes de streaming audio comme Spotify).

La donnée est donc compressée côté serveur via différents formats (.264, H.265, AV1, VP9...), puis envoyée par paquets au client qui la décompresse de son côté (souvent via un lecteur vidéo du navigateur).
Le principe de cette démarche est d'assurer une bande passante et une latence minimisées par rapport à un usage plus classique.

2.1.2 Implications de la gestion des communications entre appareils

D'un point de vue conception et distribution des classes, j'ai interprété cet usage de la manière suivante :
Si le client et le serveur, ou de manière plus générale deux entités communiquantes, possèdent tous deux les algorithmes de compression, décompression et de manipulation de la donnée (ce qui est généralement le cas, par exemple dans les protocoles utilisés sur nos machines), la seule partie à isoler et rendre indépendante est donc la donnée elle-même.
Plutôt que de stocker les entiers compressés directement dans les classes de compression (ex. Compression1 possède un attribut donnée et les méthodes de

compression/décompression de l'algorithme de compression 1, et ainsi de suite pour les algorithmes 2 et 3), j'ai créé ici une classe propre à la donnée, nommée *IntegerArray*, qui se compose de deux attributs privés et de getters et setters pour y accéder.
De cette manière, je fais en sorte de pouvoir, pour une donnée, compresser ou décompresser ses entiers sans se soucier du type d'algorithme utilisé.
On a donc une classe pour chaque type d'algorithme de compression et une classe spécifique pour la donnée.

J'ai pensé cette approche traduisible par plusieurs design patterns existants, comme le Design Pattern *Strategy*, qui me permettrait de définir plusieurs algorithmes interchangeables de compression/décompression et les principales méthodes qu'ils doivent implémenter, ainsi que le Design Pattern *Factory*, qui me permettrait de créer dynamiquement la classe en fonction du type d'algorithme utilisé à un moment donné.
Pour une meilleure gestion de la mémoire et pour éviter de créer des instances multiples d'un même algorithme de compression, j'ai utilisé la notion du Design Pattern *Singleton* (une seule instance déclarable pour une classe), gérée par la *CompressionFactory*.

2.2 Diagramme des classes & relations entre les classes

Avec ces premières réflexions en tête, j'ai pu mettre en place ce diagramme de classes des différents éléments à implémenter dans mon programme :

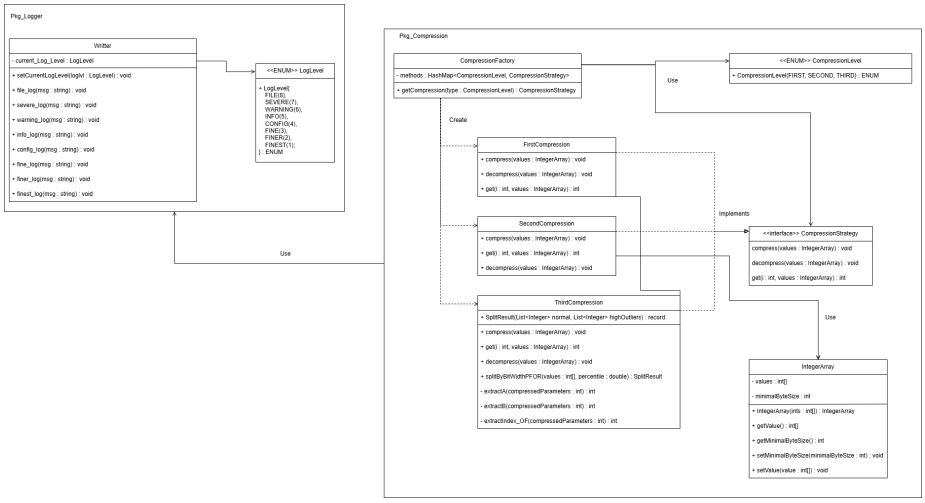

Figure 1 : Diagramme de classes

3 Développement, difficultés & solutions

Certaines parties du projet étaient difficilement anticipables sans les connaître au préalable. Ainsi, je suis tombé à plusieurs reprises sur des imprévus pour lesquels j'ai dû improviser et trouver une solution sans compromettre la structure et la stratégie initialement prévues.

3.1 First compression method

Le premier algorithme de compression a été assez simple à mettre en place, une fois son fonctionnement analysé (*Annexe, Figure 1*).
C'est sur ce premier algorithme que j'ai établi la composition de ma classe *IntegerArray*.
Cette classe me sert à stocker les entiers, sous forme compressée ou non, dans un attribut privé *value (int[])*.
Un autre attribut, nommé *minimalByteSize*, représente le nombre minimal de bits utilisés pour représenter un nombre dans le tableau des entiers compressés. On est ici obligés de le garder dans un attribut pour pouvoir correctement décompresser les valeurs contenues dans le tableau.
On s'en servira parallèlement pour connaître la forme du contenu du tableau d'entiers (savoir si les valeurs sont actuellement compressées ou non). Lorsque *minimalByteSize* vaut -1, les valeurs ne sont pas compressées, sinon elles le sont.

3.2 Second compression method

Le second algorithme ne m'a pas personnellement ajouté de difficultés.
Le fonctionnement général reste identique ; la seule partie à modifier concerne la manière de pré-calculer les paquets de 32 bits nécessaires pour stocker les données lors de la compression et le nombre de valeurs contenues dans le tableau lorsqu'elles sont compressées.

3.3 Third compression method

Le troisième algorithme a posé un peu plus de problèmes, car avec l'introduction d'une zone d'Overflow, il me fallait désormais trouver un moyen de stocker deux tailles différentes d'entiers compressés dans ma structure *IntegerArray*.
Un peu plus tard, je me suis aussi rendu compte qu'il me faudrait également l'indice auquel la zone d'overflow commence, car je n'ai pas réussi à trouver de moyen de le calculer autrement (si cela est possible).
J'ai donc fait comme pour la première méthode, en détaillant sur une note les moyens possibles d'utiliser la troisième méthode de compression sans changer ma structure d'*IntegerArray* (donc avec mon *value : int[]* pour le tableau d'entiers et uniquement mon *minimalByteSize : int* en plus du tableau).
Les notes m'ont aidé à comprendre l'ensemble des éléments ainsi que la manière de procéder pour stocker les trois informations dans *minimalByteSize* (*Annexe, Figure 2*).

Ces trois informations sont :

- le nombre de bits pour représenter un entier de la première zone ;
- le nombre de bits pour représenter un entier de la deuxième zone ;
- l'indice du commencement de la zone d'Overflow.

Pour résumer mes notes de l'*Annexe Figure 2*, qui détaillent ce processus, ces trois informations peuvent toutes tenir dans *minimalByteSize* si l'on prédéfinit au préalable des zones dédiées.
Ainsi, on aurait besoin uniquement de 2 fois 6 bits pour les deux nombres de bits des deux zones du tableau, car 6 bits permettent une représentation maximale allant jusqu'à 63, et un entier en Java ne mesurant que 32 bits, cela est donc plus que suffisant.

Cela nous laisserait donc une zone de (32 - 12) = 20 bits pour stocker l'indice de la zone d'overflow, nous permettant d'aller jusqu'à un indice potentiel de 1 048 575 (calculé par $2^{19} \times 2 - 1$).
Au final, je suis donc parvenu à utiliser ce moyen pour conserver la même structure pour l'algorithme 3.
Les méthodes *extractA*, *extractB* et *extractIndex_OF* de ma classe *ThirdCompression* servent ainsi à récupérer ces éléments stockés dans l'entier *minimalByteSize* de la classe *IntegerArray* lorsque l'on cherche à décompresser un tableau d'entiers avec l'algorithme 3.

Les noms des variables internes à cette méthode sont identiques à ceux des notes de l'*Annexe Figure 2*.

4 Tests et déduction des cas d'utilisation des méthodes

Pour me permettre de tester correctement les trois algorithmes, j'ai mis en place dans le *Main* plusieurs moyens d'initialiser aléatoirement le tableau d'entiers à compresser.
Au total, six manières différentes ont été créées :

1. Génère des entiers aléatoires entre 0 et 100 ;
2. Génère des entiers aléatoires entre 100 000 et 1 000 000 ;
3. Dans le tableau de taille x, 95% de chance de générer un entier entre 0 et 100 contre 5% de chance de générer un entier entre 100 000 et 1 000 000 pour chaque x_i ;
4. Dans le tableau de taille x, 5% de chance de générer un entier entre 0 et 100 contre 95% de chance de générer un entier entre 100 000 et 1 000 000 pour chaque x_i ;
5. Génère des entiers aléatoires entre 0 et 1 000 000 ;
6. Dans le tableau de taille x, un pourcentage aléatoire de chance de générer un entier entre 0 et 100, et donc le reste des pourcentages de chance de générer un entier entre 100 000 et 1 000 000 pour chaque x_i.

J'ai pu tester et générer des fichiers texte de sortie grâce à ces méthodes d'initialisation que l'on verra par la suite et qui nous permettront d'établir les principaux points forts, faiblesses et axes d'amélioration des algorithmes mis en place.
Il est également important de préciser que le temps indiqué dans les résultats ci-dessous comprend la compression **et** la décompression du tableau d'entiers.

4.1 Tests & graphiques

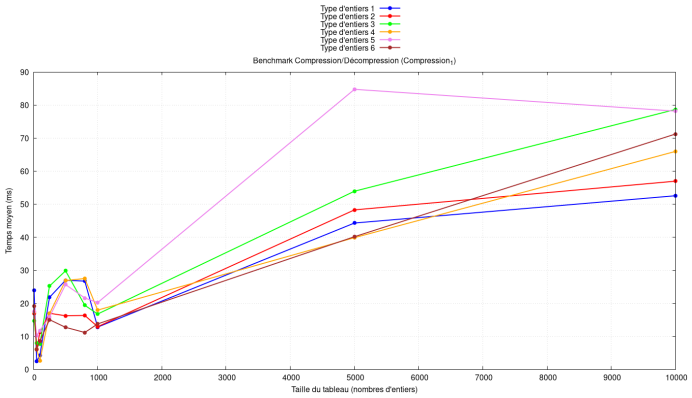

Figure 2 : Résultat des temps de calcul de l'algorithme 1 en fonction des entiers contenus

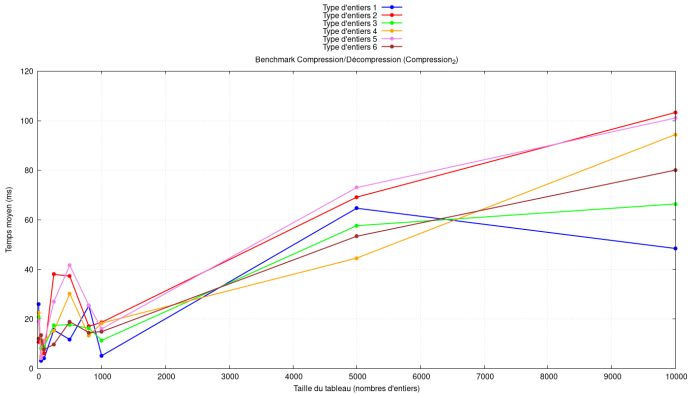

Figure 3 : Résultat des temps de calcul de l'algorithme 2 en fonction des entiers contenus

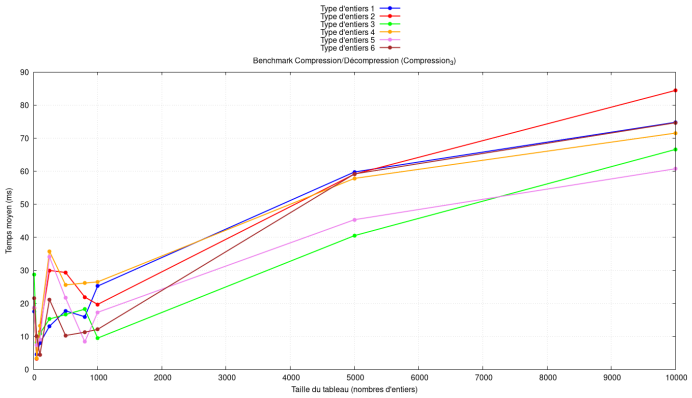

Figure 4 : Résultat des temps de calcul de l'algorithme 3 en fonction des entiers contenus

4.2 Analyse des résultats

4.2.1 Premier algorithme

On constate sur le graphe de la Figure 2 que le temps de traitement des tableaux d'entiers compressés est linéaire pour les types d'entiers dont les écarts peuvent parfois être importants (types d'entiers 3, 4 et 6), alors que les types d'entiers dont les valeurs sont plus uniformes (types 1, 2 et 5) semblent avoir un temps d'exécution légèrement plus faible lorsque la taille du tableau augmente.

Le temps d'exécution général est relativement court : c'est un algorithme fiable et prévisible, assez passe-partout. Sa seule faiblesse réside dans son manque de spécialisation, qui l'empêche d'être réellement optimisé pour un type de donnée particulier.

4.2.2 Deuxième algorithme

On constate sur le graphe de la Figure 3 que l'algorithme n°2 est tout de suite beaucoup plus chronophage à l'exécution.
Cela vient principalement des cas où des bits se retrouvent non utilisés dans un entier du tableau (sur 32 bits, peut-être que seulement 27 sont utilisés si un entier est représenté sur 9 bits).
Il est donc naturellement très chronophage pour les types d'entiers de grande taille (types 2, 4 et 5).

4.2.3 Troisième algorithme

Le troisième algorithme affiche des temps d'exécution similaires au premier, mais les différents types d'entiers produisent des résultats moins disparates. Les temps d'exécution sont proches et regroupés les uns des autres.

4.3 Marge d'amélioration

Les premier et deuxième algorithmes étant assez précis dans les contraintes qu'ils imposaient, ils sont relativement peu améliorables dans le cadre de ce projet.
Pour le troisième algorithme, cependant, les consignes imposées étaient largement moins précises, notamment sur l'algorithme interne à utiliser pour séparer les nombres de la première zone de ceux de la zone d'overflow. Je pense ne pas avoir commis de grosses erreurs d'un point de vue implémentation et structuration de mes classes, mais l'algorithme que j'utilise pour scinder mes groupes de données est à mon avis trop basique et donc pas assez efficace pour exploiter pleinement le potentiel de cette troisième méthode de compression.
D'un autre côté, utiliser un algorithme de détection d'entiers anormaux plus efficace rendrait probablement le temps d'exécution plus long, mais allégerait potentiellement le poids et la vitesse du transport de la donnée.
Le type d'algorithme de séparation de données à utiliser pour cette méthode dépend donc des caractéristiques de l'environnement. Si les données sont à

transmettre sur une longue distance, avec un trafic perturbé et des capacités réseau faibles, il serait préférable de sacrifier un peu de temps de compression/décompression pour optimiser l'envoi et la réception de celles-ci (et inversement dans le cas d'une courte distance et d'un réseau performant).

5 Conclusion

Ce projet de compression d'entiers en Java m'a permis d'explorer différentes approches algorithmiques tout en appliquant des principes solides d'ingénierie logicielle.
L'usage réfléchi de design patterns comme *Strategy*, *Factory* ou *Singleton* a apporté une structure claire et adaptable au code, facilitant la gestion de plusieurs méthodes de compression.

Le développement s'est appuyé sur une démarche progressive, chaque algorithme ayant nécessité des ajustements techniques et conceptuels pour assurer la cohérence de l'ensemble.
Les tests effectués ont finalement permis d'identifier les forces et les limites de chaque méthode, révélant des différences notables en matière de temps d'exécution et d'efficacité selon la nature des données traitées.

Si le premier algorithme se distingue par sa fiabilité et sa simplicité, le second s'avère plus coûteux en ressources, tandis que le troisième offre un bon compromis mais reste perfectible dans la gestion des zones d'overflow.
Au final, les résultats mettent en évidence la complexité du choix d'un algorithme de compression, qui dépendra principalement de son contexte d'utilisation et de ses contraintes de performance.

References

[1] Démystification du GoF (Gang of Four) : Les design patterns par la pratique en JAVA - Blog ACENSI, *https://blog.acensi.fr/demystification-du-gof-gang-four-les-design-patterns-par-la-pratique-en-java/*, ACENSI, 2015.

[2] Flyweight Design Pattern in Java, *https://www.digitalocean.com/community/tutorials/flyweight-design-pattern-java*, DigitalOcean, 2022.

[3] Opérateurs Java au niveau du bit, *https://codegym.cc/fr/groups/posts/fr.10.operateurs-java-au-niveau-du-bit*, CodeGym, 2023.

[4] Understanding PFOR (Partitioned Frame of Reference) Compression: A Powerful Technique for Optimizing Integer Sequences, *https://medium.com/@sharanmadhavd/understanding-pfor-partitioned-frame-of-reference-compression-a-powerful-technique-for-4d21d20e4e1e*, Medium, 2025.

[5] Netflix compression algorithm : Bringing AV1 Streaming to Netflix Members' TVs, *https://netflixtechblog.com/bringing-av1-streaming-to-netflix-members-tvs-b7fc88e42320*, 2021.

[6] Photo and video compression pipeline in modern web applications, *https://dmitry-barabash.medium.com/photo-and-video-compression-pipeline-in-modern-web-applications-921fa2988628*, 2023.

A Annexes

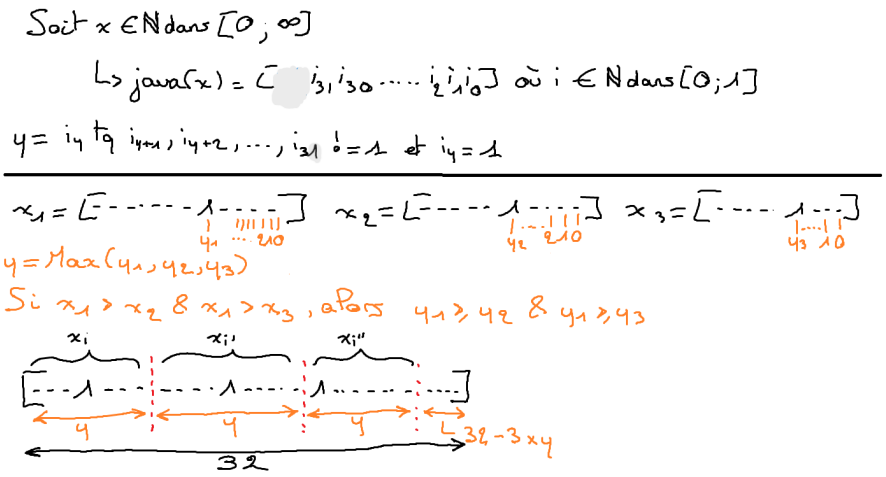

Figure 1 : Compréhension du fonctionnement de l'algorithme de compression 1

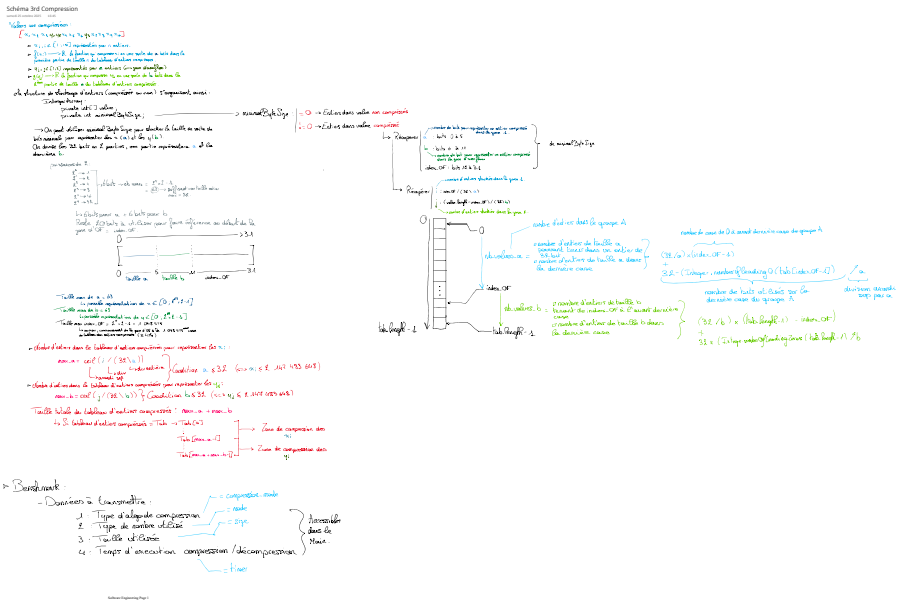

Figure 2 : Compréhension du fonctionnement de l'algorithme de compression 2 et répercussions sur la structure d'IntegerArray