Software Engineering Project

Compression de données pour accélérer la transmission

Daniel Carriba Nosrati

UE Software Engineering
Semestre 1 Master 1 Informatique
Université Côte d'Azur

2025

Sommaire

1 Introduction

Ce projet, réalisé pour l'UE "Software Engineering" du Semestre 1 du Master 1 Informatique de l'Université Côte d'Azur, est un projet sur le thème de compression de données pour accélérer la transmission.

La transmission de tableaux d'entiers est un problème majeur de l'internet. Ce projet propose une réponse à ce problème en implémentant une méthode de compression de tableaux d'entiers positifs, basés sur le nombre de bits utilisés. Cette méthode de compression est appelée "Bit Packing". Plusieurs versions de cette méthode ont été implémentés par ce projet. L'utilisateur peut ainsi compresser un tableau d'entiers positifs, ainsi que le décompresser. L'accès direct aux éléments n'est pas perdu lors de la compression, l'utilisateur peut toujours avoir un accès immédiat à un i-ème élément du tableau.

Ce projet à été réalisé en Java. Les différentes versions la méthode de compression "Bit Packing" ainsi que leurs implémentations et fonctionnements seront présentés dans ce rapport.

Ce projet donne pour résultats des benchmarks, sous forme de mesures de temps et de taux de compression, pour chaque versions de la méthode de compression. L'implémentation ainsi que la pertinence de ces benchmarks sont expliqués dans la suite de ce rapport, dans la section dédiée.

Pour finir, ce projet calcule également le temps de transmission pour une latence t où chaque version de la méthode de compression devient utile. Le fonctionnement de ceci et les conclusions à en tirées sont également expliqués dans la section dédiée, à la fin de ce rapport.

2 Bit Packing

La méthode de compression "Bit Packing" à été implémenté en trois versions différentes dans ce projet. Chaque version sera présenté en détails dans sa section dédiée.

Pour l'implémentation j'ai choisi donc de créer une classe par versions. Ces classes héritent de la classe abstraite *BitPacking*, qui elle contient les champs et méthodes communes à toutes les versions différentes de "Bit Packing". Chaque classe qui hérite donc de *BitPacking* implémente sa propre méthode pour compresser ainsi que pour accéder au i-ème élément du tableau compressé. Une méthode standard de décompression qui utilise la méthode pour accéder au i-ème élément du tableau compressé pour tout les i est également implémenté par *BitPacking*, mais peut être aussi implémenter dans les versions différentes, pour une décompression plus efficace par exemple.

Voici un schéma UML montrant une vue d'ensemble de la structure de ces classes :

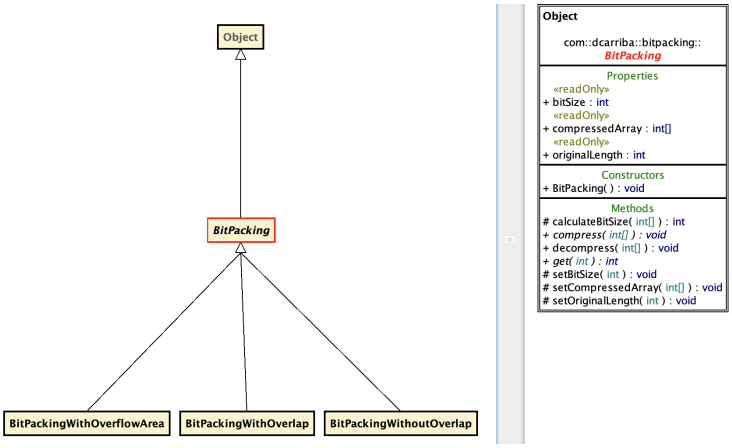

Figure 1: Schéma UML *BitPacking*

2.0.1 Implémentation commune à toutes les versions

Toutes les versions de "Bit Packing" héritent donc d'une classe abstraite *BitPacking*. Celle-ci contient les champs (i.e. les attributs) ainsi que les méthodes communes à toutes les versions différentes, soit :

- le contenu du tableau compressé
- la taille du tableau original
- le nombre de bits sur lequel chaque valeurs sera codé dans le tableau compressé

- une méthode pour calculer ce nombre de bits
- une méthode de décompression commune, qui récupère chaque élément d'index i grâce à la méthode :

  ```
  public abstract int get(int i);
  ```

  (implémenté dans chaque version)

Une classe *BitPackingTest* a également été implémenté contenant des tests unitaires pour tester le bon fonctionnement de la méthode calculant le nombre de bits sur lequel chaque valeurs sera codé dans le tableau.

2.1 Bit Packing with overlap

La première version de "Bit Packing" de ce projet est une version nommé "with overlap". Cela signifie que une valeur, du tableau à compresser, peut être compressé sur deux entiers consécutifs au sein du tableau compressé.

2.1.1 Implémentation

Cette version de "Bit Packing" a été implémenté par la classe *BitPackingWithOverlap* qui hérite de la classe abstraite *BitPacking*.

BitPackingWithOverlap implémente les méthodes abstraites de *BitPacking*, soit :

```
@Override
public void compress(int[] array) {
        ...
}
@Override
public int get(int i) {
        ...
}
```

Une classe *BitPackingWithOverlapTest* a également été implémenté contenant des tests unitaires pour tester le bon fonctionnement de la compression, décompression et accès au i-ème élément.

2.1.2 Fonctionnement

Compression : la compression fonctionne de la manière suivante :

- on calcule le nombre de bits nécessaire pour coder la plus grande valeur du tableau initial, afin de codé toutes les valeurs sur ce nombre de bits
- on donc calculer le nombre d'entiers de 32 bits nécessaire pour contenir toutes les valeurs compressé, ce qui est donc la taille du tableau compressé

- pour chaque valeur du tableau inital, on le code sur le premier entier où il peut être compressé, et si il ne peut pas être contenu entièrement sur cet entier, on continue de le compresser sur l'entier suivant.

Ainsi chaque valeur du tableau initial est codé sur N bits (i.e. le nombre de bits nécessaire pour coder la plus grande valeur du tableau initial) sur un entier du tableau compressé, ou sur deux entiers consécutifs du tableau compressé si il ne peut être contenu entièrement sur le premier entier.

Accès au i-ème élément : l'accès au i-ème élément fonctionne de la manière suivante :

- on connait le nombre de bits sur lequel toutes les valeurs ont été codées, donc il suffit de calculer sur quel entier de 32 bits la i-ème valeur à été compressé.
- on calcul la position de la i-ème valeur au sein de cet entier (de 32 bits), et on extrait la valeur, puis la renvoie. Si la i-ème valeur n'est pas entièrement compressé sur cet entier, on continu de l'extraire l'entier suivant .

Décompression : la décompression fonctionne de la manière suivante :

- pour tout valeur i il suffit d'utiliser la méthode pour accéder au i-ème élément pour la récupérer
- toutes les valeurs récupérés peuvent ainsi être rétablis dans le tableau décompressé

2.2 Bit Packing without overlap

La seconde version de "Bit Packing" de ce projet est une version nommé "without overlap". Cela signifie que une valeur, du tableau à compresser, sera toujours compressé sur un seul, et non deux, entier du tableau compressé.

2.2.1 Implémentation

Cette version de "Bit Packing" a été implémenté par la classe *BitPackingWithoutOverlap* qui hérite de la classe abstraite *BitPacking*.

BitPackingWithoutOverlap implémente les méthodes abstraites de *BitPacking*, soit :

```
@Override
public void compress(int[] array) {
        ...
}
@Override
public int get(int i) {
        ...
}
```

Une classe *BitPackingWithoutOverlapTest* a également été implémenté contenant des tests unitaires pour tester le bon fonctionnement de la compression, décompression et accès au i-ème élément.

2.2.2 Fonctionnement

Compression : la compression fonctionne de la manière suivante :

- on calcule le nombre de bits nécessaire pour coder la plus grande valeur du tableau initial, afin de codé toutes les valeurs sur ce nombre de bits
- on calcule ensuite le nombre de valeurs qui peuvent être codés sur un seul entier de 32 bits, et par conséquent la taille du tableau compressé
- pour chaque valeur du tableau inital, on le code sur le premier entier où il peut être contenu entièrement

Ainsi chaque valeur du tableau initial est codé sur N bits (i.e. le nombre de bits nécessaire pour coder la plus grande valeur du tableau initial) d'un entier (32 bits) du tableau compressé.

Accès au i-ème élément : l'accès au i-ème élément fonctionne de la manière suivante :

- on connait le nombre de bits sur lequel toutes les valeurs ont été codées, donc il suffit de calculer sur quel entier de 32 bits la i-ème valeur est situé.
- on calcul la position de la i-ème valeur au sein de cet entier (de 32 bits), et on extrait la valeur, puis la renvoie.

Décompression : la décompression fonctionne de la manière suivante :

- pour tout valeur i il suffit d'utiliser la méthode pour accéder au i-ème élément pour la récupérer
- toutes les valeurs récupérés peuvent ainsi être rétablis dans le tableau décompressé

2.3 Bit Packing with overflow areas

La troisième version de "Bit Packing" de ce projet est une version nommé "with overflow areas". Elle fonctionne de la manière suivante :

Si un nombre du tableau initial nécessite un grand nombre de bits k et que les autres nombres nécessitent seulement k' bits avec k' $<$ k, il est inutile de représenter tous les nombres avec k bits. Dans ce cas, nous pouvons attribuer une valeur spéciale à un entier compressé pour indiquer que sa valeur réelle se trouve ailleurs, à une position spécifique du tableau, appelée zone de débordement, ou "overflow area".

Par exemple, si nous voulons encoder les nombres 1, 2, 3, 1024, 4, 5 et 2048, nous pouvons encoder 1, 2, 3 et 4 sur 3 bits et les autres nombres sur 11 bits.

Nous utiliserons un bit du codage pour indiquer que nous ne représentons pas directement un nombre, mais une position dans la zone de débordement. Si 1 correspond à la zone de débordement, et si x-y signifie que le premier bit vaut x et les suivants y, nous représenterons la séquence de nombres 1, 2, 3, 1024, 4, 5, 2048 par : 0-1, 0-2, 0-3, 1-0, 0-4, 0-5, 1-1, 1024, 2048.

2.3.1 Implémentation

Cette version de "Bit Packing" a été implémenté par la classe *BitPackingWithOverflowArea* qui hérite de la classe abstraite *BitPacking*.

BitPackingWithOverflowArea implémente les méthodes abstraites de *BitPacking*, soit :

```
@Override
public void compress(int[] array) {
        ...
}
@Override
public int get(int i) {
        ...
}
```

Ainsi que une méthode de décompression plus efficace pour *BitPackingWithOverflowArea* que celle déjà existante de *BitPacking* :

```
@Override
public void decompress(int[] array) {
        ...
}
```

Une classe *BitPackingWithOverflowAreaTest* a également été implémenté contenant des tests unitaires pour tester le bon fonctionnement de la compression, décompression et accès au i-ème élément.

2.3.2 Fonctionnement

Compression : la compression fonctionne de la manière suivante :

- on calcule N le nombre de bits nécessaire pour coder la plus grande valeur du tableau initial. Toute valeur qui doit réellement être codé sur N bits sera une valeur compressé dans la "overflow area", et les autres valeurs, qui peuvent être codés sur N' $<$ N bits, seront compressé sur N' bits de manière normale.

- pour toutes les valeurs du tableau initial, on compresse dont soit la valeur sur N' bit (en ajoutant un bit 0 supplémentaire avant), soit on compresse l'index de la valeur dans l' "overflow area" (en ajoutant un bit 1 supplémentaire

avant). A la fin, toutes les valeurs de l' "overflow area", i.e. les valeurs compressé sur N bits, seront ajoutés à la fin du tableau compressé.

Ainsi les grandes valeurs (i.e. celles qui doivent être compressé sur N bits) seront compressé dans l' "overflow area", à la fin du tableau compressé, et avant dans le tableau compressé sera seulement une référence (de forme 1-overflowValueIndex) à sa position au sein de l' "overflow area".

Décompression : la décompression fonctionne de la manière suivante :

- on parcours le tableau compressé de bit en bit.
- Si le bit indicateur est 0 alors cela signifie que la valeur a été compressé de manière normale sur N' bits. On peut donc facilement la décompresser et ajouter au tableau de résultat.
- Si le bit indicateur est 1 alors cela signifie que la valeur a été compressé dans l' "overflow area". On sauvegarde donc l'indice actuel où la valeur devrait être, pour compléter l'indice à la fin lors de la décompression de la valeur de l' "overflow area"

Accès au i-ème élément : l'accès au i-ème élément fonctionne de la manière suivante :

- pour pouvoir déterminer si la i-ème valeur a été compressé de manière normale sur N' bits, ou dans l' "overflow area" sur N bits, il est nécessaire de parcourir le tableau compressé de bit en bit pour lire les bits indicateurs (0, ou 1 si la valeur est compressé dans l' "overflow area"). Ce procédé revient exactement à faire une décompression, comme l'entièreté du tableau compressé, pour notamment extraire les valeurs des éléments compressé dans l' "overflow area".
- ainsi on décompresse temporairement le tableau pour pouvoir donné la valeur du i-ème élément.

3 Tests Unitaires

3.1 Implémentation

Pour vérifier le bon fonctionnement des méthodes pour compresser, décompresser at accéder au i-ème élément du tableau compressé, on a implémenter des tests unitaires pour les classes de "Bit Packing". Ces tests unitaires sont situés dans le répertoire :

```
./src/test/java/
```

Les tests unitaires implémentés sont :

- *BitPackingTest* contient les tests unitaires pour *BitPacking*
- *BitPackingWithOverlapTest* contient les tests unitaires pour *BitPackingWithOverlap*
- *BitPackingWithoutOverlapTest* contient les tests unitaires pour *BitPackingWithoutOverlap*
- *BitPackingWithOverflowAreaTest* contient les tests unitaires pour *BitPackingWithOverflowArea*

BitPackingWithOverlapTest, *BitPackingWithoutOverlapTest* et *BitPackingWithOverflowAreaTest* étendent la classe *BitPackingVersionsBaseTest* qui contient la logique des tests unitaire, commune pour ces 3 versions de "Bit Packing".

3.2 Visualisation des résultats des tests

Les tests unitaires sont exécutés avec la commande suivante :

```
./gradlew test
```

Si "BUILD SUCCESSFUL" s'affiche, alors tout les tests unitaires on été exécutés avec succès.

Une visualisation sur l'ensemble des résultats est également disponible sur :

```
./build/reports/tests/test/index.html
```

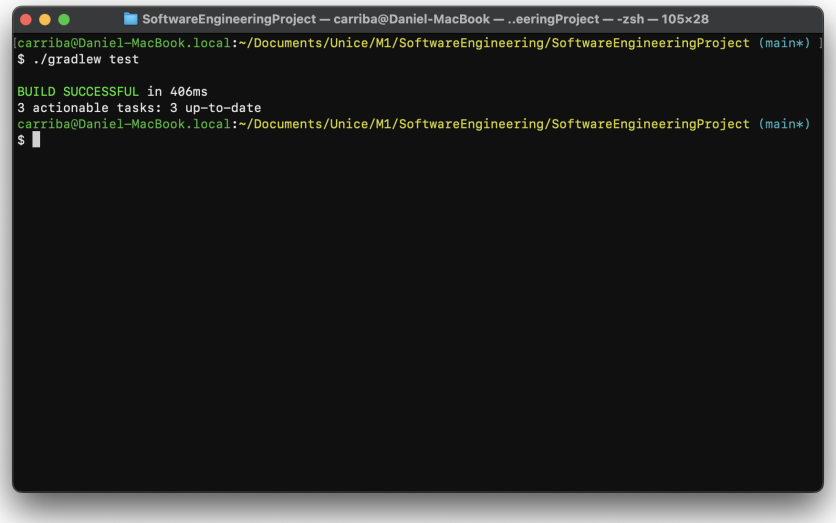

Figure 2: Tests unitaires exécutés avec succès

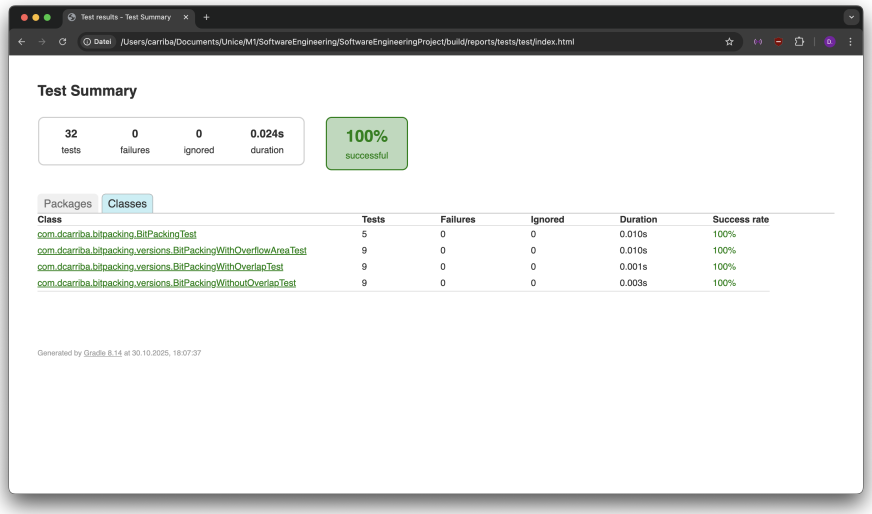

Figure 3: Visualisation des résultats des tests unitaires

4 Benchmarks

4.1 Protocole pour mesurer le temps

Pour mesurer la performance en temps des méthodes des différentes versions de "Bit Packing", on a choisit un protocole constitué des étapes suivantes.

4.1.1 Phase "warm-up"

Avant de mesurer les temps, il est important de faire une phase de "warm-up" (échauffement), afin d'assurer que les optimisations de la JVM, comme la compilation "Just-in-time", n'affecte pas les résultats finaux du benchmark.

4.1.2 Mesures des temps

La mesure de temps s'effectue grâce à la "Java Virtual Machine's high-resolution time source" en nano-secondes :

```
System.nanoTime();
```

qui est appelé avant et après l'execution de la méthode à mesurer. La différence entre ces 2 temps nous donne donc le temps écoulé pendant l'exécution de la méthode à mesurer.

La classe *TimeBenchmarks* implémente les méthodes pour mesurer ainsi le temps pour les méthodes de compression, décompression et accès au i-ème élément du tableau compressé.

4.1.3 Calcul du temps moyen sur des entrées variables

Nous choisissons de prendre la moyenne sur un nombre N (ici 10, mais modifiable grâce à une constante dédiée dans la classe *Config*) de répétitions de mesures de temps. De plus, à chaque répétition le tableau obtient de nouvelles valeurs aléatoires. Ceci nous assure donc un calcul de temps moyens plus fiable.

Ces benchmarks sont exécutés à plusieurs reprises avec plusieurs tableaux de taille différents, pour ainsi obtenir des temps moyens pour des tailles de tableaux différents, et donc pouvoir comparer les temps par rapport à la version de *BitPacking* et par rapport à la taille du tableau en entrée.

La classe *RunTimeBenchmarks* s'occupe d'exécuter ces benchmarks, et est appelée dans la fonction main de la classe *Main*.

4.2 Protocole pour mesurer le taux de compression

4.2.1 Mesures des taux de compression

La mesure du taux de compression s'effectue en compressant un tableau, et en calculant le résultat de la division suivante :

```
bitPacking.getCompressedArray().length / array.length
```

où "bitPacking" est un objet de type *BitPacking* qui contient le tableau compressé de "array"

La classe *CompressionRatioBenchmarks* implémente la méthodes qui effectue ce calcul de taux de compression

4.2.2 Calcul du taux moyen sur des entrées variables

Nous choisissons de prendre la moyenne sur un nombre N (ici 10, mais modifiable grâce à une constante dédiée dans la classe *Config*) de répétitions de mesures de taux de compression. De plus, à chaque répétition le tableau obtient de nouvelles valeurs aléatoires. Ceci nous assure donc un calcul du taux de compression plus fiable.

Ces benchmarks sont exécutés à plusieurs reprises avec plusieurs tableaux contenant chacun des valeurs maximales différentes. Ainsi on peut donc comparer le taux de compression des différentes versions en fonction des valeurs contenus à l'intérieur des tableaux en entrée.

4.3 Exemple de résultats des benchmarks

```
$ ./gradlew run

> Task :run

-----------------------------------
| *** BIT PACKING BENCHMARKS *** |
-----------------------------------

*** Compression Ratio Benchmarks ***

Results of the compression ratio benchmarks, using randomly generated arrays
with different possible maximum values:

=== Average Compression Ratio (average over 10 repetitions) ===
Max Possible Value      With Overlap      Without Overlap      With Overflow Area
1                       0,03              0,03                 0,38
10                      0,13              0,13                 0,18
100                     0,22              0,25                 0,33
256                     0,25              0,25                 0,41
1000                    0,31              0,33                 0,46
5000                    0,41              0,50                 0,46
10000                   0,44              0,50                 0,49
2097151                 0,66              1,00                 0,82
16777215                0,75              1,00                 0,91
1073741823              0,94              1,00                 1,09
2147483647              0,97              1,00                 1,13

-----------------------------------
```

```
*** Time Benchmarks ***

Warming up the JVM for better results...
Warm-up completed.

Results of the time measurements benchmarks, using randomly generated arrays
at different sizes:

=== Average Compression Time (in micro-seconds) (average over 10 repetitions) ===
Array Size    With Overlap        Without Overlap     With Overflow Area
100           0,82                0,31                7,15
500           3,59                0,86                95,45
1000          7,02                1,68                331,11
5000          33,59               7,84                7566,23

=== Average Decompression Time (in micro-seconds) (average over 10 repetitions) =
Array Size    With Overlap        Without Overlap     With Overflow Area
100           3,85                2,39                1,89
500           8,08                7,19                8,57
1000          5,96                8,40                16,58
5000          21,68               7,68                86,25

=== Average Get Time (in nano-seconds) (average over 10 repetitions) ===
Array Size    With Overlap        Without Overlap     With Overflow Area
100           5,50                2,20                1167,80
500           5,00                2,00                6083,40
1000          5,10                2,40                12364,90
5000          5,00                2,00                70411,00

BUILD SUCCESSFUL in 8s
2 actionable tasks: 1 executed, 1 up-to-date
```

4.4 Interprétation des benchmarks obtenus

4.4.1 Benchmarks de taux de compression

On peut remarquer que la version "With Overlap" a un taux de compression toujours meilleurs que les autres versions de la méthode de compression.

De plus on remarque que les taux de compression de chaque version est différent de la valeur maximale contenu dans le tableau à compressé. Ainsi plus cette valeur maximale est inférieur, mieux le taux de compression est.

Le taux de compression de la version "Without Overlap" est le second meilleur si la valeur maximale du tableau à compressé est particulièrement petite ou grande. Sinon pour une valeur maximale relativement moyenne, la version "With Overflow Area" donne de meilleurs taux que la version "Without

Overlap". Pour des petites valeurs maximales, la version "With Overflow Area" n'est pas très optimisé, et pour des valeurs maximales très élevées, la version "With Overflow Area" a même un tableau compressé de taille plus grande que le tableau original.

Ainsi on peut donc conclure que la version "With Overlap" donne toujours le meilleurs taux de compression, et que la version "With Overflow Area" n'est pas adapté si la valeur maximale du tableau à compressé est particulièrement petite ou particulièrement grande.

4.4.2 Benchmarks de temps

On peut remarquer que la version "Without Overlap" à presque toujours les meilleurs temps pour la compression, la décompression et pour l'accès au i-ème élément du tableau compressé.

La version "With Overlap" a également des temps correctes vis-à-vis de la compression, décompression et accès au i-ème élément.

Cependant, la version "With Overflow Area" a dans la majorité des cas les temps les plus élevées. Notamment pour l'accès au i-ème élément on remarque un temps particulièrement élevée. Ceci-ci est surement dû à mon implémentation de cette méthode, qui est obligée de systématiquement décompresser le tableau avant de pouvoir déterminer le i-ème élément.

Ainsi on peut donc conclure que les meilleurs versions de "Bit Packing" en fonction du temps sont les versions "Without Overlap" suivi par "With Overlap" et finalement "With Overflow Area", qui a des temps considérablement supérieur au autres versions.

4.4.3 Conclusion sur les benchmarks

Grâce au différents benchmarks obtenus, on peut conclure que la meilleur version de "Bit Packing" en considérant le temps de compression, décompression et accès au i-ème est la version "Without Overlap".

Si on considère le taux de compression, on peut conclure que la version "With Overlap" de "Bit Packing" offre les meilleurs résultats.

5 Temps de transmission pour une latence t où la compression devient utile