Software Engineering Project 2025
Data Compression for Faster Transmission

Fiona CUVILLIEZ - M1 Informatique Université Côte d'Azur

GitHub : https://github.com/fionavatar/cuvilliez-project-se.git

Automne 2025

Résumé du projet :

Ce projet étudie la transmission de tableaux d'entiers via différents modes de compression. L'objectif est de réduire le nombre d'entiers transmis tout en gardant l'accès direct aux données après compression.

Trois modes de compressions ont été codés :

- **crossing** : un entier peut être écrit sur deux blocs consécutifs
- **noCrossing** : pas de chevauchement possible
- **overflow** : basé sur la logique de noCrossing mais utilise une table d'overflow pour optimiser le stockage des entiers très grands

Principaux résultats : Benchmark sur 30 éléments

overflow | compress: 0.000588s | decompress: 0.000058s | avg get: 0.000005s | taille compressée: 12 mots
Compression ratio: 2.50x | Gain de transmission: 17.999354s

crossing | compress: 0.000438s | decompress: 0.000415s | avg get: 0.000003s | taille compressée: 9 mots
Compression ratio: 3.33x | Gain de transmission: 20.999147s

noCrossing | compress: 0.000478s | decompress: 0.000401s | avg get: 0.000053s | taille compressée: 10 mots
Compression ratio: 3.00x | Gain de transmission: 19.999121s

Introduction

Contexte :

Aujourd'hui, beaucoup de données circulent sur Internet sous forme de tableaux d'entiers, que ce soit pour des capteurs, des images ou des indices. Transmettre ces tableaux tels quels peut vite devenir coûteux en espace et en temps, surtout quand ils sont volumineux.

Problématique :

L'objectif de ce projet est de réduire la taille des données transmises tout en gardant la possibilité d'accéder directement n'importe quel élément du tableau après compression (sans décompressé entièrement le tableau).

Objectifs :

Ce projet a pour but de :

1. Implémenter différentes méthodes de compression de tableau d'entiers (crossing, noCrossing, overflow).
2. Permettre un accès direct à chaque élément via la fonction get(i).
3. Mesurer les performances avec des benchmarks : temps de compression, décompression, accès direct et gain de transmission.
4. Comparer les méthodes et déterminer dans quelles conditions chacune est la plus efficace, notamment avec des entiers très grands.

Méthodologie et conception

Présentation générale des méthodes de compression :

• Bit Packing (logique générale) :
Le principe du bit packing repose sur le fait que les entiers sont normalement codés sur 32 bits, même lorsque ce n'est pas nécessaire. L'idée est donc de réduire cette taille en fonction du plus grand entier du tableau.
On calcule d'abord un entier k via la fonction calculerK(array), qui correspond au nombre de bits nécessaires pour représenter le plus grand entier du tableau. Chaque entier est ensuite converti sur k bits (avec un padding si besoin pour compléter les bits manquants).

Exemple : data = [1, 2, 3, 4, 5] calculerK(data) -> k = 3

1 = 001 -> 1 bit
2 = 010 -> 2 bits
3 = 011 -> 2 bits
4 = 100 -> 3 bits
5 = 101 -> 3 bits

• Méthode Crossing :
La méthode crossing cherche à maximiser l'utilisation des 32 bits disponibles. Les entiers compressés sont simplement ajoutés les uns à la suite des autres dans des grandes chaînes binaires de longueur 32. L'intérêt de cette approche est une excellente compacité, avec un tableau final d'une taille de len(data) * k / 32 au lieu d'avoir un tableau de taille 32 * len(data).
La principale difficulté réside dans l'implémentation de get(i). Il faut déterminer si l'entier à l'indice i chevauche deux blocs et, le cas échéant, reconstituer correctement l'entier à partir des deux.
Cette approche limite le padding mais rend la logique d'accès plus complexe.

• Méthode No Crossing :
Dans cette version, aucun entier ne peut chevaucher deux blocs.
Cela réduit légèrement le taux de compression mais simplifie énormément la lecture : un get(i) ne nécessite de lire qu'un seul bloc. On peut placer 32 // k entiers par bloc. Si ajouter un nouvel entier dépasse la taille de 32 bits, on commence simplement un nouveau bloc.
Cette approche est plus simple et plus rapide à l'accès, au prix d'un léger gaspillage de bits.

• Méthode Overflow
La méthode overflow part du même principe que No Crossing, mais ajoute une zone d'overflow pour gérer les grands entiers.
L'idée est de ne pas fixer k selon le plus grand entier global (souvent un cas isolé), mais selon une majorité d'éléments.
On définit :
- k1 : nombre de bits nécessaires pour le plus grand entier,
- k2 : nombre de bits nécessaires pour environ 80 % des entiers du tableau.

Les entiers sont alors encodés de la manière suivante :
- si len(bin(entier)) <= k2, on code l'entier sur k2 bits avec un flag 0 en tête (k2 + 1 bits);
- sinon, on encode la position dans la zone d'overflow avec un flag 1, et l'entier complet est stocké à la fin du tableau dans cette zone d'overflow (codé sur k1 bits).

Cette approche réduit le gaspillage lorsque quelques valeurs extrêmes déséquilibrent la taille de k.
Lors de la lecture, la fonction get(i) vérifie le flag :
- 0 → lecture directe dans la zone principale ;
- 1 → lecture dans la zone d'overflow, on décompresse la zone overflow.

• Difficulté du padding et gestion des zéros significatifs
L'une des principales difficultés lors de la compression et de la décompression est la perte ou l'ajout d'informations, notamment liée à la disparition des zéros initiaux dans les représentations

binaires.
En effet, lorsqu'on convertit une chaîne binaire en entier, puis qu'on la reconvertit en binaire, les zéros de tête disparaissent : 000101 -> 5 -> 101

Mais si cette valeur correspondait en réalité à deux entiers [0, 5], on perd une information essentielle : le premier zéro.
Ce problème rend la reconstruction du tableau original incorrecte sans précaution particulière.
La solution adoptée est donc un padding systématique à chaque étape.
Chaque bloc binaire est complété à 32 bits avant d'être converti en entier.

- Lors de la compression, le padding est effectué à droite (c'est-à-dire en ajoutant des zéros à la fin) pour s'assurer que chaque bloc occupe exactement 32 bits.
  Dans le cas du mode crossing, ce remplissage n'est nécessaire que pour le dernier bloc, puisque les précédents sont déjà pleins.
- Lors de la décompression, on effectue un padding à gauche, afin que tous les blocs aient à nouveau une taille homogène de 32 bits.
  Cela permet de parcourir les données de gauche à droite sans perte d'alignement.

Mais un cas limite subsiste : comment distinguer les vrais zéros d'un remplissage artificiel ?
Pour résoudre cela, la fonction de compression renvoie également la taille réelle de la chaîne du dernier bloc avant padding, ce qui permet de savoir précisément où s'arrêter lors de la décompression.
On aurait pu transmettre le nombre d'éléments initiaux, mais cela pose problème si ce nombre ne peut pas être représenté sur 32 bits (il pourrait alors être réparti sur deux blocs).
La taille réelle avant padding est donc un indicateur plus fiable.
Concernant la zone d'overflow, le même principe est appliqué :
la zone principale est paddée, tandis que la zone d'overflow contient le nombre d'éléments et leur taille, ce qui permet une reconstruction exacte sans ambiguïté (même si on padde pour une raison homogénéité).

• Gestion des différents modes : la Factory
Pour rendre le code plus modulaire et éviter les duplications, j'ai implémenté une factory chargée de créer automatiquement l'objet de compression correspondant au mode choisi par l'utilisateur (crossing, noCrossing ou overflow).
L'idée est simple : plutôt que d'importer et d'instancier manuellement une classe différente dans chaque fichier, la factory agit comme un point d'entrée unique qui retourne la bonne instance selon le paramètre passé.
Concrètement, la fonction compressor_factory(mode, data) prend en argument le nom du mode et les données à compresser.

Cela permet de centraliser la création des objets et de simplifier le code de test ou d'exécution.
Par exemple, le fichier src/main.py ou les scripts de benchmark peuvent appeler la même fonction sans se soucier des détails d'implémentation internes.
L'utilisation d'une factory présente plusieurs avantages :

- elle rend le code plus extensible (on peut ajouter facilement un nouveau mode de compression sans modifier le reste du programme)
- elle clarifie la structure du projet en séparant la logique de sélection du mode de la logique de compression elle-même
- elle facilite les tests : les mêmes fonctions de test et de benchmark peuvent être utilisées pour tous les modes, simplement en changeant le paramètre mode

Implémentation

• Structure du projet
Le projet est organisé de manière modulaire pour séparer la logique de compression, les tests, les exemples et les mesures de performance.
Cette organisation permet de garder un code clair :

- `src/` contient le cœur du projet (logique et exécution)
- `benchmark/` regroupe les scripts de mesure du temps et du ratio de compression
- `examples/` contient des démonstrations de tests
- `tests/` regroupe des données ou cas de vérification.

Méthodes principales

- `compress(data)` : compresse le tableau d'entiers selon le mode sélectionné.
- `decompress(data)` : reconstitue le tableau original à partir des blocs compressés.
- `get(i, compressed)` : renvoie directement le i-ème entier sans décompresser entièrement le tableau.

Ces trois fonctions sont présentes dans toutes les classes, ce qui garantit une interface commune et permet de les tester ou de les comparer facilement.

Le fichier factory.py joue le rôle de point central pour créer les bons objets. Grâce à cette structure, les autres fichiers comme main.py ou benchmark/measures.py n'ont besoin que d'un import unique : from src.factory import compressor_factory.
L'exécution se fait ensuite simplement via : python3 -m src.main noCrossing ./tests/data7.txt

Benchmark et protocole de mesure

L'objectif des benchmarks est de quantifier les performances réelles des différentes méthodes de compression.
Les mesures portent sur 4 aspects principaux :

1. le temps de compression (compress()),
2. le temps de décompression (decompress()),
3. le temps moyen d'accès direct (get()),
4. et enfin le gain global de transmission, en comparant la durée totale d'envoi avec et sans compression.

Ces données permettent de déterminer à partir de quelle latence de transmission la compression devient réellement avantageuse.

• Protocole de mesure précis :
Pour obtenir des mesures fiables et précises, le protocole suivant a été mis en place :

- Utilisation de la fonction time.perf_counter() afin de mesurer les temps d'exécution avec une résolution en nanosecondes.
- Chaque opération (compress, decompress, get) est chronométrée séparément.
- Le temps d'accès direct est évalué à partir de la moyenne sur 10 accès aléatoires

Le **ratio de compression** est calculé pour estimer la réduction de taille :

$$\text{ratio} = \text{taille originale (en bits)} / \text{taille compressée (en bits)}$$

Un ratio > 1 indique une compression efficace.

Enfin, pour évaluer l'intérêt de la compression selon une latence donnée t, le **temps total de transmission** est estimé à l'aide des formules suivantes :

$$T_{\text{no comp}} = t + \text{taille originale}$$
$$T_{\text{comp}} = \text{compression\_time} + t + \text{taille compressée} + \text{decompression\_time}$$

La compression devient rentable lorsque : $T_{comp} < T_{no\ comp}$.

Présentation des résultats sous forme de tableau

• Benchmarks sur 30 éléments
Data = [8, 55, 19, 63, 71, 13, 27, 91, 44, 68, 92, 45, 79, 8, 65, 71, 2, 50, 14, 100, 33, 77, 41, 9, 56, 63, 59, 40, 2, 80]

\	compress	decompress	Moyenne get	taille comp	Comp ratio	Gain de transmissio
Overflow	0.000856s	0.000074s	0.000006s	12	2.50x	17.999071s
No crossing	0.000697s	0.000528s	0.000069s	10	3.00x	19.998775s
Crossing	0.000552s	0.000534s	0.000004s	9	3.33x	20.998914s

• Analyse des performances
Les premiers résultats montrent que dans ce cas particulier:

- **crossing** offre la meilleure densité de compression et de gain de transmission mais une décompression plus longue
- **noCrossing** est plus rapide en décompression moins efficace en taille ;
- **overflow** devient intéressant sur des données hétérogènes, où quelques valeurs très grandes faussent le calcul de k ce qui n'est pas vraiment le cas dans cet exemple. Si on passe une des valeurs à -> 10 000 overflow devient la méthode la plus intéressante

• Benchmarks sur 30 éléments
Data = [8, 55, 19, 63, 71, 13, 27, 91, 44, 68, 92, 45, 79, 8, 65, 71, 2, 50, 14, **10000**, 33, 77, 41, 9, 56, 63, 59, 40, 2, 80]

\	compress	decompress	Moyenne get	taille comp	Comp ratio	Gain de transmissio
Overflow	0.000406s	0.000042s	0.000003s	13	2.31x	16.999552s
No crossing	0.000444s	0.000408s	0.000030s	17	1.76x	12.999148s
Crossing	0.000429s	0.000457s	0.000002s	16	1.88x	13.999114s

En pratique, le choix du mode dépend du contexte :

- **crossing** pour maximiser le taux de compression,
- **noCrossing** pour les scénarios nécessitant des accès rapides,
- **overflow** pour des données très variables en amplitude.

Remarques

• Choix techniques

Le mode Overflow repose sur la logique du NoCrossing. Ce choix s'explique principalement par la simplicité à implémenter cette méthode : le fait de ne pas avoir de chevauchement entre blocs rend l'accès direct (get(i)) beaucoup plus rapide et plus prévisible.
Cela facilite également la gestion de la zone d'overflow, qui doit rester bien délimitée dans la mémoire. Si le mode Crossing avait été utilisé comme base, la détection des frontières entre la zone principale et la zone d'overflow aurait été plus complexe et risquée en termes de cohérence des bits.

• Limites de la méthode actuelle
Même si les trois modes fonctionnent et remplissent les objectifs du projet, certaines limites persistent :

- Le padding (ajout de zéros) reste coûteux, notamment pour les petits tableaux où le gain de compression peut être annulé mais semble pourtant nécessaire c'est pourquoi la méthode crossing est la plus efficace en gain d'espace.
- La gestion de la latence fixe ne reflète pas toujours les conditions réelles de transmission (réseaux instables, pertes, etc.).
- Enfin, le mode Overflow suppose un seuil (ici 80%) pour déterminer les entiers à mettre dans la zone d'overflow. Ce seuil est arbitraire et pourrait être affiné par une analyse statistique plus fine du jeu de données mais rajoute du temps surtout si le tableau est grand.
-

Cas particuliers :

- Overflow et grands nombres : la méthode est efficace pour les tableaux où quelques valeurs isolées sont très grandes, mais devient inutile si la majorité des entiers nécessitent déjà un grand nombre de bits.
- Nombres négatifs : dans cette version, les entiers négatifs ne sont pas directement pris en charge. Leur gestion nécessiterait une adaptation du codage binaire
  par exemple en utilisant un bit de signe ce qui pourrait partir du même principe que le flag '0' si positif et '1' si négatif ou une représentation complément à deux.
- Tableaux très petits : sur des jeux de données de taille réduite, le coût de la compression (en temps) est souvent supérieur au gain obtenu.

Pistes d'amélioration
Plusieurs pistes peuvent être envisagées pour améliorer la qualité et la flexibilité du système :

- Gestion des entiers signés : implémenter une version compatible avec les nombres négatifs.
- Paramétrage du seuil Overflow : permettre à l'utilisateur de définir le pourcentage utilisé pour la détection des grands nombres.
- Visualisation des performances : ajouter un module graphique pour afficher les courbes de temps de compression et de ratio selon la taille des données.

Conclusion

Ce projet a permis d'explorer différentes approches de compression d'entiers dans le but d'accélérer leur transmission tout en maintenant un accès direct aux données.
Les trois méthodes développées — **Crossing**, **NoCrossing** et **Overflow** — illustrent bien les compromis possibles entre efficacité de compression, simplicité d'accès et coût en calcul.

Le mode Crossing optimise pleinement l'utilisation des 32 bits disponibles, mais complexifie la gestion des accès directs.
Le NoCrossing, plus simple et robuste, offre un meilleur compromis entre performance et lisibilité du code, au prix d'une compression légèrement moins efficace.
Enfin, le mode Overflow s'inspire de NoCrossing tout en introduisant une table secondaire pour les grands entiers, ce qui permet d'améliorer la flexibilité du schéma sans alourdir les opérations courantes.

Les mesures effectuées ont permis de quantifier les temps de compression, de décompression et d'accès direct, ainsi que le gain de transmission obtenu.
Ces résultats montrent que, selon la latence du réseau, il existe un seuil à partir duquel la compression devient réellement bénéfique.
Cette démarche expérimentale met en évidence la nécessité d'adapter la stratégie de compression au contexte d'utilisation réel.

Sur le plan personnel, ce projet m'a permis d'approfondir la compréhension du bit packing, des compromis entre mémoire et temps, et de l'importance d'une architecture logicielle claire (notamment avec l'usage de la factory pour gérer plusieurs modes de compression).
Il m'a également confrontée à des problématiques concrètes comme la perte d'informations liée au padding ou la gestion des cas limites lors de la décompression.

En conclusion, ce travail constitue une base solide pour des développements futurs : gestion d'entiers négatifs, adaptation dynamique de la taille de codage pour l'overflow, ou encore intégration de nouveaux schémas de compression plus adaptatifs.