UNIVERSITÉ CÔTE D'AZUR

MASTER INFORMATIQUE

Rapport de TER

Compression d'ensembles ordonnés

Réalisé par
Margaux Schmied

Encadré par
Jean-Charles Régin
Marie Pelleau

Master Informatique
à Université Côte d'Azur
Second semestre de l'année 2021-2022

Table des matières

Première partie
Introduction

Le but de ce TER est d'étudier et d'implémenter des algorithmes de compression d'ensembles ordonnés afin de permettre une représentation plus compacte d'un modèle structuré.
L'idée générale est de remplacer le processus classique de transfert d'un modèle entre 2 ordinateurs par le processus suivant : compression - transfert - décompression. La difficulté est que ce second processus soit plus rapide. Cette étude a pour but de déterminer quand il est plus intéressant (en terme de temps de calcul) d'utiliser ce second processus.

Dans ce TER j'ai utilisé et défini des algorithmes de compression et décompression basés sur les spécificités de la programmation par contraintes.
En programmation par contraintes, les problèmes sont modélisés par un ensemble de variables, un ensemble de domaines, et un ensemble de contraintes. Les contraintes sont les relations qui existent entre les variables d'un problème. Les domaines sont les valeurs qui peuvent être prises par les variables. Ils peuvent être représentés par un intervalle de valeurs ($[1, 10]$, toutes les valeurs entre 1 et 10) ou par un ensemble de valeurs ($\{1, 2, 3, 4, 5, 6, 7, 8, 10\}$, toutes les valeurs de 1 à 8 et 10). La première représentation est plus compacte mais beaucoup moins précise. En effet, on ne peut pas préciser qu'une valeur à l'intérieur de l'intervalle ne peut pas être prise par les variables. La seconde représentation est beaucoup moins compacte cependant elle est beaucoup plus précise.

Afin de transférer ou de stocker nos données, celles-ci doivent être encodées sur le même nombre d'octets que le plus grand élément. Par exemple, si nous avons $\{1, 2, 3, 256\}$ le plus grand élément étant 256 alors nous devons utiliser 4 blocs de 2 octets et non 3 blocs d'1 octet et 1 de 2 octets.

Ce rapport présente dans un premier temps les données réelles et artificielles qui m'ont servies à tester les performances de chaque algorithme. Puis les algorithmes étudiés sont présentés. Certains sont très connus mais ne sont pas pensés pour les particularités des données issues de la programmation par contraintes. D'autres sont plus récents et plus adaptés aux ensembles ordonnés.

Et enfin nous verrons le travail effectué ainsi que le déroulement et les résultats des expériences.

Deuxième partie
Ensemble de données

L'idée initiale est d'utiliser la compression au service de la programmation par contraintes. C'est pour cette raison que mes encadrants m'ont fourni des données issues de plusieurs modélisations du problème du voyageur de commerce. Il s'agit des valeurs pour des variables utilisées dans des modèles de programmation par contrainte.

Ces données sont des ensembles de nombres entiers pseudo-ordonnés, autrement dit ils peuvent être divisés en sous-ensemble ordonnés.

J'ai donc pu travailler avec 197 sets de données qui en plus d'être pseudo croissants ont comme particularité d'avoir des nombres négatifs. En plus de cela pour chaque sous-ensemble il n'y a aucun doublon. Ce qui sera appelé doublon par la suite est une suite d'éléments identiques.
Ces données étant réelles on peut noter qu'il y a de grands écarts de valeur dans chaque sous-ensemble, des suites successives ainsi que de grands écarts d'une valeur à l'autre. Ces données m'ont permis de tester les algorithmes que j'ai implémentés sur des valeurs réelles.

Tous ces sets de données sont composés de deux entiers excepté les 5 derniers qui sont de taille conséquente, ils sont composés respectivement de 16861, 5532, 10753, 4286 et 2749 éléments. Pour compenser le manque de grand set, j'ai créé en parallèle des sets de données artificielles dans le but de mettre à l'épreuve les algorithmes.

J'ai donc créé des ensembles de nombres entiers strictement positifs, sans doublon et croissant. Ils sont compris dans des intervalles de valeur allant de 100 à 10 000 000 et sont composés de 10 à 1 000 000 nombres. Certains ont donc des intervalles larges entre chaque élément tandis que d'autres sont plus denses.

En plus de cela j'ai également généré des ensembles positifs, pseudo croissant et sans doublon dans chaque sous-ensemble croissant avec les mêmes intervalles que pour les données artificielles strictement croissantes.

Troisième partie
Préparation des données

Nous présentons dans un premier temps différents algorithmes de préparation des données puis différents algorithmes de compression.

1 Différence delta

La première méthode abordée sera de faire une différence successive sur tous les éléments. Cette méthode pourra être composée avec tous les algorithmes présentés plus tard.

L'approche standard est d'utiliser un décalage de 1 :

$$\forall i \in [1, n] \quad \delta_i = x_i - x_{i-1}$$

avec $\delta_0 = x_0$.

Autrement dit la suite $x_0, x_1, \ldots, x_n$ devient $x_0, x_1 - x_0, \ldots, x_n - x_{n-1}$.

Par exemple, $1, 2, 3, 4, 5, 6, 7, 8, 10$ devient $1, 1, 1, 1, 1, 1, 1, 1, 2$.

Dans le cas standard on peut retrouver la forme initiale en faisant la somme des deltas :

$$\forall j \in [1, n] \quad x_j = x_0 + \sum_{i=1}^{j} \delta_i$$

Comme pour tout algorithme de compression l'objectif est de réduire la taille finale des données. Les nombres négatifs prenant beaucoup de place cette méthode ne s'applique que sur des données croissantes pour empêcher cela. Elle est donc complètement adaptée à nos données en effet il m'a suffi de traiter séparément chaque sous-ensemble croissant.

Il existe des méthodes utilisant un décalage différent.

2 Single-instruction-multiple-data

Les SIMD sont des types d'instructions bas niveau qui permettent de faire des actions bien plus rapidement qu'en utilisant des commandes classiques.

Dans l'article Lemire et Boytsov (2021) les auteurs recommandent de les utiliser pour faire une différence delta avec un décalage de 4. L'avantage de choisir un décalage de 4 est que l'on peut compresser 4 éléments à la fois en utilisant les instructions SIMD. Les instructions SIMD permettent de faire 4 additions ou soustractions simultanément.

Si on applique ces instructions à partir de la liste $(3, 4, 12, 1)$ la taille de la liste peut être rapportée à 2^L où L sera le nombre de déplacements et d'additions, ici $4 = 2^2$ donc $L = 2$, il y aura 2 déplacements et 2 additions.

Un déroulement de la compression pas à pas donnerait :

$$\begin{aligned} (3, 4, 12, 1) + (0, 3, 4, 12) &= (3, 7, 16, 13) \\ (3, 7, 16, 13) + (0, 0, 3, 7) &= (3, 7, 19, 20) \end{aligned}$$

3 Complémentaire de l'ensemble

Une propriété intéressante des ensembles croissants est d'utiliser le complémentaire :

$$\forall x_i \in X \quad \forall j \in]x_0, x_n[\quad x_i \neq j$$

avec comme premier et dernier élément de la liste x_0 et x_n.

Où x_i est le i^e élément de la liste à compresser.

En appliquant cette méthode à $(1, 3, 5, 6, 8)$ cela donnera $(1, 2, 4, 7, 8)$.

On notera que c'est le seul algorithme, avec le Codage par plages, présenté qui une fois terminé peut renvoyer une quantité d'entiers plus petite qu'à l'entrée. Les autres algorithmes vont renvoyer une quantité égale ou supérieure d'entiers, mais de taille inférieure.

4 Compression successive

La compression successive, va compresser uniquement les suites croissantes successives. Si on garde le même exemple $(1, 3, 5, 6, 7, 8, 15, 16, 17)$ devient $(1, 3, 5, 8, 15, 17)$. Afin de pouvoir

décompresser un marqueur 0 est ajouté devant les éléments à décompresser. On obtient donc $(1, 3, 0, 5, 8, 0, 15, 17)$.

Quatrième partie
Algorithmes de compression

5 Codage par plages

Le codage par plage est basé sur le nombre d'éléments côte à côte identique. Chaque sous-ensemble d'élément identique est remplacé par la quantité suivie de l'élément, c'est ainsi que $(1, 1, 1, 1, 2, 3, 3, 3, 3, 3, 4)$ devient $(4, 1, 1, 2, 5, 3, 1, 4)$.

6 Codage de Huffman

Le principe du codage d'Huffman est de créer un nombre binaire unique associé à chaque élément des données à compresser. Pour cela en fonction de la fréquence de chaque élément nous créons un arbre qui nous donnera leurs codes associés.
Pour créer l'arbre, on réunit les deux nœuds de plus faibles fréquences pour donner un nouveau nœud dont le poids équivaut à la somme de ses fils puis on continue jusqu'à ce qu'il ne reste qu'un nœud, la racine. Pour trouver les codes, il suffit de parcourir l'arbre, les branches de gauche sont des 0 et celle de droite des 1.

Un Exemple d'arbre d'Huffman sera bien plus parlant. Si on prend la suite de chiffres $(1, 3, 4, 1, 5, 6, 4)$ ici 1 et 4 ont une fréquence de 2 tandis que les autres ont une fréquence de 1.
Pour la création de l'arbre, nous pouvons réunir 3 et 6 qui font partie des fréquences les plus basses et qui forment désormais un nœud de fréquence 2. Puis 5 est le dernier avec une fréquence de 1 je choisis arbitrairement de l'associé au dernier nœud créé mais il aurait pu également être associé à 4 ou 1 qui ont eux aussi une fréquence de 2. 4 et 1 sont désormais les plus petits si on les associe ensemble cela forme une fréquence de 4 alors que si on en associe un au dernier nœud créé on aura 5 donc on choisit de les réunir. Il ne nous reste plus qu'à réunir les deux derniers nœuds.

Si on parcourt l'arbre en mettant comme dit précédemment 0 à gauche et 1 à droite on obtient donc le code visible dans la figure 2.

Le codage d'Huffman crée un code qui est non ambigu ce qui permet de ne pas tout formater sur le même nombre de bits contrairement à la forme non compressée. À la fin de la compression, il reste donc les éléments remplacés par leur code ainsi que le code du tableau qui servira à la décompression.

7 VByte

L'algorithme VByte est présenté dans l'article Lemire *et al.* (2017), la première étape est d'appliquer l'algorithme des Différence delta et on peut y associer les Single-instruction-multiple-data.

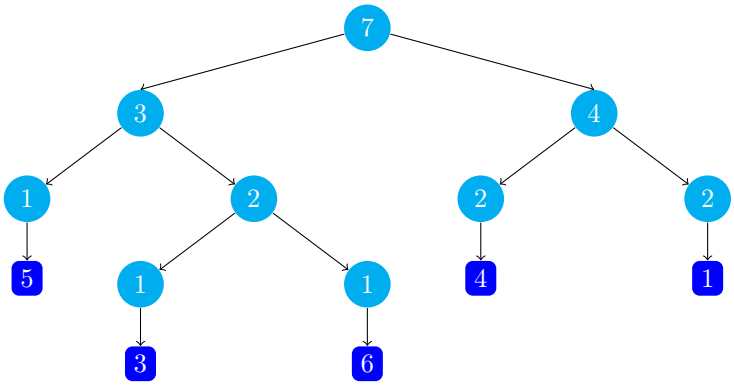


FIGURE 1 – Exemple d'arbre d'Huffman

Nombre	Binaire (1 octet)	Code
1	0000 0001	11
3	0000 0011	010
4	0000 0100	10
5	0000 0101	00
6	0000 0110	011

FIGURE 2 – Code d'Huffman de l'exemple

Ensuite chaque entier après cette première étape est traduit en binaire. Si l'entier tient sur 7 bits alors le 8^e bit est marqué à 0 tandis que si l'entier est plus grand le 8^e bit est marqué à 1 et continue sur le bloc suivant.

32 ⇒ 00100000
128 ⇒ 00000001 10000000
32, 128 ⇒ 00100000 00000001 10000000

FIGURE 3 – Exemple de l'algorithme VByte

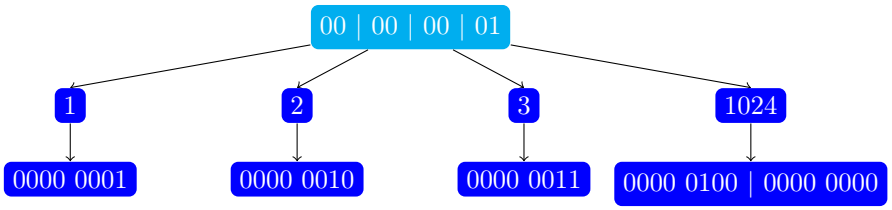
Dans la figure montrant un Exemple de l'algorithme VByte en rouge on peut voir les bits de marquage. 128 s'écrit normalement 10000000 mais comme VByte nous impose de l'écrire sur uniquement 7 bits par bloc le 8^e bit de 128 est décalé sur un second bloc. nous marquons le premier bloc avec un 1 pour indiquer que la lecture se continue sur le prochain bloc et donc 128 devient 00000001 10000000.

8 Stream VByte

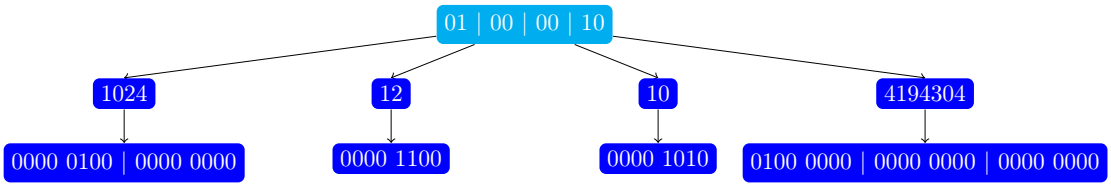
L'algorithme stream VByte également de l'article Lemire *et al.* (2017) commence en appliquant la Différence delta. Puis les entiers sont convertis en binaire. Cet algorithme limite la taille des entiers qui doivent absolument tenir sur 4 octets maximum. Pour chaque entier converti on stocke également le nombre d'octets utilisés auquel on soustrait 1 dans 2 bits de contrôle donc pour un élément qui tient sur 1 octet on va garder 00, sur 2, 3 et 4 octets vont garder respectivement 01, 10 et 11.

Si donc on essaie de compresser avec un Exemple de l'algorithme stream VByte (1024, 12, 10, 4194304, 1, 2, 3, 1024) qui tiennent respectivement sur 2, 1, 1, 3, 1, 1, 1 et 2 octets cela nous donne 01000010 00000001 pour le stockage et 00000100 00000000 00001100 00001010 01000000 00000000 00000000 00000001 00000010 00000011 0000100 00000000 pour la compression des chiffres.

Cette méthode nous permet donc de stocker chaque entier sur un nombre d'octets différents.



(a) Nombres correspondant au premier octet de contrôle



(b) Nombres correspondant au second octet de contrôle

FIGURE 4 – Exemple de l'algorithme stream VByte

9 SIMD-BP128*

L'algorithme SIMD-BP128* de l'article Lemire et Boytsov (2021) commence avec une Différence delta faite avec des SIMD. Ensuite on prépare un stockage divisé en 16 blocs de 128 bits. On convertit chaque entier en binaire et on les formate tous sur le même nombre de bits que le plus grand des nombres et on place le 1^{er} dans le 1^{er} bloc, le 2^{e} dans le 2^{e} bloc, ainsi de suite jusqu'au 16^{e} nombre. Le 17^{e} nombre sera rajouté au 1^{er} bloc et ainsi de suite jusqu'à ce qu'un bloc ne puisse plus accueillir de nombres. À ce moment-là on finit d'écrire les nombres sur de nouveaux blocs verticalement, c'est-à-dire en multiple du numéro du bloc. Puis on continue jusqu'à ce qu'on ait stocké toutes les données.

Pour illustrer cet algorithme vous pouvez voir l'Exemple de l'algorithme BP32 qui a un déroulement très similaire à l'exception qu'il ce fait sur 4 blocs de 32 bits. On remarquera qu'ici les nombres sont formatés sur 5 bits.

Ainsi le 1^{er}, 2^{e}, 3^{e} et 4^{e} nombre sont sur les blocs 1, 2, 3 et 4 etc. Comme 32 n'est pas un multiple de 5 les 25^{e}, 26^{e}, 27^{e} et 28^{e} nombres doivent être coupés en deux entre les blocs 1, 2, 3

et 4 et les blocs 5, 6, 7 et 8.
D'après les travaux de Lemire la version BP128 est meilleure que la version BP32.

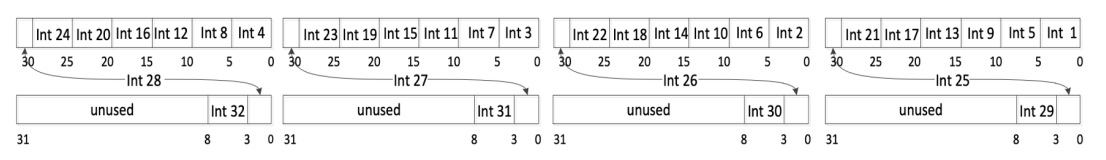


FIGURE 5 – Exemple de l'algorithme BP32

10 PFOR

L'algorithme de PFOR ainsi que ces variants, dans l'article Lemire et Boytsov (2021) sont les plus compliqués que j'ai étudié durant ce TER. L'idée générale est de formater tous les éléments sur un certain nombre de bits tout en stockant les exceptions à part. les exceptions sont les éléments qui naturellement ne tiennent pas sur le nombre de bits prédéfinis et qui ont donc dû être tronqués.

Afin de faciliter la compréhension je vais l'associer à un Exemple de l'algorithme PFOR.

On commence par effectuer la Différence delta. Ensuite nous pouvons remarquer que la plus part de nos éléments sont sur 1 ou 2 bits à l'exception des 3 plus grands qui tiennent sur 6 bits. Nous allons chercher à minimiser le coût de stockage total et donc également celui des exceptions. Pour calculer le coût de stockage nous allons utiliser une variable b qui sera le format en nombre de bit par éléments parmi ceux qui ne sont pas des exceptions. Nous utiliseront également une variable c qui représentera le nombre d'exceptions donc le nombre d'éléments qui dépassent b bit. Le coût de stockage de chaque exception est de $8 + (6 - b) = 14 - b$, les plus grandes exception sont sur 6 bits et une fois mises a l'écart il faudra les stocker sur un octet. Si on le ramène au coût global cela donne $\min(b \times 16 + (14 - b) \times c)$. En effet il faut additionner le coût total des exceptions, leur nombre (c) multiplié par leur taille ($14 - b$), et le coût de stockage des éléments tronqués, ici nous avons 16 éléments de taille b. Comme nous voulons minimiser les coûts totaux et que c dépend de b il va falloir faire varier b. Les plus grands éléments étant de taille 6, considérer les éléments de taille supérieure n'aurait pas de sens. Il va donc falloir faire varier b de 1 à 6 dans le but de minimiser le stockage total.

En testant les 6 possibilités d'affectation de b on voit que $b = 2$ donne le plus petit résultat, vous pouvez voir les différences dans le tableau 6.

Une fois nos 3 exceptions formatées il faut stocker les informations qui nous permettront de décompresser la suite de nombres. Ainsi, 2, 6, 3, 4, 9, 11 représentent respectivement notre variable b, la taille maximale de bits utilisés initialement, le nombre d'exceptions et d'adresses de chaque exception.

Pour finir il ne nous restera que nos expressions à compresser pour ça j'ai étudié deux extensions de PFOR : FastPFOR et SimplePFOR. Ces deux extensions diffèrent dans la compression des exceptions mais sont pas détaillées dans ce rapport.

b	Stockage total
1	185
2	68
3	81
4	94
5	107
6	224

FIGURE 6 – Tableau représentant le stockage total en fonction de la variation de b.

Données à compresser :	10, 10, 1, 10, 100110, 10, 1, 11, 10, 100000, 10, 110100, 10, 11, 11, 1
Données tronquées : $(16 \times 2 = 32$ bits)	10, 10, 01, 10, 10, 10, 01, 11, 10, 00, 10, 00, 10, 11, 11, 01
Adresses des exceptions : $(6 \times 8 = 48$ bits)	2, 6, 3, 4, 9, 11
Données des exceptions : (À compresser)	1001, 1000, 1101

FIGURE 7 – Exemple de l'algorithme PFOR

11 Stream VByte*$^{-1}$

Cet algorithme est une composition de plusieurs algorithmes précédents et va s'appliquer sur des Ensemble de données ordonnés ou pseudo ordonnés sans doublon. Considérant la nature des données dans les modèles de programmation par contraintes, j'ai proposé cet algorithme espérant conserver les avantages des autres algorithmes et d'être mieux adapté à ce type de données.

On commence par décaler la valeur de chaque élément de façon à ce que le plus petit élément soit égal à 1. Si le minimum m est positif on retranche $m - 1$ à tous les éléments, s'il est négatif on ajoute $1 - m$ à chaque élément. Cela permet d'une part de commencer à diminuer la taille de chaque élément, en effet si m est positif on diminue la valeur de tous les éléments, s'il est négatif nous n'avons pas besoin de "perdre" un bit de signe. De plus, cela permet de préserver la valeur 0 qui nous servira de marqueur pour l'étape suivante.

À cette étape nous allons chercher des suites successives de taille supérieure ou égale à deux auxquelles nous allons appliquer la compression successive. Pour garantir la décompression, nous rajoutons le marqueur 0 devant les éléments maintenant compresser. Une fois la compression successive appliquée sur les suites successives la suite contient uniquement des singletons ou des paires précédées par un 0 nous allons donc appliquer la différence delta sur ces paires afin de diminuer encore leurs valeurs.

Pour finir, nous appliquons l'algorithme Stream VByte. La superposition de tous ces algorithmes nous permet de cumuler deux propriétés très intéressantes en plus de la réduction de la taille de chaque élément. La première nous est donnée par la compression successive, nous pouvons transmettre une plus petite quantité d'éléments après compression qu'avant. La seconde nous est donnée par Stream VByte qui nous permet de transférer des éléments sur un nombre différent d'octets. Le fait que Stream VByte rajoute des éléments à transférer devraient être amorti par la compression successive.

Pour la décompression il suffira d'appliquer la décompression des mêmes algorithmes en faisant bien attention au marqueur 0.

Cet algorithme comportant beaucoup d'étapes il me semble avisé d'y ajouter un Exemple de l'algorithme Stream VByte^{*-1} du déroulement de chaque étape.

Nous partons de ces données (1, 2, 3, 4, 5, 7, -1, 0, 254), on remarque que le minimum est -1 nous rajoutons donc 2 à tous les éléments afin que le minimum soit à 1 puis nous rajoutons le décalage en tête de liste. Nous divisons notre liste en sous-listes croissantes, à l'intérieur desquelles nous cherchons des listes successives. Ici il n'y en a qu'une (3, 4, 5, 6, 7), on lui applique le compression successive marquée par 0 ce qui nous donne (0, 3, 7). Sur les entiers restants de la dernière compression, on applique la différence delta, il reste (0, 3, 4). Au total il nous reste désormais (2, 0, 3, 4, 9, 1, 2, 256), comme normalement chaque entier tient sur le même nombre d'octets tout est formaté sur le plus grand donc ici 2 octets. Cela nous fera un total de 16 octets or en appliquant stream VByte nous passons à 9 octets pour les données auquel on ajoute 2 octets pour mémoriser la taille de chaque élément.

12 Suite successive et différence

Cette méthode s'applique sur des listes croissantes et marche sur un principe d'emplacement pair et impair.
Le premier élément est conservé sans modification ensuite chaque indice impair est le nombre d'éléments successifs directement après la valeur courante et chaque indice pair est l'écart entre la valeur courante et la suivante.
Si on prend l'exemple (1, 3, 5, 6, 7, 8) donne (1, 0, 2, 0, 2, 3) car on conserve le 1 en premier, il n'appartient pas à une suite successive donc on met un 0, la prochaine valeur est 3 or il y a un écart de 2 entre 1 et 3, pas de suite donc un 0, un écart de 2 entre 3 et 5 et 5 appartient à une suite qui continue sur 3 éléments.
On peut aussi écrire (1, 0, 3 - 1, 0, 5 - 3, 8 - 5).

Avec un autre exemple (1, 2, 3, 5) devient (1, 2, 2, 0) en effet le premier élément est 1 et il est dans une suite successive avec 2 éléments de plus, il y a un écart de 2 entre 3 et 5 et 5 n'appartient pas à une suite.

Données à compresser :	1, 2, 3, 4, 5, 7, -1, 0, 254
Décalage du min à 1 :	2 | 3, 4, 5, 6, 7, 9 | 1, 2, 256
Compression successive :	2 | 0, 3, 7, 9 | 1, 2, 256
Différence delta :	2 | 0, 3, 4, 9 | 1, 2, 256
Conversion binaire : (16 octets)	0x0 0x2 | 0x0 0x0, 0x0 0x3, 0x0 0x4, 0x0 0x9 | 0x0 0x1, 0x0 0x2, 0x1 0x0
Stream VByte : (9 octets + 2 octets)	0x2 | 0x0, 0x3, 0x4, 0x9 | 0x1, 0x2, 0x1 0x0 + 00, 00, 00, 00, 00, 00, 00, 01

FIGURE 8 – Exemple de l'algorithme Stream VByte*$^{-1}$

Cinquième partie
Travail effectué

Ce TER étant un sujet de recherche une partie non négligeable a consisté à lire et comprendre les articles Lemire *et al.* (2017) et Lemire et Boytsov (2021) mis à ma disposition par mes encadrants. C'est dans ces articles j'ai pu découvrir les algorithmes de différence delta, VByte, stream VByte, SIMD-BP128, SIMD-FastPFOR et simplePFOR, cela m'a pris un certain temps étant donné qu'il n'y avait aucun pseudo-code, il était expliqué uniquement a l'écrit. J'ai également étudié d'autres algorithmes plus connus comme Huffman et le codage par plages.

Puis j'ai commencé à implémenter en java Huffman, le codage par plages ainsi que la différence delta. S'en est suivie l'implémentation de l'algorithme Stream VByte qui a pour particularité de transférer des éléments sur des octets de différentes tailles.

Après discussions avec mes encadrants nous avons décidé de nous tourner vers le complémentaire des ensembles. En effet comme nous disposions de données croissantes et sans doublon nous pouvions donc exploiter cette propriété très intéressante. Comme dit précédemment c'est le seul, avec le codage par plages, permettant de réduire la quantité de nombres à transférer. Jusque-là pour Huffman il fallait transférer un code par élément plus l'arbre du code donc finalement une quantité plus grande d'éléments, mais chaque élément de plus petite taille. La différence delta transfère exactement le même nombre d'éléments. Quant au codage par plage il permet bien de réduire la quantité d'éléments à transférer, mais dans le pire des cas il double la quantité, c'est un cas assez fréquent, et il est difficile d'en faire des heuristiques qui restent décompressables.

À ce stade les algorithmes qui semblaient avoir les meilleurs résultats étaient le complémentaire et la différence delta. Là nous est venue l'idée de composer différentes méthodes notamment le codage par plage et la différence delta, mais surtout ce que j'ai appelé précédemment l'algorithme Stream VByte^{*-1}. Trouver l'idée ainsi que faire la liaison entre chaque étape m'a demandé beaucoup de temps et de concentration en effet trouver un moyen de garantir le bon enchaînement n'était pas trivial.

En parallèle de cela, il a fallu tester chaque algorithme implémenté sur les différents sets de données créés précédemment.

Malheureusement, par manque de temps je n'ai pas pu implémenter tous les algorithmes présentés. Je regrette notamment de ne pas avoir pu travailler sur une partie de ceux de Lemire qui sont particulièrement intéressants. J'aurais aussi aimé implémenter plus de compositions différentes.
Vous pouvez également retrouver l'intégralité de mon travail sur https://github.com/margauxschmied/TER.

Sixième partie
Expérimentation

Afin d'évaluer les performances des algorithmes implémentés, il a fallu les tester sur les sets de données présentés précédemment. Le set de données réelles comporte 197 instances, la plupart étant de petite taille, le tableau récapitulatif des résultats ne montre que 25 instances significatives dont 5 très grandes instances.

Vous pouvez voir les tailles avant et après compression dans les tableaux 9, 10 et 11 pour les sets de données réelles, artificielles croissantes et artificielles pseudo croissantes. Dans ces tableaux chaque ligne est un set de donnée, les cases vertes représentent les tailles les plus petites et les cases rouges sont les tailles les plus grandes. Les tailles sont le nombre de bits après chaque type de compression.
Dans les tableaux 13, 14, 15, 16, 17 et 18 vous pouvez également voir le temps de compression et décompression des mêmes sets de données en bit par nanoseconde. Comme pour la taille chaque ligne est un set de données les cases vertes représentent les plus petits temps et les cases rouges les plus grands temps.

On peut voir dans les tableaux 9, 13 et 14 que le complémentaire a vraiment un très mauvais résultat que ce soit pour le temps ou la taille donc pour des raisons de temps je ne l'ai pas appliqué aux sets de données artificielles.

Toutes les expériences ont été réalisées sur mon ordinateur sous macOS Monterey version 12.4 avec un processeur 2,6 GHz Intel Core i7 6 cœurs et une mémoire 16 Go 2667 MHz DDR4.

12.1 Taille après compression

L'objectif premier d'une compression est de réduire la taille du set de données que ce soit pour les stocker ou les transférer.

Original	Complémentaire	Comp. succ.	Stream VByte^{*-1}	Diff	Huffman	Plage	Plage + Diff	Stream VByte
16	24	24	30	16	34	32	16	20
16	24	24	30	16	34	32	32	20
16	24	24	30	16	34	32	32	20
16	24	24	30	16	34	32	32	20
32	48	48	30	32	66	64	64	20
16	24	24	30	16	34	32	32	20
16	24	24	30	16	34	32	32	20
16	24	24	30	16	34	32	32	20
16	24	24	30	16	34	32	32	20
32	48	48	30	32	66	64	64	20
16	24	24	30	16	34	32	32	20
16	24	24	30	16	34	32	32	20
32	48	48	30	32	66	64	64	20
32	48	48	30	32	66	64	64	20
32	48	48	30	32	66	64	64	20
32	48	48	30	32	66	64	64	20
16	24	24	30	16	34	32	32	20
32	48	48	30	32	66	64	64	20
32	48	48	30	32	66	64	64	20
16	24	24	30	16	34	32	32	20
32	48	48	30	32	66	64	64	20
404664	4864104	203880	153214	269776	567441	809328	210336	404498
132768	9707712	92424	74766	132768	168899	265536	151296	131688
258072	9375672	147072	113560	258072	349326	516144	232560	257434
102864	10369056	72936	59142	102864	126919	205728	119856	100684
65976	7194912	37800	26718	65976	76574	131952	59232	57066

FIGURE 9 – Taille de chaque set de donnée réelle pour chaque type de compression.

Dans le tableau 9 représentant les données réelles on peut voir que pour une majorité de sets de taille 2 la meilleure compression est de ne pas compresser ou bien d'appliquer la différence delta. En effet cela signifie que tous les autres augmentent la taille des éléments à transférer. Huffman augmente particulièrement la taille cela peut être expliqué par le tableau de code transféré. Comme il n'y a que deux éléments dans ces données il n'y a pas de répétition donc le code comprenant chaque élément ainsi que son code associé est déjà plus grand que l'original. On notera que Stream Vbyte a tout de même de bons résultats sur les petits sets car certains de ces sets ont de très grands écarts de valeur, il leur permet donc d'être stockés sur un nombre d'octets différent.
Pour ce qui est des 5 grands sets Stream VByte^{*-1} est le meilleur avec un taux de compression de presque 2.26 et le pire est de loin le complémentaire suivi du codage par plage.

Le tableau 10 représentant le set des données artificielles pseudo croissantes est trié en partant du plus petit nombre d'éléments avec le plus petit intervalle jusqu'au plus grand nombre d'éléments avec le plus grand intervalle. Ici globalement les résultats sont plutôt unanimes il vaut mieux ne pas les compresser ou utiliser la différence delta. Les algorithmes ayant les moins bonnes performances restent le codage de Huffman et le codage par plage car il y a trop peu de valeurs qui se répètent.

Pour finir dans le tableau 11 des données artificielles croissantes, on peut y voir clairement que la différence est la meilleure. Comme il n'y a pas de pallier on peut soustraire à chaque élément tous ses prédécesseurs ce qui les réduira grandement. Le codage de Huffman et le codage par plage restent les algorithmes ayant les moins bonnes performances.

Original	Comp. succ.	Stream VByte^{*-1}	Diff	Huffman	Plage	Plage + Diff	Stream VByte
80	88	110	80	320	160	160	100
160	176	174	160	354	320	320	172
160	176	190	160	320	288	320	180
240	264	254	240	464	432	480	244
240	264	270	240	514	480	480	260
320	352	270	320	674	640	640	260
1600	1616	1722	1600	4413	3200	3136	1744
1600	1616	1810	1600	5222	3200	3200	1800
2400	2424	2410	2400	5422	4800	4800	2480
2400	2424	2594	2400	5472	4800	4800	2600
3200	3232	2610	2400	7072	6400	4800	2600
16000	16016	17954	16000	47410	32000	31872	17992
24000	24024	24922	24000	63488	48000	48000	24928
24000	24024	25962	24000	72326	48000	48000	25984
32000	32032	25986	32000	73778	64000	64000	25976
240000	240024	250994	240000	196165	480000	477120	251456
240000	240024	259482	240000	257514	480000	479712	259480
320000	320032	259962	320000	315551	640000	639936	259960

FIGURE 10 – Taille de chaque set de donnée artificiel pseudo croissante pour chaque type de compression.

Original	Comp. succ.	Stream VByte^{*-1}	Diff	Huffman	Plage	Plage + Diff	Stream VByte
80	88	110	80	354	160	144	100
160	176	174	160	354	320	320	172
160	176	190	160	354	320	320	180
240	264	246	240	514	480	480	244
240	264	270	240	514	480	480	260
240	264	270	240	514	480	480	260
1600	1616	1706	800	5472	3200	1488	1688
1600	1616	1802	1600	5472	3200	3136	1792
2400	2424	2154	1600	5472	4800	3200	2160
2400	2424	2562	2400	5472	4800	4800	2584
3200	3232	2602	2400	7072	6400	4800	2592
16000	15984	17622	8000	73976	32000	15136	17776
24000	24024	20658	16000	73976	48000	31872	20656
24000	24024	25458	16000	73976	48000	31968	25456
32000	32032	25978	24000	73976	64000	48000	25968
240000	239736	205658	80000	316166	480000	151824	207984
240000	240024	254490	160000	326102	480000	318272	254560
320000	320032	259498	160000	336130	640000	319936	259488
2400000	2397696	2517528	800000	3926845	4800000	1515824	2547640
3200000	3200000	2594312	1600000	4026571	6400000	3183872	2594776

FIGURE 11 – Taille de chaque set de donnée artificiel strictement croissant pour chaque type de compression.

12.2 Temps de compression et de décompression

En plus de la taille finale, le temps de compression et décompression nous intéresse particulièrement, notamment dans le cas où l'objectif est de transférer rapidement les informations. La taille va être primordiale à cause du temps de transfert par bits, mais le temps de compression et de décompression n'est pas à négliger. Par exemple le temps de compression et décompression de la méthode avec le meilleur taux de compression peut dépasser à eux seul le temps total de transfert des données originales ce qui en ferait une mauvaise méthode.

Le tableau 13 du temps de compression des données réelles nous montre clairement que le plus rapide est la différence delta. Cela s'explique par le fait qu'elle n'est composée que d'opérations élémentaires. Le plus lent est sans surprise Stream VByte^{*-1}, comme il est composé de plusieurs algorithmes il peut difficilement être plus rapide que chacun d'eux indépendamment.
Si on s'intéresse maintenant au temps de décompression des données dans le tableau 14. Le codage par plage est le plus rapide suivi par la différence delta et le plus lent et globalement Stream VByte. Que ce soit pour la compression ou la décompression le complémentaire et aussi très long. Ce résultat était attendu puisque cet algorithme doit traiter tous les entiers entre les valeurs minimums et maximales de chaque palier.

Pour les données artificielles pseudo croissantes 15 et strictement croissants 17 les méthodes les plus rapides et les moins rapides ont un temps de compression très similaire. La différence delta reste le plus rapide et cette fois c'est Huffman qui est le plus lent. Comme c'est un temps par bit compressé et qu'il produit les plus grandes données, c'est naturellement lui le plus lent par bit.
Pour ce qui est du temps de décompression pour les données pseudo croissantes 16 les meilleurs sont la différence delta et le codage par plage et le plus lent est Stream Vbyte. Pour les données strictement croissantes 18 le plus rapide à décompresser est le codage par plage et le plus lent est Stream VByte.

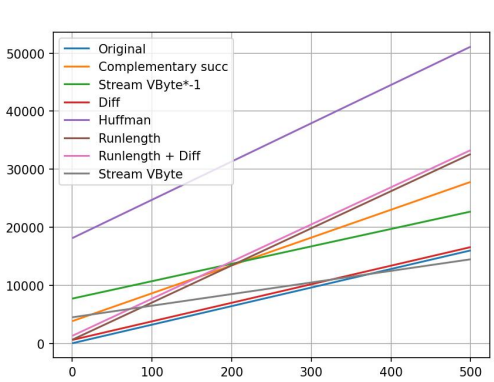
12.3 Transfert des informations

Maintenant, essayons de faire varier le temps de transfert par bit afin de voir quelle méthode est la meilleure pour quel temps. Pour cela j'ai réalisé un graphique par set de données montrant la variation du temps total en fonction du temps de transfert par bit pour chaque méthode de compression et chaque type de set de données (figures 12, 21, 24). Les fonctions suivent donc (temps total de compression + temps total de décompression + temps de transfert part bit * nombre de bits total après compression). Dans ces graphiques je fais varier le temps de transmission de 0 à 500 nanosecondes.

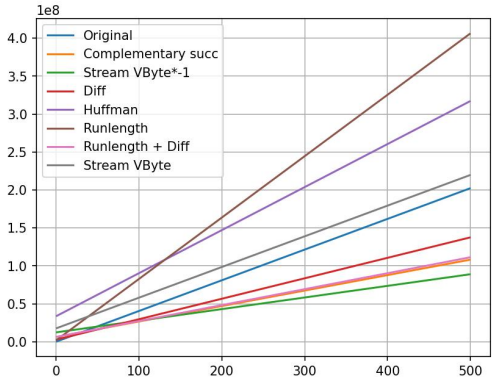
12.3.1 Données réelles

Afin de simplifier la compréhension, pour le set de données réelles j'ai choisi de représenter uniquement un set de taille 2 (figure 12a) et les 5 plus gros sets de données réelles (figures 12b, 12c, 12d, 12e et 12f).

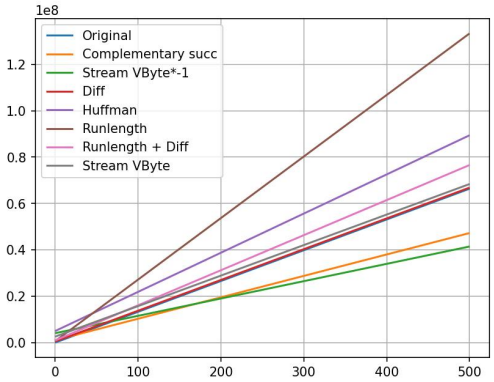
Un premier point intéressant à relever est qu'il y a toujours un algorithme de compression plus performant que de transférer simplement l'original. Pour le set de données de 2 éléments Stream VByte est le meilleur à partir de 384 nanosecondes de transmission, en deçà il vaut



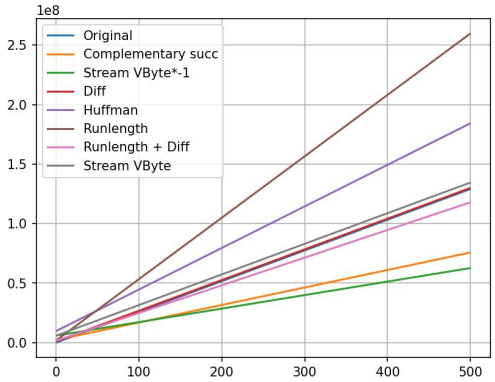
(a) Set de donnée originellement constitué de 2 éléments.



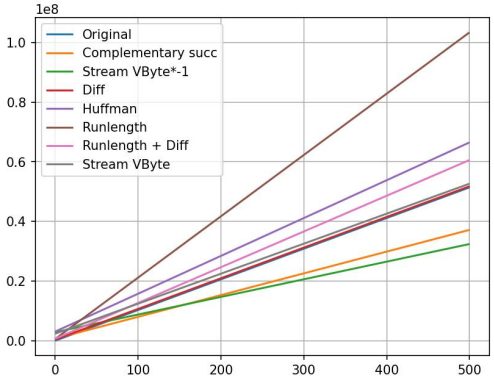
(b) Set de donnée originellement constitué de 16861 éléments.



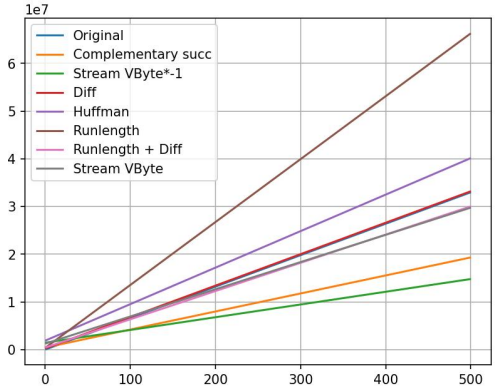
(c) Set de donnée originellement constitué de 5532 éléments.



(d) Set de donnée originellement constitué de 10753 éléments.



(e) Set de donnée originellement constitué de 4286 éléments.



(f) Set de donnée originellement constitué de 2749 éléments.

FIGURE 12 – Graphique représentant le temps de compression, décompression et transfert pour les cinq plus gros set de données réelles ainsi qu'un petit set de données réelles.

mieux transférer directement l'original. Pour les 5 autres sets de données les écarts sont plus ou moins marqués mais à terme Stream VByte^{*-1} est le meilleur pour chacun d'entre eux. Stream VByte^{*-1} se démarque à respectivement 120, 207, 146, 172 et 117 nanosecondes.
On peut donc dire que sur un gros set de données réelles l'algorithme Stream VByte^{*-1} est le meilleur pour faire du transfert de données.
Sur les graphiques des 5 gros sets on peut remarquer que le second meilleur de prêt est la compression successive. Stream VByte^{*-1} est construit en partie avec cet algorithme, on peut supposé que la compression successive a le plus d'impact sur ces performances. Il pourrait être intéressant d'essayer d'améliorer les autres parties afin de creuser l'écart avec la compression successive.

12.3.2 Données artificielles pseudo croissantes

On peut voir les graphiques représentants le temps de transfert des données artificielles pseudo croissantes dans la figure 21. Contrairement aux données réelles la meilleur option n'est pas toujours de compresser les données. Si on analyse de plus près chaque graphique on remarque qu'il devient intéressant de compresser les données pour un intervalle large, à partir de 100 000 valeurs pour les grands ensembles sinon à partir de 1 000 000 valeurs.

Le set de données 20d avec 100 éléments et un intervalle de 1 000 000 est le premier qui est plus rapide en étant compressé. La méthode différence delta est ici la meilleur pour un temps de transfert de 50 nanosecondes. Pour les tableaux 21b et 21f représentant respectivement des données de 1 000 éléments avec un intervalles de 10 000 000 et des données de 10 000 éléments avec un intervalles de 10 000 000 la meilleur compression est Steam VByte pour 193 et 94 nanosecondes. Le dernier set qu'il vaut mieux compresser est celui du tableau 21c pour les données avec 10 000 éléments et un intervalle de 100 000. Dans ce cas la meilleure méthode est d'utiliser Huffman qui est la plus rapide à partir de 285 nanosecondes. Tous les autres sets sont plus rapides à transférer sans les compresser.

12.3.3 Données artificielles strictement croissantes

Si on analyse maintenant les données strictement croissantes à partir de plus de 10 éléments transféré l'original est plus lent que de passer par une méthode de compression. En l'occurrence la meilleure méthode est de loin la différence delta. Quel que soit le set c'est toujours la plus rapide et à chaque fois pour très peu de temps de transfert selon les sets entre 11 et 41 nanosecondes suffise à le rendre meilleur. Ces données étant strictement croissantes faire la différence est à la fois rapide comme il s'agit uniquement d'opérations élémentaires et performant puisque le plus grand nombre sera réduit de la somme de ces prédécesseurs.

J'ai fait le choix de tester les performances des algorithmes sur la même variation de temps de transfert, de 0 à 500 nanosecondes. Il aurait peut-être été intéressant d'essayer des temps de transfert plus large notamment pour les données artificielles pseudo croissants. Avec une durée plus longue, différence delta n'étant pas loin il pourrait devenir plus rapide.

Septième partie
Conclusion et perspectives

Les algorithmes étudiés ont montré leurs efficacité sur les données générées. Cependant sur les données réelles, leurs performances sont plus mitigées.

Suite à plusieurs discussions avec mes encadrants ainsi que les personnes que j'ai rencontrées au laboratoire I3S et en m'inspirant des algorithmes existants, j'ai proposé 2 méthodes. La méthode de compression successive et Stream VByte^{*-1}. Je suis satisfaite des résultats obtenus avec Stream VByte^{*-1} bien que cette méthode ne soit pas très efficace sur les données que j'ai générées.

Il serait intéressant de continuer et d'étudier d'autres améliorations possibles. En effet, certains algorithmes imbriqués ont peut-être un temps d'exécution long pour un faible gain en terme de taille. Il serait aussi intéressant de combiner d'autres algorithmes de compressions telle que la combinaison de la compression successive avec la différence delta.

Une autre piste intéressante serait d'étudier les compressions possibles (si elles existent) pour le code du tableau servant à la décompression transmise par Huffman. De même pour la suite représentant le nombre d'octets utilisés pour chaque élément par Stream VByte.

Afin de compléter cette étude, il serait intéressant d'implémenter toutes les méthodes présentées dans les travaux de Lemire. Ainsi que de générer plus de sets de données réelles de grande taille afin de confirmer ou d'infirmer les performances de Stream VByte^{*-1}.

Parmi les méthodes existantes que j'ai pu étudier la plus performante globalement que ce soit en matière de taille après compression ou de temps de compression et décompression est la différence delta. Cette méthode est incluse dans plusieurs algorithmes de Lemire, que je n'ai pas implémenté, ce qui me donne d'autant plus envie d'étudier leurs performances et surtout de les comparer à l'algorithme Stream VByte^{*-1}.

Je suis très contente d'avoir pu découvrir le monde de la compression et les travaux de Lemire. Les articles ne contenant aucun pseudo-code cela m'a permis de prendre confiance en mes capacités de compréhension puisque j'ai réussi à en implémenter plusieurs. Ce fut une bonne expérience d'avoir essayé et parfois même réussi à concurrencer ces algorithmes.

Bibliographie

Lemire, D. et Boytsov, L. (2021). Decoding billions of integers per second through vectorization.

Lemire, D., Kurz, N. et Rupp, C. (2017). Stream vbyte : Faster byte-oriented integer compression. *CoRR*, abs/1709.08990.

Huitième partie
Annexe

Complémentaire	Comp. succ.	Stream VByte^{*-1}	Diff	Huffman	Plage	Plage + Diff	Stream VByte
48.3125	399.0	2706.5625	9.9375	421.625	18.4375	75.1875	36.0
49.9375	639.25	2595.1875	25.3125	430.6875	18.875	66.0625	33.0625
1070.5	1784.5	2570.875	10.0625	422.0	18.9375	53.875	31.875
308.3125	420.25	3354.9375	10.0	431.625	17.75	51.375	32.8125
75.6875	176.4375	1446.1875	5.40625	1437.59375	9.0	27.40625	15.3125
117.1875	504.9375	2648.75	10.0	543.8125	18.4375	57.0625	34.125
110.75	253.8125	2765.1875	10.25	425.5625	18.25	50.6875	31.875
103.6875	239.6875	2654.4375	10.125	389.25	23.3125	51.25	32.0
106.375	229.3125	3102.625	10.375	369.25	20.375	53.25	32.1875
52.09375	159.15625	1544.40625	5.25	185.125	9.625	26.46875	14.78125
99.0	249.1875	2733.25	12.0625	351.5625	21.0625	54.5	34.75
97.3125	245.125	2702.5625	10.0	341.0	36.125	51.8125	32.8125
50.875	122.84375	1376.25	5.21875	177.65625	10.4375	27.25	14.71875
48.0625	139.125	5651.71875	5.15625	171.90625	11.28125	25.34375	14.59375
57.71875	152.75	1426.15625	5.125	1129.53125	9.34375	27.96875	15.0
146.03125	122.8125	900.875	5.15625	411.65625	9.09375	26.53125	14.96875
105.0	238.0625	2244.875	10.8125	479.9375	18.1875	176.8125	31.1875
34.4375	116.1875	676.71875	5.46875	298.15625	9.40625	25.84375	14.59375
37.34375	352.03125	942.0	10.09375	867.3125	12.03125	36.875	21.5625
55.4375	284.875	1566.875	13.9375	506.9375	19.25	55.0625	35.5
23.59375	130.59375	1050.625	6.4375	386.125	9.25	24.40625	14.5625
7385.69757	16.9269	28.27419	1.61919	81.40071	4.58502	10.5041	11.93772
7030.21596	5.4763	42.66689	1.65198	25.08014	2.97908	4.22085	9.76261
6389.57075	4.22616	15.47464	1.5884	22.47567	2.71103	3.69821	12.17958
6267.08572	3.5424	14.27311	1.57421	24.11869	2.83786	3.87552	10.98479
4636.10073	3.10736	12.65257	1.62852	23.22967	2.8066	3.47693	9.32247

FIGURE 13 – Temps moyen de compression en nanoseconde par bit de chaque set de donnée réelle pour chaque type de compression.

Complémentaire	Comp. succ.	Stream VByte^{*-1}	Diff	Huffman	Plage	Plage + Diff	Stream VByte
26.0625	29.875	78.5625	27.5625	66.75	21.125	39.5	242.5625
25.75	26.4375	76.75	24.6875	65.1875	21.0625	40.9375	302.875
26.8125	26.5625	77.375	20.25	66.6875	22.375	36.75	304.3125
26.5625	30.1875	78.6875	29.125	64.4375	21.8125	40.0625	270.4375
13.4375	14.25	38.90625	13.3125	46.84375	10.21875	19.03125	133.78125
27.25	28.25	78.3125	20.75	113.375	20.6875	42.125	272.5
26.75	28.9375	79.75	24.0	115.625	25.0	36.625	283.5
26.625	31.0	80.0	22.9375	118.25	19.0	40.375	264.5625
27.1875	24.8125	76.9375	22.5	127.25	19.4375	38.0625	270.5
13.78125	16.21875	38.03125	11.5	61.625	8.21875	20.09375	178.5625
26.625	27.6875	76.9375	23.125	117.125	22.375	37.0	273.25
26.0625	59.6875	77.8125	21.625	115.125	20.5625	38.1875	267.0
14.46875	17.59375	38.96875	12.0	55.3125	9.9375	19.1875	132.0625
14.09375	21.875	39.25	11.90625	57.6875	9.15625	18.6875	126.96875
13.75	61.96875	38.4375	68.1875	58.90625	14.03125	20.21875	190.71875
13.75	13.875	39.34375	13.0	60.5625	8.40625	19.78125	177.625
27.375	29.6875	76.75	23.6875	124.75	21.25	36.5625	341.25
14.125	14.59375	38.5	12.65625	56.625	11.34375	19.0	150.53125
13.40625	12.9375	39.3125	17.75	59.3125	10.25	19.1875	146.03125
27.25	29.5	79.0625	25.5625	115.9375	22.9375	38.5	293.9375
13.4375	14.5625	42.84375	10.71875	57.5	9.09375	19.65625	124.875
42813.89909	3.02684	10.37434	5.10392	12.875	1.43265	6.78193	27.35639
241097.39506	3.01141	14.11174	1.44501	7.64388	1.77879	3.40703	8.93239
131454.14084	3.14365	10.44857	1.33624	7.55658	2.45023	3.13339	9.46725
322140.16172	3.01829	10.81782	1.38619	6.93415	1.3755	3.27464	12.493
315872.0099	2.3788	9.88167	1.34672	7.11136	1.3912	3.12786	10.42044

FIGURE 14 – Temps moyen de decompression en nanoseconde par bit de chaque set de donnée réelle pour chaque type de compression.

Comp. succ.	Stream VByte^{*-1}	Diff	Huffman	Plage	Plage + Diff	Stream VByte
576.0125	6080.1375	58.6125	10363.075	192.9625	209.7125	88.425
95.34375	335.43125	22.43125	456.2	22.38125	37.5125	22.45625
84.08125	216.9625	11.6875	695.70625	20.2	108.3	17.65625
65.35417	152.4625	6.60417	440.67917	12.33333	25.79583	15.56667
56.29583	202.25	6.4875	241.95833	11.95417	19.33333	13.34167
37.6375	115.11562	4.30938	151.44688	9.21562	13.80937	7.78437
57.13938	172.78375	7.90313	281.035	12.07563	20.5225	14.65125
55.44625	174.09875	7.6275	166.64	12.33687	21.23688	20.00125
40.2525	102.6825	5.44333	184.09083	8.37542	12.98083	20.32375
27.20667	84.67125	34.96792	150.3025	8.12167	14.17958	10.19292
14.07156	58.425	4.01344	124.88469	6.56	10.72406	10.19344
20.03675	62.54644	7.38694	71.74438	12.03688	19.77894	17.80319
13.54558	53.14454	4.78792	64.66196	7.54529	13.22492	12.08996
13.29558	46.57854	4.75279	61.11658	7.42321	13.07375	8.62417
9.86569	30.56466	3.53987	126.81509	5.53353	9.82328	6.94038
14.73137	37.4124	4.88236	39.30635	6.90022	13.19489	7.18017
23.8973	26.83989	4.65282	47.39965	6.68455	11.61774	7.68389
8.29886	17.30776	3.15472	39.89062	4.98587	8.18381	6.16433

FIGURE 15 – Temps moyen de compression en nanoseconde par bit de chaque set de donnée artificiel pseudo croissante pour chaque type de compression.

Comp. succ.	Stream VByte*$^{-1}$	Diff	Huffman	Plage	Plage + Diff	Stream VByte
104.1	416.0	530.525	681.3	49.5375	176.7625	796.3875
49.75	116.28125	20.74375	35.76875	19.23125	48.5125	606.80625
30.5625	55.5375	15.18125	25.675	11.5625	20.20625	195.91875
33.52083	33.8125	7.2125	19.30417	7.00833	12.1625	99.95417
22.6625	46.625	6.9125	15.1125	19.17083	11.52083	111.09167
11.9625	42.24688	5.3875	11.34375	9.17812	15.20312	96.15937
34.38063	40.89625	7.625	17.76938	8.31125	15.31563	116.9
13.66313	34.49	8.27937	15.77562	23.91375	15.35	123.925
15.46125	30.1375	5.29583	11.95292	5.70875	9.96583	123.4825
8.45667	22.93333	5.17667	11.22042	11.98042	25.49833	39.87208
6.30187	17.21219	3.93438	8.35938	4.01406	8.11313	17.75656
11.67819	70.00719	7.01531	13.34081	4.56581	13.86412	26.68044
7.38567	38.38938	4.71171	10.56237	2.84675	9.10458	24.03054
7.44958	29.73304	4.62458	14.11667	1.98592	10.60767	33.92621
5.574	22.47384	3.59716	9.45856	1.13163	6.79259	24.907
7.20449	20.43962	4.94191	8.6183	1.69255	9.24988	13.4132
7.00307	21.5746	5.11104	8.97142	1.72341	8.97238	12.00663
5.05868	15.07815	5.04893	9.11827	1.15611	6.85685	11.77792

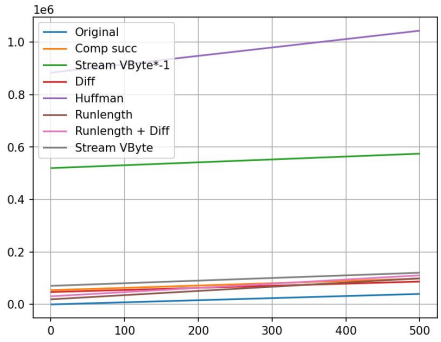
FIGURE 16 – Temps moyen de decompression en nanoseconde par bit de chaque set de donnée artificiel pseudo croissante pour chaque type de compression.

Comp. succ.	Stream VByte*$^{-1}$	Diff	Huffman	Plage	Plage + Diff	Stream VByte
670.1	10801.05	86.275	15460.75	143.7125	230.75	77.475
132.4375	837.95	7.1125	556.59375	6.125	45.58125	12.075
103.55	267.31875	3.2875	1434.85625	3.01875	29.975	9.83125
84.6375	208.0625	1.6125	704.5625	1.71667	18.4125	8.5875
74.87917	150.35	1.23333	357.40417	1.3	17.73333	7.0375
66.62917	747.88333	3.075	223.85833	1.4375	109.9	6.9875
79.87188	187.15938	6.20188	337.90938	1.52812	19.38938	5.51938
71.86	127.32438	1.10375	220.96938	1.18375	20.15063	4.9825
51.64042	72.44792	0.53083	193.26292	0.73625	17.4375	3.77958
34.30083	101.76333	0.67458	131.69875	0.68125	13.93917	5.02875
18.53375	139.88594	0.30969	96.9775	0.51531	9.77063	3.45719
285.00325	50.01087	0.41044	105.47837	2.72869	18.10481	6.08719
31.28508	32.07637	0.40063	82.02463	0.64825	13.1615	3.69383
97.63712	42.80842	0.31692	69.28033	0.64117	12.74721	4.41204
20.63384	20.75947	0.19391	38.59972	0.47141	9.13588	2.97112
28.3611	24.79683	0.35519	48.24662	0.66478	12.44985	3.82897
37.03278	17.61448	0.38105	48.94214	0.7739	11.57382	5.57305
11.95698	8.44701	0.23858	27.20972	0.59334	8.72123	3.22088
9.79478	19.59923	0.24853	27.47749	0.76063	5.17157	4.03239
2.41797	5.49911	0.30439	18.42445	0.51876	0.8867	3.21481

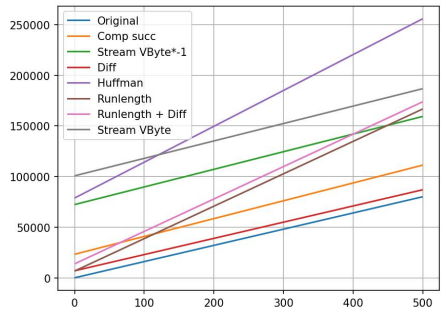
FIGURE 17 – Temps moyen de compression en nanoseconde par bit de chaque set de donnée artificiel strictement croissant pour chaque type de compression.

Comp. succ.	Stream VByte*$^{-1}$	Diff	Huffman	Plage	Plage + Diff	Stream VByte
58.1625	304.4875	149.6	1294.325	15.7625	186.6375	1060.25
8.4125	50.475	21.8375	33.91875	3.53125	18.3375	186.86875
5.0	268.45625	9.9625	22.68125	2.7375	13.7625	137.0
3.20417	27.10833	6.02083	14.20417	1.18333	7.99167	111.77917
2.375	23.61667	5.9125	15.025	1.3125	11.1125	90.50833
2.14167	23.825	6.05	12.97083	165.94583	109.35417	102.45417
5.7125	418.38063	6.94688	16.41	0.99438	33.3975	88.2275
2.41562	44.68812	6.33625	15.20625	0.775	21.485	76.8125
0.8225	17.90542	4.58458	10.57292	0.43917	5.71708	58.80292
0.78333	16.49875	4.29042	10.04625	0.42583	5.33292	39.16125
0.96906	11.59094	3.19281	8.98969	0.30781	4.09219	29.27594
1.46194	20.31619	5.8595	14.38588	0.45844	11.35787	43.97825
0.93017	13.14517	4.02604	9.70033	0.30325	4.87771	17.32479
0.92271	13.75021	3.88542	10.15721	0.29858	4.77538	15.63125
1.56337	9.87337	2.93444	7.67009	0.218	4.90072	11.71053
1.01499	20.37388	3.83663	9.98165	0.29078	4.89037	16.20328
0.97348	12.20661	3.87241	9.997	0.29832	4.66993	14.06988
0.71038	9.08675	2.89798	8.06835	0.38743	3.87478	10.44268
0.95186	10.17932	2.08838	10.65143	0.31887	7.33016	12.38317
0.69714	4.46332	0.33554	8.40638	0.23015	0.53721	6.90448

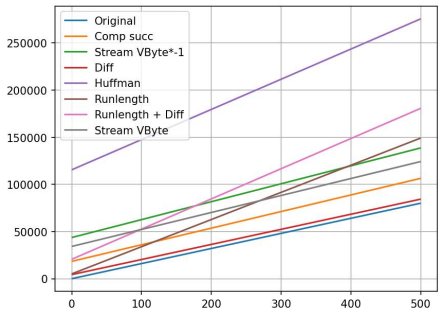
FIGURE 18 – Temps moyen de decompression en nanoseconde par bit de chaque set de donnée artificiel strictement croissant pour chaque type de compression.



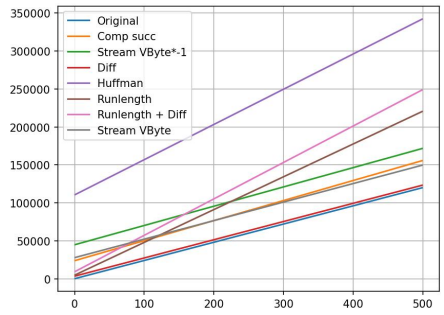
(a) Set de donnée originellement constitué de 10 éléments allant de 0 à 100.



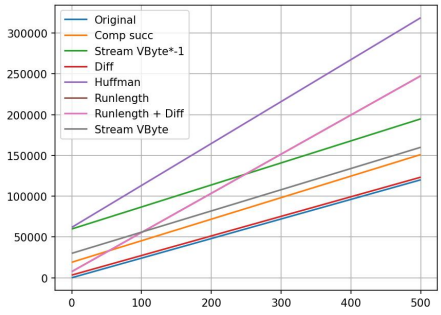
(b) Set de donnée originellement constitué de 10 éléments allant de 0 à 1 000.



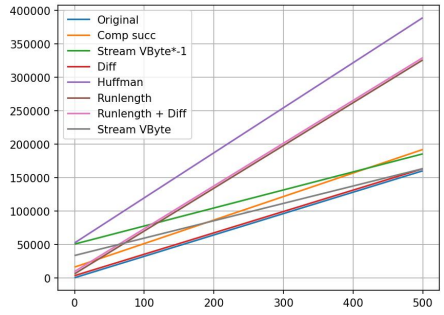
(c) Set de donnée originellement constitué de 10 éléments allant de 0 à 10 000.



(d) Set de donnée originellement constitué de 10 éléments allant de 0 à 100 000.

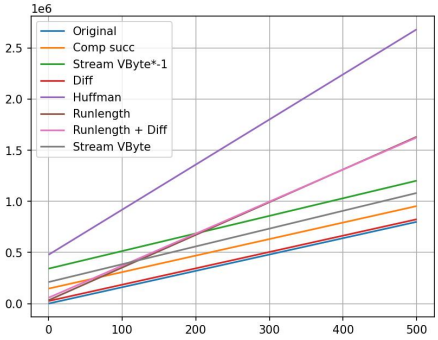


(e) Set de donnée originellement constitué de 10 éléments allant de 0 à 1 000 000.

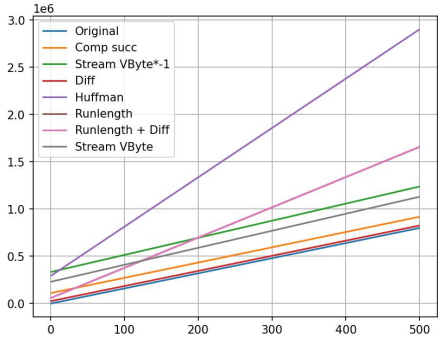


(f) Set de donnée originellement constitué de 10 éléments allant de 0 à 10 000 000.

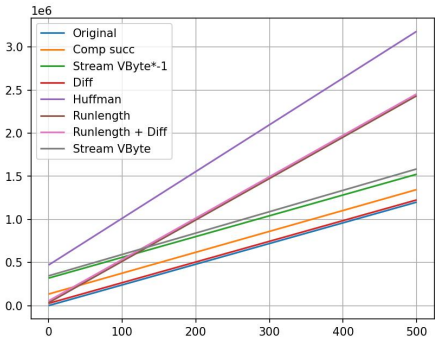
FIGURE 19 – Graphique représentant le temps de compression, décompression et transfert pour les 6 sets de données artificielles pseudo croissantes d'ensembles de 10 éléments.



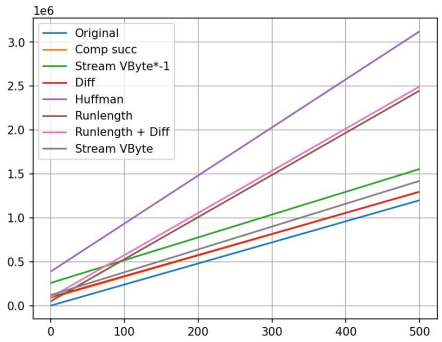
(a) Set de donnée originellement constitué de 100 éléments allant de 0 à 1 000.



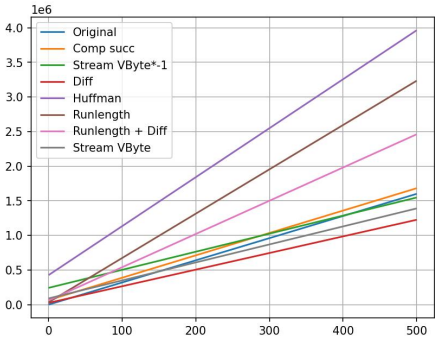
(b) Set de donnée originellement constitué de 100 éléments allant de 0 à 10 000.



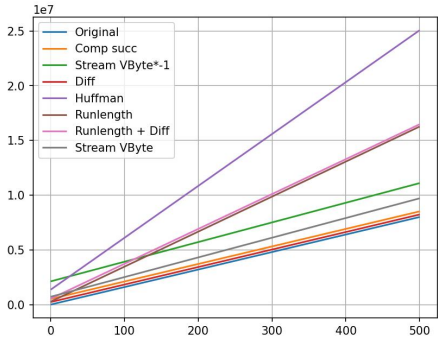
(c) Set de donnée originellement constitué de 100 éléments allant de 0 à 100 000.



(d) Set de donnée originellement constitué de 100 éléments allant de 0 à 1 000 000.

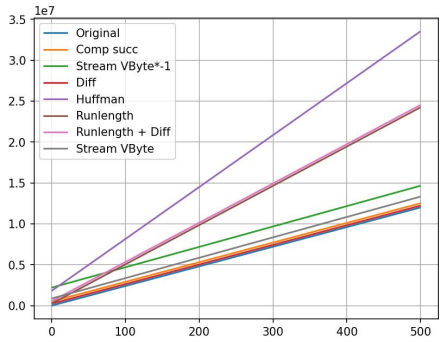


(e) Set de donnée originellement constitué de 100 éléments allant de 0 à 1 000 000.

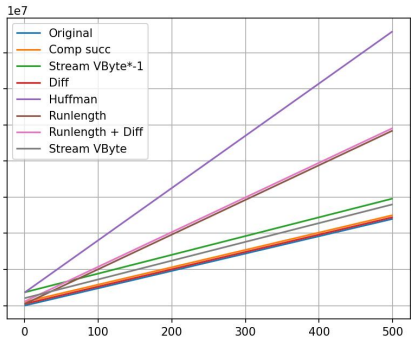


(f) Set de donnée originellement constitué de 1 000 éléments allant de 0 à 10 000.

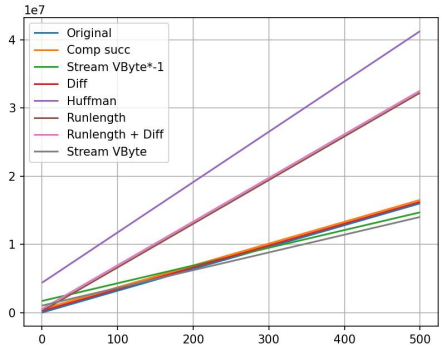
FIGURE 20 – Graphique représentant le temps de compression, décompression et transfert pour les 5 sets de données artificielles pseudo croissantes d'ensembles de 100 éléments et 1 set de donnée de 1 000 éléments.



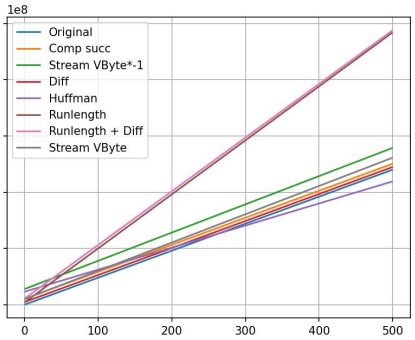
(a) Set de donnée originellement constitué de 1 000 éléments allant de 0 à 100 000.



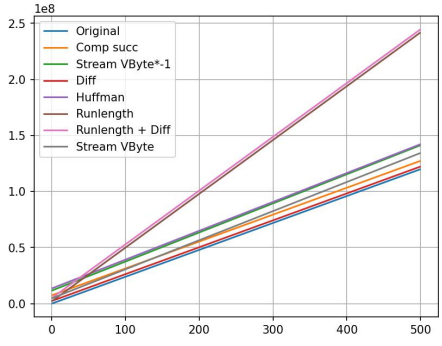
(b) Set de donnée originellement constitué de 1 000 éléments allant de 0 à 1 000 000.



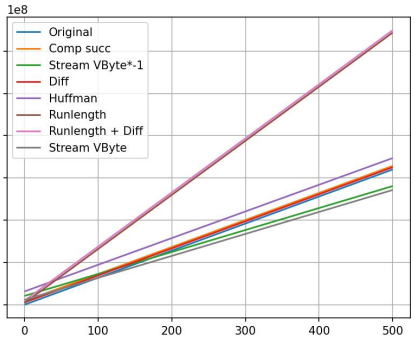
(c) Set de donnée originellement constitué de 1 000 éléments allant de 0 à 10 000 000.



(d) Set de donnée originellement constitué de 10 000 éléments allant de 0 à 100 000.

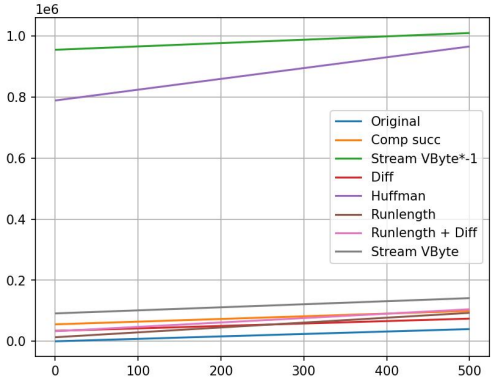


(e) Set de donnée originellement constitué de 10 000 éléments allant de 0 à 1 000 000.

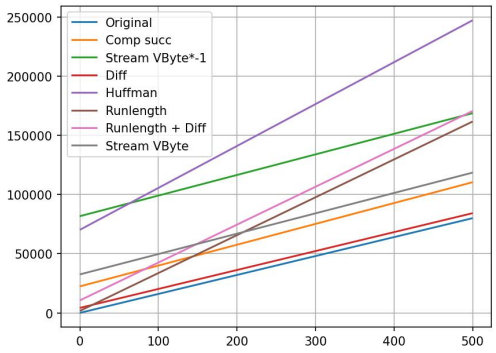


(f) Set de donnée originellement constitué de 10 000 éléments allant de 0 à 10 000 000.

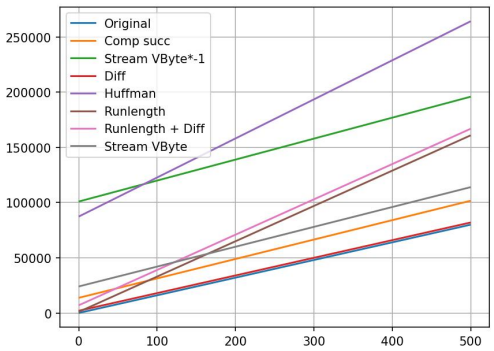
FIGURE 21 – Graphique représentant le temps de compression, décompression et transfert pour les 6 sets de données artificielles pseudo croissantes d'ensembles de plus de 1000 éléments.



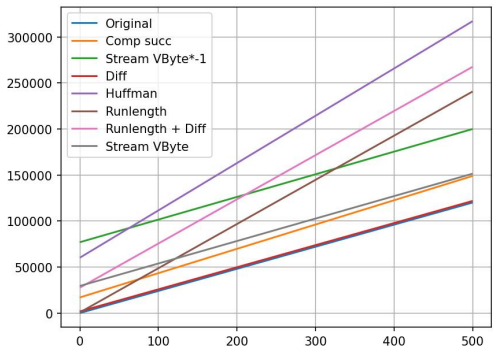
(a) Set de donnée originellement constitué de 10 éléments allant de 0 à 100.



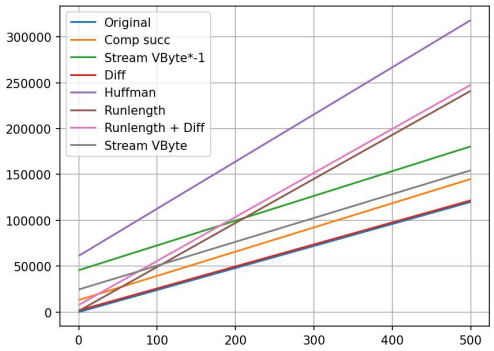
(b) Set de donnée originellement constitué de 10 éléments allant de 0 à 1 000.



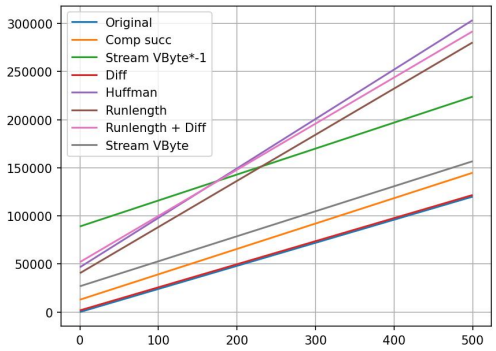
(c) Set de donnée originellement constitué de 10 éléments allant de 0 à 10 000.



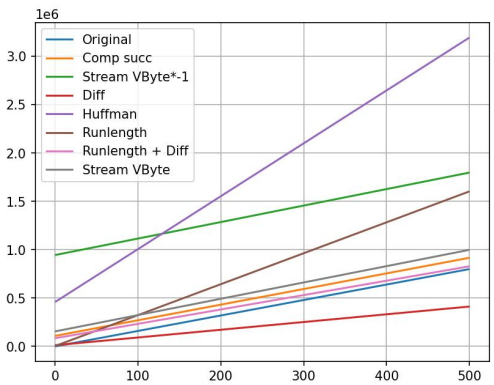
(d) Set de donnée originellement constitué de 10 éléments allant de 0 à 100 000.



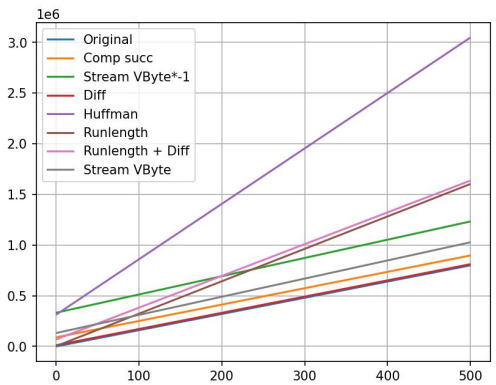
(e) Set de donnée originellement constitué de 10 éléments allant de 0 à 1 000 000.



(f) Set de donnée originellement constitué de 10 éléments allant de 0 à 10 000 000.

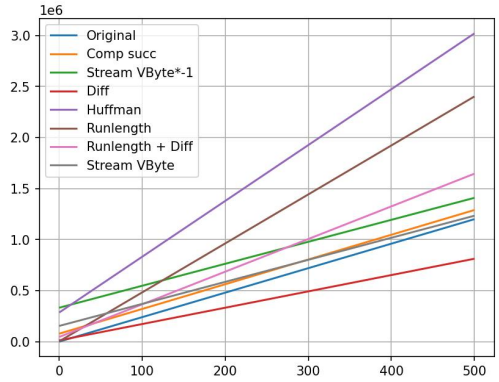


(g) Set de donnée originellement constitué de 100 éléments allant de 0 à 1 000.

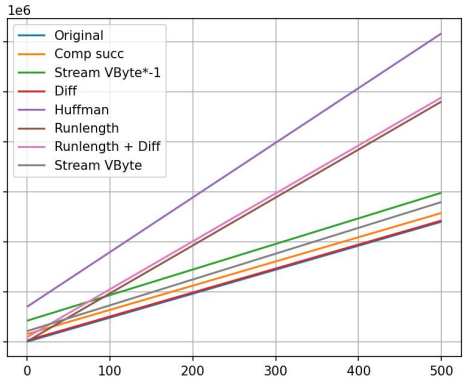


(h) Set de donnée originellement constitué de 100 éléments allant de 0 à 10 000.

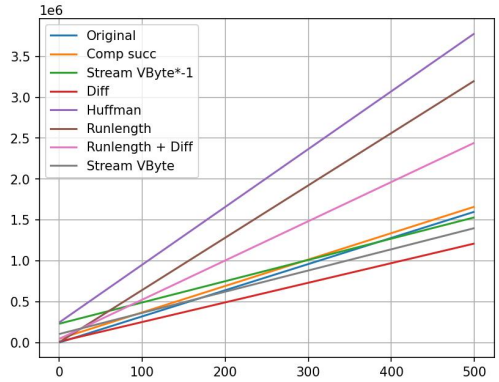
FIGURE 22 – Graphique représentant le temps de compression, décompression et transfert pour les 4 sets de données artificielles strictement croissantes d'ensembles de 10 éléments et 2 sets de données de 1 000 éléments.



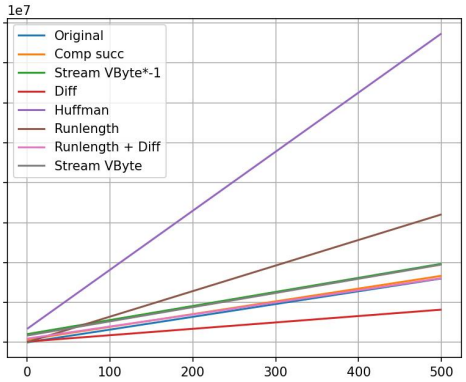
(a) Set de donnée originellement constitué de 100 éléments allant de 0 à 100 000.



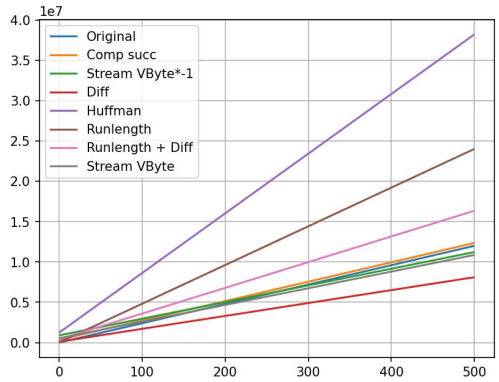
(b) Set de donnée originellement constitué de 100 éléments allant de 0 à 1 000 000.



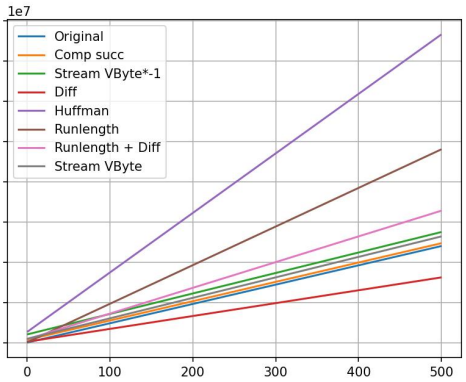
(c) Set de donnée originellement constitué de 100 éléments allant de 0 à 10 000 000.



(d) Set de donnée originellement constitué de 1000 éléments allant de 0 à 10 000.

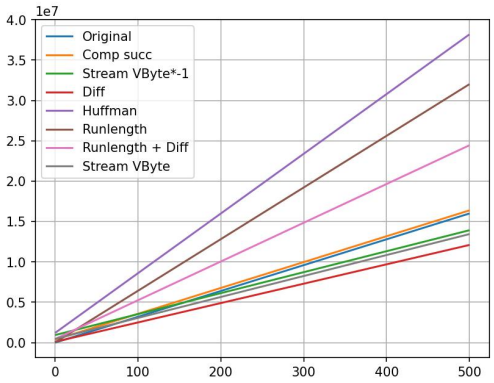


(e) Set de donnée originellement constitué de 1 000 éléments allant de 0 à 100 000.

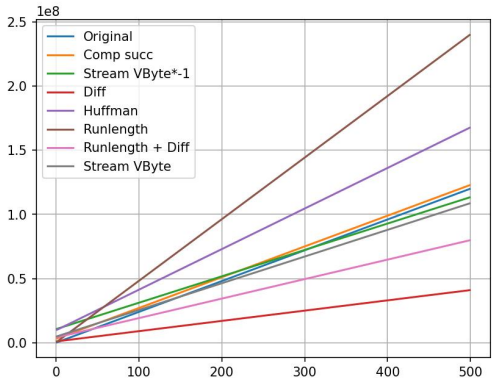


(f) Set de donnée originellement constitué de 1 000 éléments allant de 0 à 1 000 000.

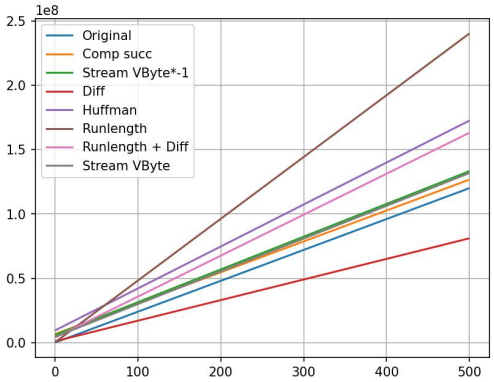
FIGURE 23 – Graphique représentant le temps de compression, décompression et transfert pour les 3 sets de données artificielles strictement croissantes d'ensembles de 100 éléments et 3 sets de données de 1 000 éléments.



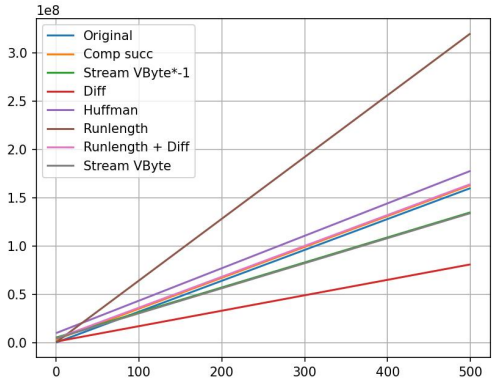
(a) Set de donnée originellement constitué de 1 000 éléments allant de 0 à 10 000 000.



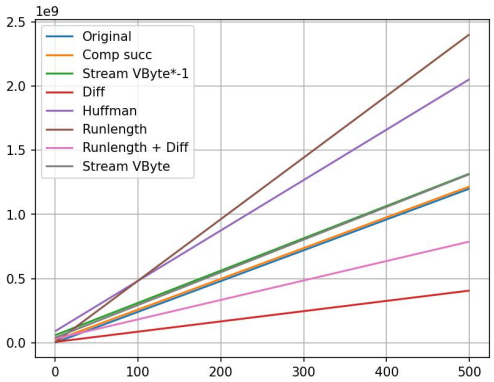
(b) Set de donnée originellement constitué de 10 000 éléments allant de 0 à 100 000.



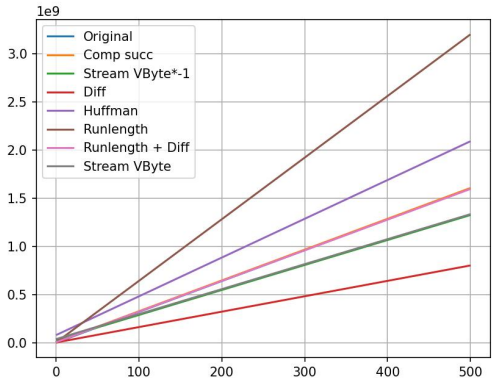
(c) Set de donnée originellement constitué de 10 000 éléments allant de 0 à 1 000 000.



(d) Set de donnée originellement constitué de 10 000 éléments allant de 0 à 10 000 000.



(e) Set de donnée originellement constitué de 100 000 éléments allant de 0 à 1 000 000.



(f) Set de donnée originellement constitué de 100 000 éléments allant de 0 à 10 000 000.

FIGURE 24 – Graphique représentant le temps de compression, décompression et transfert pour les 6 sets de données artificielles strictement croissantes d'ensembles de plus de 1000 éléments.