UE ALG

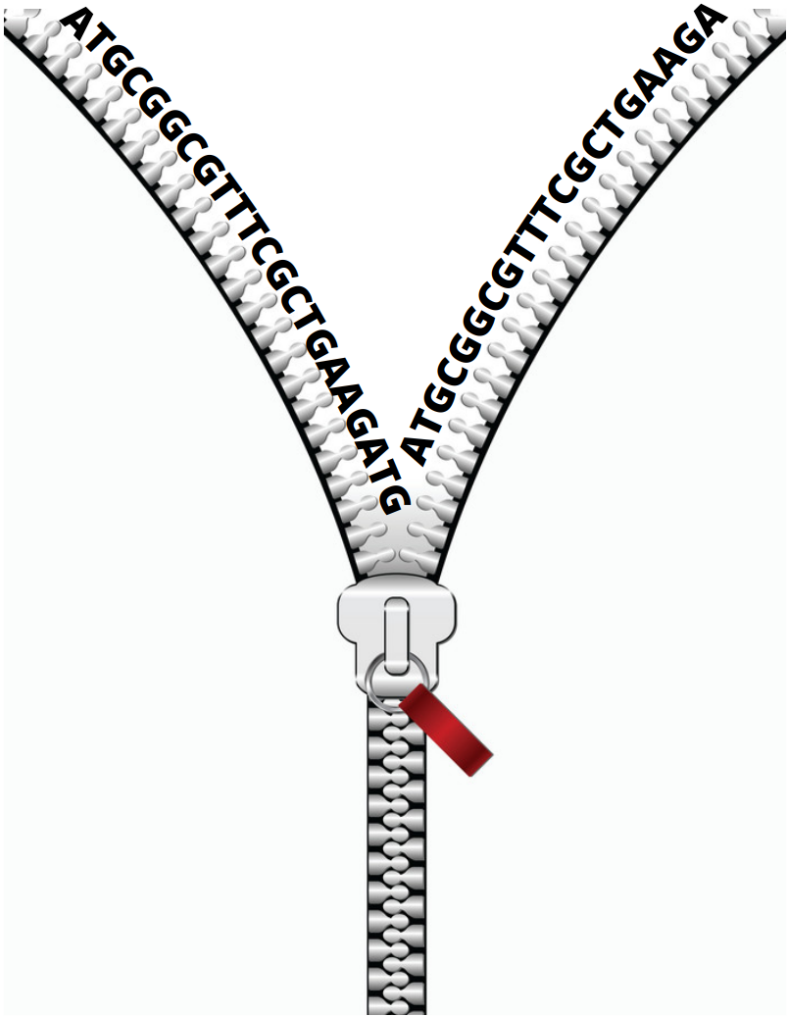
Master 2 Bio-Informatique parcours IBI

Université de Rennes 1

Siegfried DUBOIS et Moana AULAGNER

2022 - 2023

Projet ’read organizer’



1 Introduction

Avec l'émergence de la technologie numérique, le monde est devenu de plus en plus connecté et génère de plus en plus de données. De fait, la génération des données croît de 23% chaque année et d'ici 2025, l'espace de stockage total utilisé par ces données atteindra les 175 Zettabytes. Les collecter massivement et les stocker pour mieux les analyser est le rôle du Big Data. Actuellement, ces données sont stockées dans des datas centers qui consomment énormément d'énergie et prennent beaucoup de place, chaque data center pouvant stocker environ 1 Exabyte. Une des solutions pour réduire l'espace de stockage utilisé par ces données est de les compresser, en se basant sur les redondances du language par exemple. En effet, lorsqu'on écrit un texte dans un document nous produisons de nombreux motifs répétés et des algorithmes tels que LZ77 ont été crées pour réduire l'espace de stockage de ces fichiers. LZ77 parcourt en boucle le contenu d'un fichier pour identifier les séquences répétées qui peuvent être remplacées par des métadonnées ou des séquences plus courtes, réduisant ainsi le nombre total de bits nécessaires pour représenter l'information. Un point important est qu'aucune information n'est perdue. Ceci diffère des algorithmes de compression à perte comme jpeg qui peuvent rendre un fichier image plus petit mais des informations sont perdues au cours du processus. GZIP est un logiciel libre développé par Jean-Loup Gailly et Mark Adler en 1991. C'est un outil de compression de fichiers sans perte, célèbre pour réduire la taille de gros morceaux de données. Son implémentation est basée sur l'algorithme de compression DEFLATE qui est une combinaison des algorithmes LZ77 et Huffman.

Le but de ce travail est d'utiliser GZIP pour compresser des données issues des data sciences, un domaine qui génère énormément de données. Plus précisément, nous allons améliorer la compression de fichiers FASTA contenant des reads issus de séquençage, en réorganisant le fichier de façon à regrouper les reads les plus similaires entre eux. L'algorithme LZ77 utilisé par GZIP est un algorithme de compression sans perte qui va supprimer les redondances au sein d'un texte brut en utilisant une fenêtre glissante. Il va parcourir et stocker les motifs d'une séquence au fur et à mesure. Il va aussi détecter à chaque itération si le motif a déjà été rencontré. Si c'est le cas, au lieu de le stocker de nouveau, il va le remplacer par une balise qui va faire référence par chaînage à la première occurrence du motif. Huffman quant à lui classe les motifs dans une structure d'arbre composé de nœuds, où les motifs les plus fréquents se retrouve en haut de l'arbre et les moins présents en bas. Il attribue ensuite aux motifs les plus fréquents le plus petit nombre de bits et inversement, réduisant encore plus l'espace occupé. En regroupant les reads similaires, on va ainsi regrouper ceux qui partagent les mêmes motifs et ainsi faciliter la compression par LZ77. Les fichiers compressés prendront moins de place. Nous présentons donc ici 2 principaux algorithmes permettant, étant donné un fichier de reads pris en entrée, d'écrire ces mêmes reads dans un ordre différent afin d'optimiser la compression du fichier FASTA. La stratégie que nous avons décidé d'implémenter est de rassembler les séquences par motifs partagés, indépendamment des positions de ces motifs sur la séquence. En effet, du fait du mécanisme de l'algorithme LZ77 qui se base sur du référencement de motifs par chaînage, c'est ce qui nous a semblé le plus prometteur.

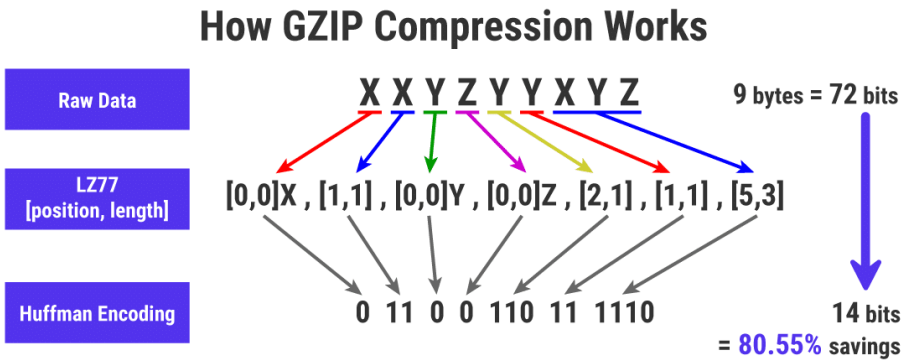


Figure 1. Une illustration approximative du fonctionnement de la compression GZIP

2 Méthodologie

Nous avons tout d'abord testé des méthodes de classement naïves en ordonnant les reads en fonction du taux de GC, par ordre alphabétique ou des proportions de A,T,C,G. Ces méthodes "naïves" ont permis de réduire respectivement l'espace de stockage de 15%, 35% et 50% sur des fichiers de profondeur 120x. Puis nos premières stratégies se sont basées sur les positions d'un ou deux k-mers ou minimizers dans les séquences, avec une réduction de la taille de compression d'environ 60-65% mais ont donné des résultats moins bons que les deux stratégies que nous allons présenter ici, notamment en temps de calcul.

Toutes les fonctions ont été testées sur un ordinateur portable, équipé de 7.7Go de RAM et d'un processeur Intel® Core™ i5-8300H CPU @ 2.30GHz × 8.

Les informations d'usage mémoire ont été mesurées grâce à $memory\_usage$ de la librairie python $memory\_profiler$, et celles de temps par $process\_time$ du module $time$. Le ratio de compression a été établi par la fonction $getsize$ de la librairie os.

2.1 Première stratégie

Le première stratégie consiste à ordonner les séquences en se basant sur le(s) x k-mer/minimiser le(s) plus fréquent(s). Concrètement, il va s'agir d'énumérer respectivement les différents k-mers et minimisers et de filtrer par ordre de présence. Algorithmiquement, on applique les étapes suivantes :

- On compte le nombre d'occurences de chaque k-mer/minimiser dans la séquence, pour chaque lecture.
- On filtre les x plus présents parmi eux.
- On concatène les x mots en une unique chaine de caractères, qui devient l'identifiant associé à la lecture.
- On trie par ordre lexicographique ces identifiants.
- L'information d'ordre de tri permet de donner l'ordonnancement des lectures en sortie.

Il va s'agir d'élaborer sur une notion de signature spécifique au read, et de comparer ces signatures dans l'espoir que regrouper les reads avec des signatures similaires va permettre une bonne compression, car on placera proches dans le fichier à compresser des reads partageant des motifs à la fois partagés entre les séquences et répétés au sein de chacune d'entre elles.

2.2 Seconde stratégie

Le seconde consiste à ordonner les séquences en se basant sur les fréquences de k-mers supérieures ou inférieures à un seuil ou sur la présence/absence de minimiseurs. L'idée première est de se baser sur les probabilités qu'un A soit suivi d'un A, d'un T, d'un C ou d'un G et ainsi de suite. Les étapes de l'algorithme appliqué aux k-mers sont les suivantes :

— On compte le nombre d'occurences de chaque k-mers dans la séquence.
— On calcule ensuite le pourcentage associés à chaque valeur.
— On réalise une simplification des résultats en ramenant les différents pourcentages à des valeurs binaires. Si le pourcentage est supérieur à un seuil alors la valeur associée au k-mer sera "1", sinon "0". On obtient ainsi une séquence ordonnée de 0 et 1 qui rend compte des proportions de chaque k-mers pour chaque séquence. Le seuil correspond au pourcentage moyen.
— On regroupe les reads en fonction de leur séquence de 0 et de 1. Cela à pour effet de regrouper entre elles les séquences très similaires.
— On tri ensuite ces groupes en triant les séquences de 0 et de 1 par ordre alphanumérique croissant. Cela à pour effet de rapprocher les reads moins similaires, mais partageant plusieurs motifs communs.

Les étapes de l'algorithme appliqué aux minimiseurs sont très similaires :

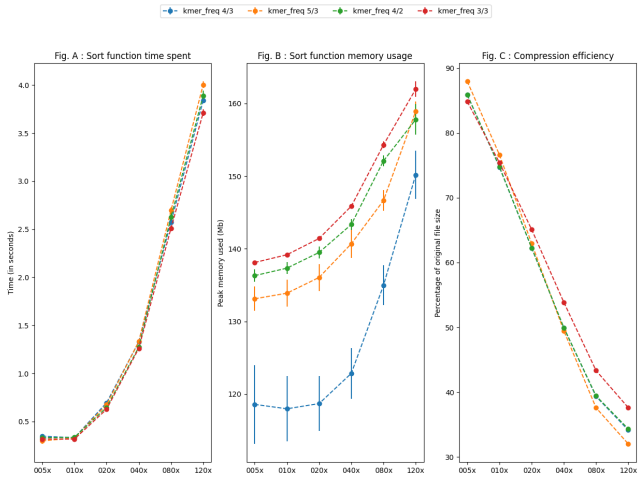
— On récupère les minimiseurs uniques d'une séquence.
— On crée une séquence de 0 et de 1 basée sur le présence (1) ou absence (0) de tous les minimiseurs possibles.
— On continue de la même facon que pour les k-mers.

3 Résultats

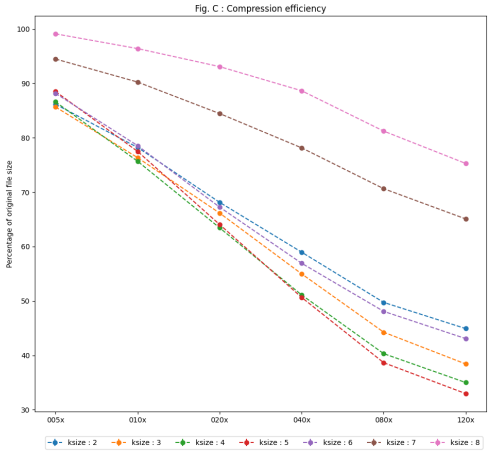
Les figures présentées dans la partie résultats contiennent 3 graphiques chacune. Ils rendent compte respectivement, de gauche à droite du temps d'exécution de la fonction de tri, du pic de mémoire maximum atteint ainsi que de son efficacité à améliorer la compression par gzip. Ce dernier graphique est obtenu en calculant le pourcentage de compression, 100% correspondant à l'espace occupé par le fichier nettoyé gzip sans tri. Un résultat autour de 50% voudrait dire que la fonction de tri permet de réduire la taille du fichier compressé par deux. Les courbes sont tracées en fonction de la taille des fichiers à compresser, du plus petit au plus grand. Chacune des courbes représente une moyenne sur 10 lancements de la fonction de tri et inclut un intervalle de confiance, représenté par les barres verticales. La première partie des tests a été réalisé sur des sorties de séquençage de génome d'*E. coli* à différentes profondeurs allant de 5x à 120x. Les algorithmes ont ensuite été testés sur le des reads du chromosome 1 humain puis sur un échantillon de métagénomique regroupant des reads beaucoup plus variés.

3.1 Première stratégie

Ces deux figures présentent les paramètres optimaux obtenus pour la compression ainsi que l'exploration partielle de diverses tailles de k-mers sur des fichiers de séquençage d'*E. coli* avec diverses profondeurs de séquençage.



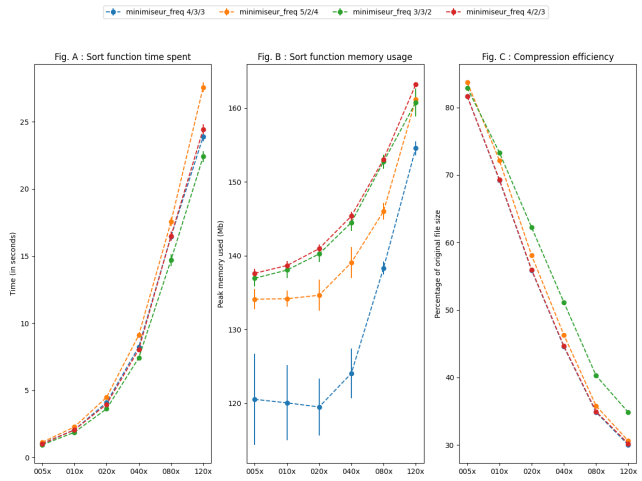
(a). Résultats globaux des optimums



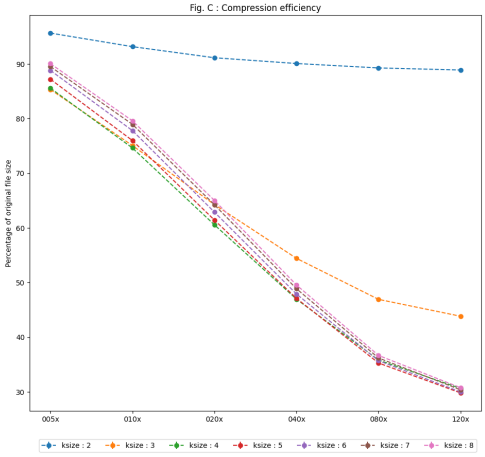
(b). Exploration de tailles de k-mers

Figure 2. Figures récapitulant l'efficacité de l'algorithme de tri basé sur la fréquence lexicographique des k-mers les plus communs sur des sorties de séquençage d'E. coli. Les valeurs indiquées dans la légende correspondent à la taille des mots/le nombre de mots.

Ces deux figures présentent les paramètres optimaux obtenus pour la compression ainsi que l'exploration partielle de diverses tailles de minimiseurs sur des fichiers de séquençage d'*E. coli* avec diverses profondeurs de séquençage.



(a). Résultats globaux des optimums

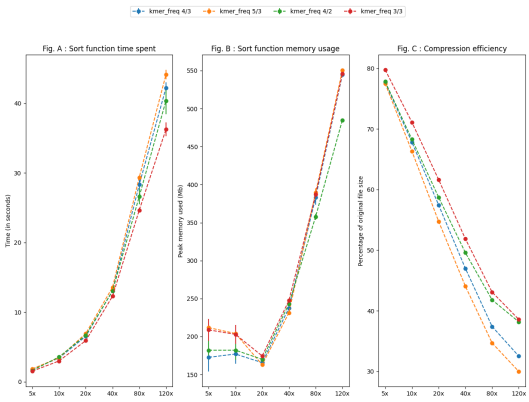


(b). Exploration de tailles de minimiseurs

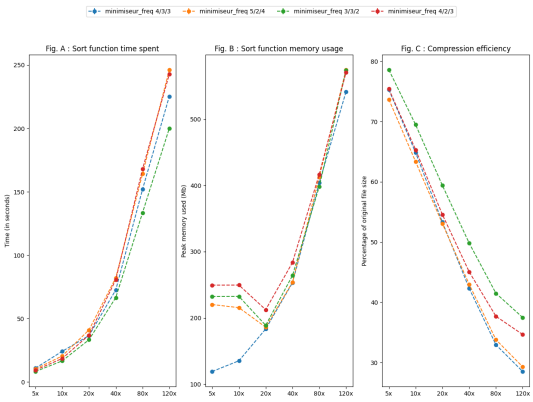
Figure 3. Figures récapitulant l'efficacité de l'algorithme de tri basé sur la fréquence lexicographique des minimiseurs les plus communs sur des sorties de séquençage d'E. coli. Les valeurs indiquées dans la légende correspondent à la taille des mots/la nombre de mots/la taille de la fenêtre.

3.1.1 Efficacité sur des données d'origine humaine

Ces figures rendent compte des résultats de compression sur des fichiers de séquençage du chromosome 1 humain à diverses profondeurs de séquençage, pour les k-mers comme pour les minimiseurs.



(a). Basé sur k-mers

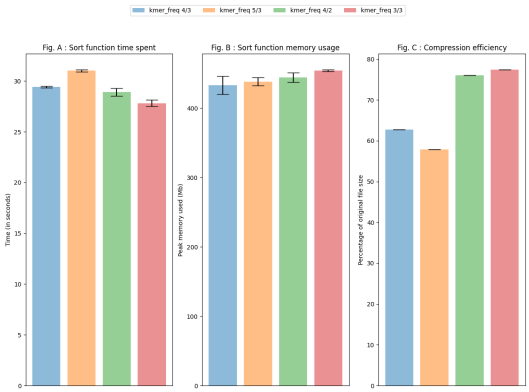


(b). Basé sur minimisers

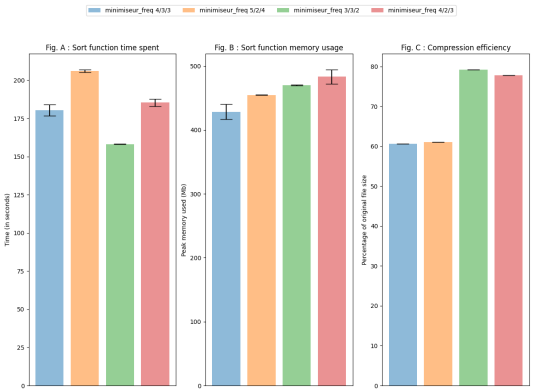
Figure 4. Figures récapitulant l'efficacité de l'algorithme de tri basé sur la fréquence lexicographique des k-mers les plus communs (gauche), et la fréquence lexicographique des minimisers (droite) sur le chromosome 1 humain. Les valeurs indiquées dans la légende correspondent à la taille des mots/le nombre de mots/la taille de la fenêtre (spécifique aux minimiseurs).

3.1.2 Efficacité sur un échantillon de métagénomique

Ces figures rendent compte des résultats de compression sur un fichier de séquençage métagénomique, pour les k-mers comme pour les minimiseurs.



(a). Basé sur k-mers



(b). Basé sur minimisers

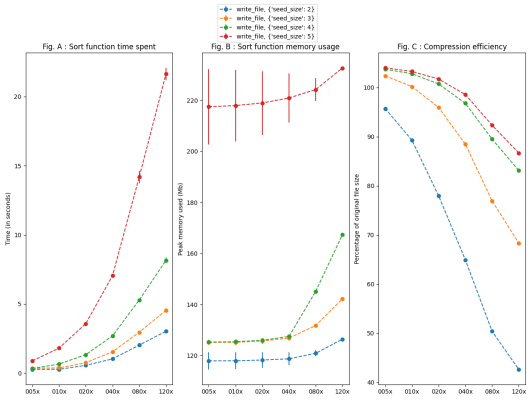
Figure 5. Figures récapitulant l'efficacité de l'algorithme de tri basé sur la fréquence lexicographique des k-mers les plus communs (gauche), et la fréquence lexicographique des minimisers (droite) sur un échantillon métagénomique. Les valeurs indiquées dans la légende correspondent à la taille des mots/le nombre de mots/la taille de la fenêtre (spécifique aux minimiseurs).

3.2 Seconde stratégie

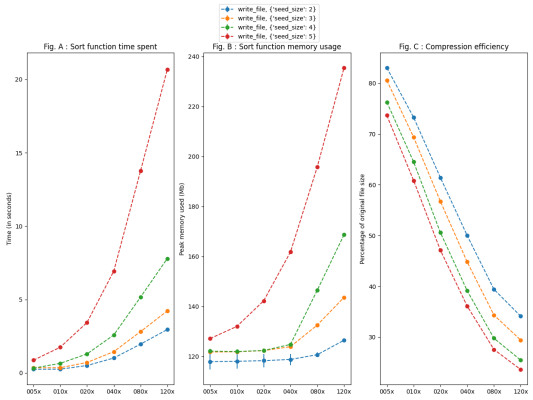
Nous allons présenter ici deux algorithmes de tri basés sur les proportions de k-mers et de minimiseurs des reads.

3.2.1 Fréquence de k-mers sur des données *E. coli*

Ces deux figures rendent compte des résultats liés à la stratégie de tri basée sur les fréquences de k-mers.



(a). Sans tri

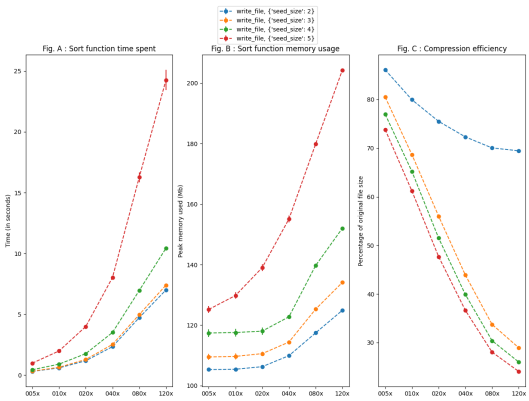


(b). Avec tri

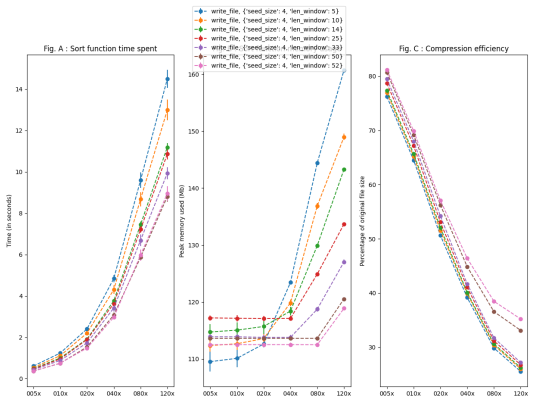
Figure 6. Figures récapitulant l'efficacité de l'algorithme de tri basé sur la fréquence des k-mers, avec et sans inclure l'étape de tri alphanumérique des séquences de 0 et de 1 associées à chaque reads, pour différentes tailles de k-mers allant de 2 à 5.

3.2.2 Présence/absence de minimiseurs sur des données *E. coli*

Ces deux figures rendent compte des résultats liés à la stratégie de tri basée sur la présence/absence de minimiseurs, en incluant à chaque fois l'étape de tri alphanumérique des séquences de 0 et de 1 associées à chaque reads.



(a). Taille de fenêtre fixe et taille de minimiseurs variable.

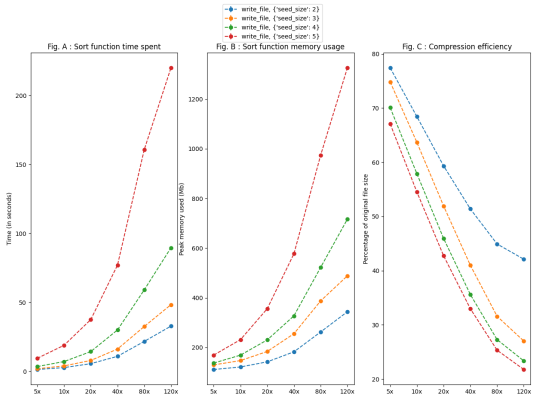


(b). Taille de minimiseur fixe et taille de fenêtre variable.

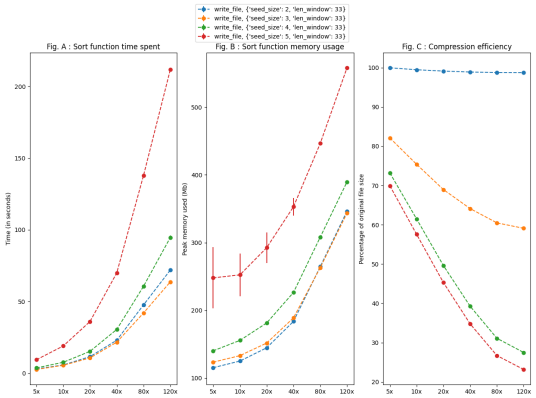
Figure 7. Figures récapitulant l'efficacité de l'algorithme de tri basé sur la présence/absence des minimiseurs, pour différentes tailles de k-mers allant de 2 à 5 et une taille de fenêtre glissante égale à 10 pour la figure a. La figure b représente une taille de fenêtre allant de 5 à 52 et une taille de minimiseur fixée à 4.

3.2.3 Efficacité sur des données d'origine humaine

Nous avons ensuite testé les algorithmes avec les paramètres optimaux identifiés précédemment sur différents échantillons provenant du séquençage du chromosome 1 humain.



(a). Basé sur la fréquence de k-mers.

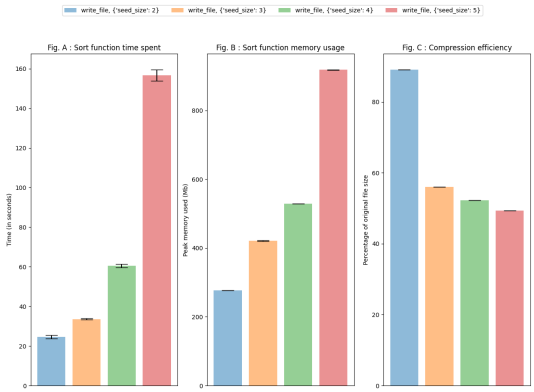


(b). Basé sur la présence/absence de minimiseurs.

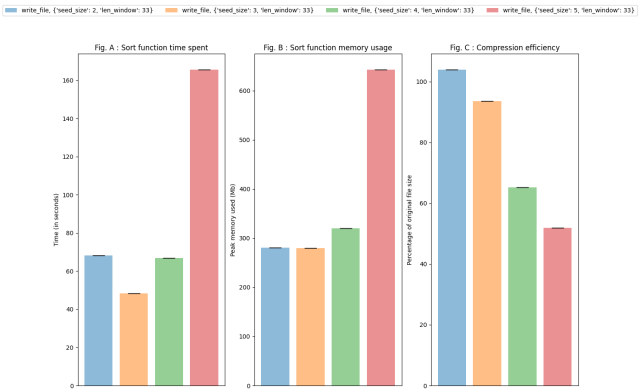
Figure 8. Figures récapitulant l'efficacité de l'algorithme de tri sur des échantillons de chromosome humain, basé sur la fréquence des k-mers et sur la présence/absence des minimiseurs, pour une taille de k-mers allant de 2 à 5 et une taille de fenêtre glissante de 33

3.2.4 Efficacité sur un échantillon de métagénomique

Nous avons ensuite testé les algorithmes avec les paramètres optimaux identifiés précédemment sur un échantillon provenant du séquençage de métagénomique.



(a). Basé sur la fréquence de k-mers.



(b). Basé sur la présence/absence de minimiseurs.

Figure 9. Figures récapitulant l'efficacité de l'algorithme de tri sur des échantillons de métagénomique, basé sur la fréquence des k-mers et sur la présence/absence des minimiseurs, pour une taille de k-mers allant de 2 à 5 et une taille de fenêtre glissante de 33

4 Discussion

4.1 Première stratégie

4.1.1 Evaluation de l'efficacité de compression sur des données d'*E. coli*

Fréquence lexicographique de k-mers

La figure 2a présente les résultats de compression des $4, 5, 4$ et 3-mers pour respectivement les $3, 3, 2$ et 3 x-plus présents. Les 4,3 et $5, 3$-mers sont les plus efficaces en termes de tri.

L'exploration de différentes tailles de k-mers a permis de montrer la propriété suivante : pour des reads de 100 bases de long, avec une forte couverture, $k = 5$ est la solution optimale. En dessous de cette valeur, on reste proche en taux de compression pour $k = 4$ et $k = 3$ dans une moindre mesure, et

on s'effondre pour $k = 2$. A l'inverse, si l'on dépasse les 5-mers, on perd également très rapidement en qualité de compression, car on ne trouve plus assez de k-mers qui soient redondants dans les séquences pour trier par k-mers les plus communs.

Tout le jeu de trouver le meilleur paramètre k consiste en le maintenir suffisament petit pour avoir des k-mers redondants au sein du read, et le maintenir suffisament grand pour avoir une notion de signature spécifique au read. Expérimentalement (fig. 2b), on peut placer le k optimal à $k = 0.05 * |read|$. Trouver le paramètre x s'est avéré rapide : à travers tous les scénarios, $x = 1$ et $x = 2$ était trop peu porteurs d'informations, tandis que l'on ne gagnait rien à calculer au-delà de 3, plaçant notre optimum à $x = 3$.

Avec cet algorithme, le tri s'effectue rapidement en comparaison des autres fonctions qui seront abordées dans ce rapport, au prix d'un taux de compression plus faible.

Cette stratégie, dans sa version basée sur les k-mers, se place en méthode rapide de tri sur des fichiers de reads à forte couverture. Elle a également l'avantage de n'avoir qu'un seul paramètre à explorer, dont la valeur optimale peut être retrouvée par calcul, et donc automatisée. En revanche, elle ne présente pas le meilleur taux de compression, ni n'excelle sur les fichiers présentant une faible couverture, ou avec peu de zones en commun.

Fréquence lexicographique de minimiseurs

Une variante de cet algorithme a été testée sur les minimiseurs (fig. 3a). Au lieu de prendre en compte les occurences de k-mers, il a été pris en compte les occurences de minimiseurs. Cette méthode présente des temps d'exécution bien plus conséquents (5.5 fois) pour un gain en compression d'environ 3%.

Diverses tailles de mots ont ainsi été explorées (fig. 3b), avec un optimum à 4, à travers tous paramètres de taille de fenêtre pour construire les minimiseurs et nombre de mots agrégés pour former les signatures.

Dans les deux approches, les données mémoire (fig. 2a et 3a) semblent peu cohérentes pour la courbe bleue (correspondant à $ksize = 4$, $k\_number = 3$ et $window\_size = 3$ dans le cas des minimiseurs) et devrait bien plus se confondre avec les autres courbes, de par la nature des structures utilisées et des autres paramètres testés, possiblement par un mauvais calcul de pic mémoire lors de la première exécution, qui se trahirait par les forts écart-type visibles sur le graphe. En faisant abstraction de ce résultat, on peut observer que sur des fichiers présentant des profondeurs de séquençage de 5x à 80x qu'augmenter le nombre de mots a plus d'impact mémoire qu'augmenter la taille de k, mais que cet effet tend à s'inverser sur des profondeurs de séquençage atteigant 120x.

4.1.2 Généralisation des tests d'efficacité à d'autres fichiers

Nous avons testé différents algorithmes sur des fichiers d'origines différentes afin de voir si leur efficacité de compression était généralisable à un vaste ensemble de fichiers provenant de différents organismes et avec des profondeurs de séquençage variables.

Sur des fichiers de chromosome 1 humain

La figure 4a présente les résultats de compression sur le jeu de données humain. On y trouve des résultats très corrélés avec les résultats sur *E. coli*, avec un taux de compression croissant selon la redondance des reads au sein du fichier.

Concernant les minimiseurs (fig. 4b) on retrouve ici aussi des résultats de compression identiques

aux données d'*E. coli* ; avec toujours une différence de 5.5 fois en temps et de +3% environ de taux de compression.

Dans les deux cas, l'usage en mémoire n'est que peu impacté par le choix des paramètres (aux alentours de 500-550 Mo pour 120x de profondeur de séquençage, à travers tous les paramètres), mais est fonction de la taille du fichier. En revanche, le temps est impacté par le choix des paramètres ; plus k est petit, plus le temps de calcul est court. La balance entre un k grand ou non se fait donc non seulement en termes de perte de capacité de compression, mais aussi de temps de calcul, poussant sur de très grands fichiers à privilégier k=4 à k=5.

Ce jeu de fichiers nous permet de valider la méthode par k-mers en tant que fonction de tri rapide sur des fichiers à forte redondance. On pourrait, sur de très gros fichiers, l'imaginer en tant que méthode de permier ordre avant d'appliquer une seconde méthode, plus locale, fonctionnant par batches de données déjà ainsi pré-triées.

Sur un échantillon de métagenomique

Les figures 5a et 5b présentent les résultats de compression sur le fichier de données métagénomique respectivement pour la méthode par k-mers et la méthode par minimiseurs. Ici, les taux de compression sont bien moindres, atteigant avec des paramètres optimaux respectivement 57% et 60% de taux de compression par rapport au fichier non trié.

Ces résultats nous permettent de conclure sur la capacité de ces méthodes à trier pour compression des fichiers avec une forte redondance avec un succès non négligeable avec respect au temps d'exécution, mais également sur son inaptitude à trier des fichiers présentant une forte disparité de données, par rapport aux méthodes que nous allons détailler dans la suite de ce rapport.

4.2 Seconde stratégie

4.2.1 Evaluation de l'efficacité de compression sur des données d'*E. coli*

Fréquence de k-mers

La figure 6a rend compte d'une comparaison d'exécution de la fonction de tri pour des tailles de k-mers allant de 2 à 5, sans inclure le tri sur l'index avant écriture du fichier trié. On remarque que plus la taille du k-mers augmente, plus la compression est mauvaise, et donc que le tri est de moins en moins efficace. Cependant les résultats obtenus avec un k-mer de taille 2 sont plutôt bons, on divise la taille du fichier compressé par presque 2,5 (40%), avec un temps d'exécution autour de 3 secondes pour le plus gros fichier et une utilisation mémoire autour de 120Mb.

Si on inclut le tri sur l'index, on remarque cette fois-ci sur la figure 6b que plus la taille du k-mers augmente, plus l'effet sur la compression est efficace, et donc plus la fonction de tri est efficace. Le fait de trier en fonction de la valeur de hash à donc un fort impact sur la compression du fichier. En effet, avec une taille de k-mer de 5, on divise la taille du fichier compressé d'origine par presque 5 (20%). Cependant c'est aussi celle qui prend le plus de temps - autour de 20 secondes pour le fichier le plus gros – et le plus de mémoire – autour de 235Mb. Le compromis idéal qui semble se dégager sur le graphique est avec une taille de k-mers de 4, qui compresse avec une efficacité similaire mais a un temps d'exécution de seulement 7 secondes et un pic de mémoire autour de 170Mb.

Minimiseurs et comparaison avec la fréquence de k-mers

Si on regarde la figure 7a, on observe que les résultats de compression sont similaires à ceux obtenus pour les k-mers, avec une importante différence d'efficacité dans le cas de l'utilisation de minimiseurs de taille 2. On remarque cependant que le temps d'exécution est plus long, autour de 25 secondes et que l'occupation de mémoire est globalement inférieure et augmente moins vite. En effet l'algorithme est un peu plus complexe ce qui peut prendre plus de temps, mais on ne stocke pas tous les k-mers présents, uniquement un set plus réduit de minimiseurs, ce qui prend moins de place.

La figure 7b rend compte d'une comparaison d'exécution de la fonction de tri pour des tailles de fenêtre allant de 5 à 52. Ces valeurs ont été choisies afin de simuler l'utilisation d'un nombre de minimiseurs maximal de 20, 10, 7, 4, 3, 2 et 1 respectivement. En effet, plus on augmente la taille de la fenêtre, plus on réduit le nombre de minimiseurs potentiels que l'on peut extraire de la séquence. On remarque que les résultats de compression sont assez similaires pour les différentes tailles de fenêtres, avec un baisse significative uniquement pour les fenêtres de taille 50 et 52, correspondant à un nombre de minimiseurs maximum de 2 et 1 respectivement. En termes de temps de calculs, les plus grandes fenêtres sont plus intéressantes que les petites, car on parcourt la séquence en moins d'étapes. Il est intéressant de noter que pour la mémoire, il semble plus intéressant de prendre une petite fenêtre pour les petits fichiers et une grande pour les gros fichiers. L'inversion de tendance semble se faire à partir des fichiers de profondeur 20x. Cependant la variation de mémoire utilisé pour les petits fichiers n'est que de 8Mb entre les différentes tailles de fenêtres, ce qui ne représente pas grand chose en termes de variation de mémoire. Par contre, elle passe de 112Mb pour une fenêtre de 52 à 160Mb pour une fenêtre de 5. Une fenêtre taille 33, en violet, semble être la solution optimale pour bien classer les reads, le fichier trié avec ces paramètres est environ 4 fois plus petit que l'original, avec un bon compromis temps d'exécution et mémoire.

4.2.2 Généralisation des tests d'efficacité à d'autres fichiers

Nous avons testé différents algorithmes sur des fichiers d'origines différentes afin de voir si leur efficacité de compression était généralisable à un vaste ensemble de fichiers provenant de différents organismes et avec des profondeurs de séquençage variables.

Sur des fichiers de chromosome 1 humain

Les figures 8a et 8b ont été obtenues à partir de fichiers plus volumineux de séquençage sur le génome humain à différentes profondeurs. On remarque que les résultats de compression sont très similaires, avec un temps de calcul et d'utilisation de la mémoire plus important dû à la taille globale des fichiers qui a augmentée. L'optimum reste d'utiliser une taille de k-mer et de minimiseur de 4, la courbe verte semblant être encore une fois le meilleur compromis. On pourrait également envisager d'utiliser une taille de 3 en ce qui concerne les k-mers pour les très gros fichiers, cela divise le temps d'exécution par 2 ici, avec un gain de presque 200Mb en mémoire, pour une perte de compression de moins de 5%.

Sur un échantillon de métagenomique

Les figures 5a et 5b ont été obtenues à partir d'un fichier de séquençage issu d'une analyse de métagénomique. La technique qui semble la plus efficace en terme de compression est celle utilisant les minimiseurs, avec les mêmes paramètres optimaux identifiés précédemment. On compresse ici au maxi-

mum à 50%, réduisant la taille du fichier par 2, ce qui est moins bon que les résultats précédents et peut s'expliquer par une hétérogénéité plus importante au sein des reads, qui proviennent d'espèces différentes.

5 Conclusion

Les deux méthodes présentées à travers cet article présentent chacune leurs points forts et faibles. L'approche par présence/absence de minimisuers se propose comme solution efficace pour le tri de fichiers présentant une grande disparité d'informations, là où l'approche par nombre d'occurences de k-mers se place comme alternative compétitive pour des fichiers présentant une forte redondance, de par son bien plus faible temps d'exécution, au prix d'une faible perte en qualité de compression.

Si les résultats présentés permettent effectivement de compresser de manière significative des fichiers de reads au format fasta avec perte d'informations de séquençage, d'autres méthodes auraient pu être explorées, ainsi que d'autres stratégies de validation. On pourra notamment mentionner que la référence utilisée en cet article pour les taux de compression consiste en un fichier brut nettoyé de ses headers, compressé par GZIP, mais que le caractère "brut" du fichier n'est que peu représentatif ; il aurait pu être judicieux que de trier ce fichier lexicographiquement afin de pouvoir donner un protocole de référence commun plutôt qu'obtenir une approximation stochastique d'un taux de compression.

Une stratégie alternative pensée aurait été d'utiliser un modèle de *machine learning* sur les données, pour dégager des patterns d'organisation de reads au sein d'un fichier. Concrètement, il s'agirait d'ordonner aléatoirement les reads, de mesurer un ensemble de métriques (GC%, occurences de k-mers, présence/absence de minimiseurs...) et de donner une note (qui serait donc une classe, discrétisée depuis le pourcentage d'espace gagné sur le disque) au taux de compression obtenu ainsi aléatoirement. On ferait apprendre au modèle à reconnaître comment une organisation donnerait un taux de compression, puis on remonterait le modèle pour en extraire les jeux de métriques permettant une bonne compression, que l'on appliquerait alors (à la manière d'une *rule-based approach*) à un jeu brut de données pour le compresser de la meilleure manière. Cela donnerait lieu à une structure de tri d'ensemble, où un ensemble de trieurs faibles seraient agrégés selon un certain ratio dans la volonté d'obtenir un trieur fort.