Rapport développeur

Structure générale du code

Le programme se découpe en trois fichiers distincts. Le script `read_organizer.py` contient le parser en ligne de commande nécessaire pour lancer la compression d'un fichier. Après vérification des arguments fournis par l'utilisateur, il exécute une fonction contenue dans `sort_functions.py`. Grâce à l'import de `sort_functions.py` dans `read_organizer.py`, la fonction `getattr(sort_functions, args.func)(...)` permet de lancer, depuis le nom de fonction donné en argument, la fonction correspondante dans le fichier contenant les fonctions de tri.

Le script `tests.py` permet d'analyser les comportements en temps, en mémoire et en compression des fonctions implémentées dans `sort_functions.py`. Il ne sera pas détaillé en profondeur dans ce rapport ; en quelques mots, son fonctionnement repose sur le passage d'une liste de tuples `(func, {**kwargs})` qui seront les fonctions avec leurs paramètres à tester, et le code itère *n* fois à travers tous les fichiers fournis en test pour fournir des figures présentant la moyenne de ces itérations avec respect à l'écart-type en barre d'erreur.

Fonctions partagées

Comptage d'occurences dans des chaînes

Les fonctions `frequency_kmer` et `frequency_minimizer` consistent à retourner, pour le read donné en argument, le dictionnaire contenant respectivement les kmers de ksize de long, et les minimiseurs de ksize de long d'une fenêtre glissante sur le read de len_window de long. Pour chaque position, on récupère un kmer/minimiseur, qui se trouve inclus au dictionnaire comptabilisant ceux-ci. L'objet retourné est un dictionnaire, au format `kmer/minimiseur : nombre d'occurences du kmer/minimiseur` pour chaque kmer/minimiseur rencontré dans le read.

Nettoyage des fichiers de séquençage

La fonction `clean_fasta` prend en entrée un chemin vers un fichier FASTA-like valide, et se charge de le retourner en une liste. Attention, pour de très grands fichiers ne rentrant pas dans la mémoire, une telle méthode ne serait pas appropriée, et il faudrait renvoyer le contenu du fichier en tant que générateur. La fonction nettoire toutes les lignes précédées de chevrons (headers) et nettoie les éventuels espaces et retours à la ligne.

Gestion des entrées/sorties

Le traitement des entrées/sorties s'effectue via une fonction `write_output(func)` utilisée en tant que décorateur. On ajoute aux paramètres de `func` la liste des reads contenus dans le fichier (appel à `clean_fasta`). L'écriture dans le fichier de sortie est appelé sur le retour de la fonction `func` décorée. Cela permet d'implémenter et de maintenir facilement les vérifications sur les fichiers d'entrée et de sortie, sans impacter la capacité à mesurer la mémoire et le temps requis pour l'exécution de l'algorithme de tri.

Cela permet d'implémenter toute autre nouvelle méthode de tri sans avoir à se questionner sur les entrées/sorties, tant que la fonction nouvellement implémentée prend en paramètres un fichier d'entrée, un

de sortie, et un dictionnaire optionnel de reads. Toute nouvelle méthode doit suivre cette signature minimale :

```
@write_output
def some_sort_function(input: str, output: str, reads: list = []) -> list:
```

Afin d'ajouter une nouvelle fonction de tri, après implémentation, il faut la décorer et donner son nom à `PARSER_FUNCTIONS` pour qu'il la propose dans la liste de ses choix.

Fonctions de tri

Algorithme d'occurences des kmers : kmers_lexico

La fonction principale `kmers_lexico` repose sur une compréhension de liste qui, pour chaque read, extrait une signature formée des *kmer_number*, *ksize*-mers les plus communs.

```
[reads[i] for i in [i for i, _ in sorted(enumerate(['',join([key for key, _ in
frequency_kmer(read, ksize).most_common(kmer_number)]) for read in reads]),
key=lambda x:x[1])]]
```

On récupère ainsi le nombre souhaité de kmers les plus présents ainsi que leur nombre d'occurences dans la séquence avec `frequency_kmer(read, ksize).most_common(kmer_number)`. Puis, on filtre afin de ne conserver que les kmers par ordre de présence avec la compréhension `[key for key, _ in sorted_counter]`. Ensuite, on concatène grâce à la fonction `join` les *n* kmers en une signature, que l'on trie lexicographiquement selon cette signature en fonction de l'index de la signature avec `sorted(list,key=lambda x:x[1])` qui se trouve à la position 1 de chaque tuple. On récupère une liste de positions, et on construit la liste en récupérant chaque read à la position `i` pour chaque position dans la liste de positions, ce qui réordonne nos reads pour la sortie.

Algorithme d'occurences des minimisers : minimisers_lexico

La seule différence dans la fonction principale `minimisers_lexico` par rapport à `kmers_lexico` est dans l'appel à la fonction permettant d'obtenir le comptage des minimiseurs au lieu des kmers. On fait ici appel à la fonction globale `frequency_minimizer` et non `frequency_kmer` dans l'appel `frequency_minimizer(read, ksize, len_window).most_common(kmer_number)`

Algorithme de fréquence des kmers : kmers_frequency

La fonction principale `kmers_frequency` récupère un dictionnaire indexant les reads à partir des fonctions annexes, l'index contient la métrique associé aux reads. Cette métrique est une séquence de 0 et de 1 rendant compte de façon simplifiée des proportions en kmers de la séquence. Les reads sont stockés via leur index dans la liste `reads` qui contient tous les reads du fichier, non triés. Cela permet d'économiser en mémoire. La fonction retourne ensuite une liste, contenant les reads triés par ordre alphanumérique. Elle est calculée de la façon suivante:

```
list(chain(*[[reads[int(seq)] for seq in index[key]] for key in
sorted(index.keys())]))
```

La boucle `for key in sorted(index.keys()` trie les clés dans l'ordre alphanumérique. La boucle `[[reads[int(seq)]for seq in index[key]]` récupère les valeurs associées à chaque clé (leur position dans la liste reads) et renvoie les reads associés. On obtient à cette étape une liste contenant des listes. La méthode `chain*` permet d'applatir la liste en itérant avec `*` sur toutes les listes que contient la liste. On pourrait choisir de trier les reads au moment de les écrire dans le fichier pour éviter de les stocker dans une liste, mais le choix a été fait d'avoir une seule fonction commune pour écrire le fichier trié et de faire moins d'action d'écriture dans le fichier d'output.

La fonction `indexation` génère le dictionnaire indexant les reads. Elle commence par récupérer un dictionnaire pour chaque read, contenant les kmers présents dans le read ainsi que leur nombre d'occurences. Elle simplifie ensuite le nombre d'occurences, en le passant à 1 si il est supérieur à un seuil, ou à 0 sinon et récupérant le résultat sous la forme d'une séquence de 0 et de 1. Elle ajoute ensuite cette séquence dans le dictionnaire si elle n'existe pas encore et ajoute la position associée au read dans la liste des valeurs du dictionnaire. On génère au début de la fonction `list_xmers` qui contient toutes les combinaisons de kmers possibles, de facon à s'en servir de référence pour que les positions de 0 et de 1 dans la séquence générée correspondent aux mêmes kmers pour tous les reads.

La fonction `binary` prend en entrée le dictionnaire contenant les occurences de kmers et permet de simplifier ce nombre d'occurences, en les passant à 1 si ils sont supérieurs à un seuil, ou à 0 sinon. Elle renvoie le résultat sous la forme d'une séquence de 0 et de 1. On considère ici que si le nombre d'occurences du kmer est supérieur au nombre d'occurences si les kmers étaient répartis da façon uniforme dans la séquence, alors ce kmer est surreprésenté et il sera associé à la valeur 1. Le calcul du seuil représente cette répartition uniforme théorique.

Algorithme de présence/absence des kmers : minimiser_presence_absence

La fonction principale `minimiser_presence_absence` fonctionne de la même façon que la fonction principale `kmers_frequency`. Le différence réside dans le fait que cette fois-ci on récupère un nombre d'occurences de minimiseurs plutôt que de kmers et qu'on simplifie le problème en se basant sur la présence/absence de ces minimiseurs dans la séquence plutôt que leur fréquence d'apparition.

La fonction `indexation_minimisers` fonctionne sur le même principe que la fonction annexe `indexation`, mais s'applique sur les minimiseurs et non les kmers.

La fonction `binary_minimisers` fonctionne sur le même principe que la fonction annexe `binary`, à la différence qu'on ne calcule pas de seuil, si le minimiseur est présent dans le read on ajoute un 1 à la séquence, sinon un 0.

La fonction `frequency_minimizer` fonctionne sur le même principe que la fonction annexe `frequency_kmer`, en incluant une fenêtre glissante qui parcourt la séquence et dont on peut ajuster la taille. Le minimiseur est récupéré en listant tous les kmers présents dans la fenêtre et en récupérant le plus petit (ordre lexicographique). On récupère aussi sa position j dans la séquence. Pour améliorer la vitesse de parcours de la séquence par la fenêtre glissante, le terme `i += minimiser[1] + 1` est utlilisé. Dés que l'on a

trouvé un minimiseur, on déplace la fenêtre de façon à dépasser ce minimiseur avant de recommencer à en chercher un. On utilise pour cela la position `j` du minimiseur dans la séquence stocké dans `minimiseur[1]`.