Université de Lille

Thèse

pour obtenir le grade de :

Docteur de l'Université de Lille

dans la spécialité « Informatique et applications »

par **Léa Vandamme**

Structures de données efficaces pour l'indexation des séquences de troisième génération

Thèse soutenue le 18 décembre 2025 devant le jury composé de :

M.	Antoine Limasset	Chargé de recherche, Laboratoire CRIStAL, CNRS, Université de Lille	(Directeur de Thèse)
M.	Bastien Cazaux	Maître de Conférences, Laboratoire CRIStAL, Université de Lille	(Co-Encadrant de Thèse)
M.	Dominique Lavenier	Directeur de Recherche CNRS, IRISA, Rennes	(Rapporteur)
M	Daniel Gautheret	Professeur, I2BC, Université Paris-Saclay	(Rapporteur)
Mme	Thérèse Commes	Professeure, IRMB, Université de Montpellier	(Examinateur)
Mme.	Marie-Eléonore Kessaci	Professeure, Laboratoire CRIStAL, Université de Lille	(Examinateur)

Laboratoire CRIStAL
Bâtiment ESPRIT, Avenue Henri Poincaré, 59655 Villeneuve d'Ascq
France

Remerciements

Je tiens tout d'abord à remercier les membres du jury. En premier lieu Dominique Lavenier et Daniel Gautheret qui ont accepté le rôle de rapporteur et ont donc participé à l'amélioration de ce manuscrit de part leurs retours. Merci également à Thérèse Commes et Marie-Éléonore Kessaci pour l'intérêt porté à mes travaux et à leur présence lors de la soutenance.

Les plus gros remerciements ici reviendront à mes deux encadrants de thèse : Antoine Limasset et Bastien Cazaux. Merci pour tous vos conseils, votre aide quotidienne, d'avoir partagé mes moments de désespoir pendant mes mois de débuggage, et ça toujours dans la bonne humeur et de manière positive. Vous avez réalisé l'exploit de m'avoir supportée (dans tous les sens du terme) pendant quatre années depuis mon stage de fin de Master, je pense qu'une légion d'honneur s'impose.

Je souhaite évidemment remercier l'ensemble de l'équipe BONSAI pour tout ce temps passé ensemble. Il est compliqué de citer tout le monde, mais merci à Hélène de m'avoir accompagnée lors de mon premier stage dans l'équipe, à Jean-Stéphane de m'avoir aidée dans les démarches pour être doctorant assistant, à Laurent pour m'avoir grandement aidée à la préparation de mes TD. Un grand merci également à Camille, Yoann SANS H, Areski pour leur bienveillance, à Mickael pour sa patience face aux questions administratives. Merci à Antoine pour la période Mahjong, je me souviendrai à jamais de ce Riichi Ippatsu Chiitoitsu Chinitsu Tanyao Dora / Sanbaiman dealer qui t'auras ruiné ton score, milles excuses.

Un immense merci également à tous les Précaritos. En premier lieu Coralie qui m'a accueillie dans son bureau à mon arrivée (et m'a donné un bon avant goût de ce qu'est la fin de thèse), à Thomas d'être venu m'embêter en chaise à roulettes dès qu'il en avait l'occasion (quand il perdait sa partie d'échecs en gros), à Lilian d'avoir fait sonner toutes les portes du couloir au moins une fois, au fameux duo Timothé/Caleb pour la bonne ambiance apportée. Merci également à Igor pour la splendide affiche de thèse, à Mathilde d'avoir contribué à cette thèse grâce à OReO, à Yohan AVEC UN H, Karl, Florian, Etienne, Pierre, Valentin et Bastien D pour toutes les conversations que nous avons pu avoir, et pour certains d'avoir partagé mon bureau (dans le noir et très chauffé, désolée).

Cette période de thèse n'aurait jamais pu être aussi supportable sans cette équipe, MERCI. Je souhaite beaucoup de courage et de réussite aux futurs docteurs (faut bientôt commencer à écrire pour certains.... :)), vous allez gérer j'en suis sûre et je serais ravie de pouvoir être présente à votre soutenance.

Je tiens également à remercier toutes les personnes qui d'un peu plus loin m'ont soutenue dans cette aventure. En premier lieu ma famille qui est toujours présente, mes amis qui m'ont entendue paniquer pendant 3 ans, notamment les joueurs/coéquipiers/coachs/parents Roncquois/Béthunois/Lillois au ping qui auront subi mon stress en compèt comme à l'entraînement.

Parce que mon régime alimentaire plus que doûteux le nécessite, je remercie le CROUS pour les pâtes carbo et bolo de qualité, l'Atelier pour les supers sandwichs steack haché/cheddar/harissa que vous m'avez servis sans jamais me juger, ainsi qu'au distributeur du rez-de-chaussée que j'ai régulièrement vidé.

Merci à Bonsai Will Survive (BWS pour les intimes) d'avoir supporté l'ensemble de mes benchs.

Je ne peux terminer ces remerciements sans une pensée pour les chats dont j'ai pu m'occuper ou dont j'ai vu les photos passer sur la SPA Bonsai : Bélial, Rayna, Spatule, Chocolat, Spike, Chatouille, Rosalie et le meilleur d'entre tous Miekki (tu es le plus beau et le plus intelligent, n'écoute pas les rageux).

Pour terminer merci Nadine, je ne connais toujours pas ton histoire, mais ton nom est apparu partout en variables, prints de débuggage, noms de fichier, exemples, tu as donc fait en quelques sortes partie de mon quotidien.

Résumé

L'émergence de la troisième génération de séquençage (TGS), technologie produisant des longs reads, a transformé les approches d'analyse des données génomiques. Bien que ces longs reads permettent de surmonter certaines limites associées aux reads courts, notamment la résolution des régions répétées, leur assemblage et leur traitement posent encore aujourd'hui de nombreux défis. L'analyse de novo sans recours à un génome de référence s'impose dans certains contextes comme une stratégie particulièrement pertinente dans de nombreux cas, par exemple lorsque aucun génome de référence n'est disponible, dans le cadre de la transcriptomique ou encore dans le cas d'études métagénomiques où les données proviennent de multiples organismes souvent inconnus à l'image du projet *Tara Oceans* récoltant des échantillons planctoniques variés. Pour que ces analyses puissent passer à l'échelle et être efficaces, notamment face au volume croissant des données mais aussi adaptées aux spécificités des longs reads, il est indispensable de s'appuyer sur des structures d'indexation efficaces et adéquates.

Cette thèse s'inscrit dans ce contexte, avec pour objectif principal le développement de solutions de recherche d'informations qui permettront l'analyse de novo adaptées aux données issues du séquençage de troisième génération. Après une étude de l'état de l'art et de l'identification de leurs limites, nous avons proposé de nouvelles méthodes d'indexation de longs reads, pour permettre une exploitation efficace de ces séquences : détection de variations, quantification, génotypage ou encore comparaison entre jeux de données.

L'apport central de cette thèse est la mise en place de stratégies, permettant l'association de *k*-mers aux reads auxquels ils appartiennent, capables de passer à l'échelle. Une première solution mise en place et utilisable est notre implémentation K2R (*k*-mer to Reads), capable d'indexer des séquençages de grande taille (plus de 100X de génome humain). Ce nouvel outil repose sur une stratégie d'indexation par minimizers, optimisée pour la performance et paramétrable afin de s'adapter à de nombreux cas d'utilisation. Nous avons par la suite étudié comment améliorer l'impact mémoire de K2R, grâce à une méthode de réordonnancement de reads et observé que ce réordonnancement permettait également d'optimiser la compression de séquençages longs reads.

Dans un second temps, nous avons développé des outils complémentaires à K2R. Nous avons proposé un outil permettant la recherche en streaming de *k*-mers : K2Rmini. Il adopte une approche inverse à celle de K2R en indexant les requêtes plutôt que le jeu de données. Nous avons également exploré un autre type d'index, complémentaire à K2R : ONIKA. ONIKA repose sur une représentation des séquences sous forme de sketchs, qui sont des sous-ensembles de *k*-mers, permettant une empreinte mémoire réduite. À l'image de K2R, chaque élément est associé aux jeux de données dans lesquels il apparaît, ce qui permet d'effectuer des requêtes rapides et efficaces, y compris lors de la comparaison de grands jeux de données.

Table des matières

Liste des figures

Liste des tableaux

Chapitre 1

Contexte

La bioinformatique est un domaine interdisciplinaire à la croisée de la biologie, de l'informatique et des mathématiques. En particulier, la bioinformatique des séquences vise à développer des outils, méthodes et algorithmes pour analyser et interpréter les données biologiques, en particulier celles issues des technologies de séquençage. Avec l'essor rapide des technologies de séquençage au cours des deux dernières décennies, la quantité de données génomiques et transcriptomiques produites a connu une croissance exponentielle, rendant la bioinformatique incontournable dans de nombreux domaines : médecine personnalisée, recherche sur les maladies génétiques, études évolutives, agriculture ou encore écologie.

Les progrès du séquençage ont permis de générer une quantité massive d'informations génétiques, rendant possible une étude plus poussée des génomes et transcriptomes. À titre d'exemple, le nombre de nucléotides séquencés contenus dans l'ENA (European Nucleotide Archive) double approximativement tous les 45 mois et comprend actuellement plus de 108 peta paires de bases (Pbp) de données de séquençage brutes, dont 67 Pbp sont accessibles au public (61). Bien que constituant une véritable mine d'or pour la recherche, ces quantités de données nécessitent d'être analysées par la suite. Leur étude représente un défi majeur en termes de stockage, de traitement et de capacité d'analyse.

Dans ce contexte, la bioinformatique des séquences vise à développer des structures de données et algorithmes toujours plus efficaces, pouvant manipuler ces volumes considérables d'informations tout en étant capables de s'adapter aux contraintes spécifiques des données biologiques : erreurs de séquençage, redondance, répétitions...

Dans ce contexte, nous commencerons par introduire les données de séquençage en soulignant l'importance de leur analyse. Nous aborderons ensuite les différentes approches permettant leur étude, en mettant en perspective les avantages et les limites des méthodes utilisant ou non un génome de référence (séquence ADN "modèle", construite à partir d'un ou plusieurs individus, utilisée comme base commune pour comparer, aligner et analyser d'autres séquences génomiques). Enfin, cette réflexion nous amènera naturellement à étudier les solutions permettant d'indexer ces données, qui constituent une solution centrale pour rendre ces analyses à la fois efficaces en temps et en mémoire.

1.1 Introduction

1.1.1 ADN et étude des séquences génomiques

L'acide désoxyribonucléique (ADN) constitue la molécule essentielle au stockage et à la transmission de l'information génétique chez tous les organismes vivants. Chez les eucaryotes, il est confiné dans le noyau, tandis que chez les procaryotes, il se situe dans une région spécifique appelée nucléoïde. L'ADN adopte une structure complexe sous forme d'une longue chaîne de nucléotides, composée de quatre bases azotées : l'adénine (A), la thymine (T), la cytosine (C) et la guanine (G). Ces bases établissent des liaisons complémentaires (A avec T et C avec G), conférant ainsi à l'ADN sa structure caractéristique en double hélice (Voir Figure 1.1).

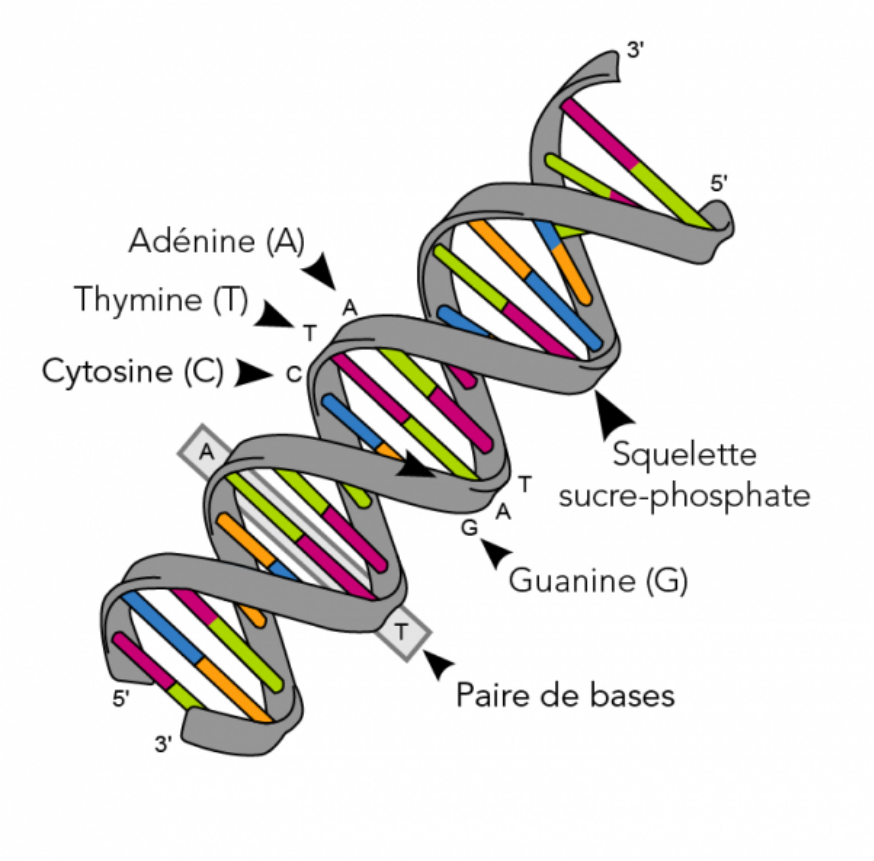


FIGURE 1.1 – *Représentation schématisée de l'ADN.*

L'étude des séquences génomiques (issues du génome, ensemble des instructions d'ADN présentes dans un organisme) est devenue un pilier fondamental dans de nombreux domaines, notamment en médecine, où elle permet le développement de thérapies ciblées et le diagnostic des maladies génétiques, en phylogénie pour la reconstruction des arbres évolutifs et la compréhension des liens entre espèces. Parmi de nombreuses autres applications nous pouvons également citer l'agronomie, pour l'amélioration des cultures et la résistance aux maladies, et l'environnement, pour la surveillance et la conservation des espèces menacées ainsi que l'évaluation de l'impact des changements climatiques sur la biodiversité.

Au cœur de ces analyses se trouve la détection des variants, un processus clé permettant d'identifier les différences au sein des génomes. Une seule variation, selon sa nature et sa position, peut avoir des répercussions majeures sur le fonctionnement d'un organisme, influençant des traits biologiques spécifiques ou entraînant des pathologies (18).

Ce que nous appelons variations ici, sont en réalité des **mutations génétiques**. Trois types de mutations existent, et vont modifier la séquence de nucléotides :

— L'insertion : ajout d'un nucléotide supplémentaire dans la séquence
— La délétion : suppression d'un nucléotide de la séquence
— La substitution : modification d'un nucléotide (exemple : $A \rightarrow T$, ou $C \rightarrow G$)

Lorsqu'une mutation affecte un seul nucléotide, on parle de SNP (Single Nucleotide Polymorphism). L'impact de cette mutation sur la chaîne d'acides aminés, et éventuellement sur la protéine résultante, dépend de sa position au sein du gène concerné. Les protéines jouent un rôle central dans les processus cellulaires, en ayant par exemple des rôles structurels, enzymatiques ou encore de régulation. Des exemples de mutations sont illustrés dans la Table 1.1.

Les acides aminés sont encodés par des triplets de nucléotides appelés codons. Comme plusieurs codons peuvent coder pour un même acide aminé, certaines mutations n'altèrent pas la séquence protéique ; on parle alors de mutation silencieuse. En revanche, si la mutation entraîne un changement d'acide aminé, on parle de mutation non synonyme ou missense. Enfin, dans certains cas, la mutation peut transformer un codon codant en un codon STOP, entraînant une terminaison prématurée de la synthèse protéique et conduisant à une protéine tronquée, souvent non fonctionnelle.

Séquence nucléotidique	CGC	CAC	UGC
Séquence protéique associée	Arg	Pro	Cys
Séquence nucléotidique modifiée	CGA	CAA	UGA
Séquence protéique modifiée	Arg (mutation silencieuse)	Gln (missense)	STOP (nonsense)

TABLE 1.1 – *Exemple de mutations en région codante (substitution d'un A en dernière position dans chaque codon). Dans le premier cas, la protéine reste la même, la mutation est donc silencieuse. Pour le second cas, CAC code pour une proline (Pro) mais CAA code pour une glutamine (Gln), on a donc un changement d'acide aminé. Dans le dernier cas, UGC code pour une cystéine (Cys), tandis qu'UGA est un codon STOP.*

Grâce à l'analyse des séquences génomiques, de nombreux cas concrets de SNPs entraînants de graves conséquences sur la santé sont désormais connus. Par exemple, nous savons que des SNPs dans le gène APOE (E4) auront une influence sur le risque de développer la maladie d'Alzheimer (4).

Lorsque la mutation s'étend au-delà d'un seul nucléotide, généralement 50 paires de bases, plusieurs types de variations peuvent être observées. Parmi eux, on distingue les variants structuraux (voir Figure 1.2), qui incluent : les duplications (copie supplémentaire d'un fragment d'ADN), inversions (inversion d'un segment d'ADN) et translocations (changement de position d'un segment). En parallèle, certaines mutations affectent le nombre de copies d'un gène. Contrairement à une duplication où une région du génome est copiée et insérée à proximité de son origine, cette augmentation du nombre de copies d'un gène désigne le nombre total de copies présentes dans le génome, qui peut résulter de duplications locales ou d'autres réarrangements. Elle peut conduire à une surexpression du gène, tandis qu'une diminution, voire une perte complète, peut altérer ou abolir sa fonction.

La détection de ces mutations à plus grande échelle permet notamment d'expliquer des différences inter-espèces. Ces mutations peuvent entraîner des variations phénotypiques significa-

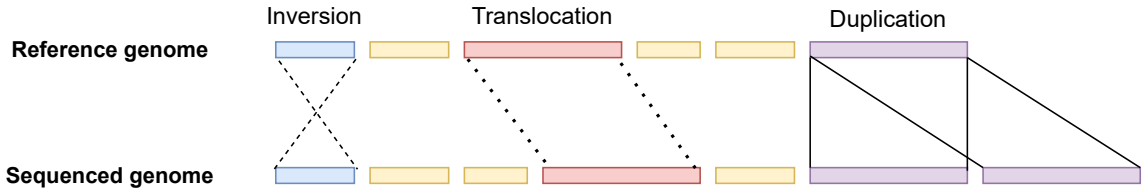


FIGURE 1.2 – *Illustration des différents variants structuraux.*

tives, influençant des traits tels que la morphologie, le comportement ou les adaptations physiologiques. Par exemple, nous pouvons citer le gène FOXP2 ayant subi deux mutations chez l'humain par rapport au chimpanzé, qui est associé au développement du langage et de la parole (40).

La détection de ces variants repose sur une comparaison systématique entre les génomes de deux individus. C'est par exemple en confrontant le génome d'un individu atteint d'une pathologie à celui d'un individu sain que l'on peut identifier les variations responsables de la maladie. Cette approche comparative, recherchant les éléments communs et différents grâce à un procédé d'alignement, permet de mettre en évidence les mutations spécifiques associées à certaines conditions génétiques. Dans la Section 1.1.2, nous expliquerons plus en détails les notions d'alignement et de mapping de séquences. Nous commencerons par présenter les algorithmes d'alignement exact, avant d'exposer les limites de ces approches en termes de performances, qui nécessitent le recours à des méthodes heuristiques.

1.1.2 Analyse par référence

L'opération fondamentale qui permet ces analyses est **l'alignement** entre les séquences à étudier et une référence, processus clé permettant d'identifier les régions similaires ainsi que les différences entre les séquences, facilitant ainsi l'identification des variations éventuelles. L'alignement constitue donc une étape incontournable pour la plupart des applications en bioinformatique et en génomique, et est un sujet d'optimisation continue en raison de la taille des génomes étudiés, mais aussi de la quantité croissante du nombre de jeux de données disponibles.

Alignement exact

L'alignement consiste à calculer une distance entre deux séquences. Il peut s'agir de la distance de Hamming, qui dénombre les positions où deux séquences de même longueur diffèrent, ou de la distance d'édition, comme celle de Levenshtein, qui correspond au nombre minimal de mutations (insertions, délétions, substitutions) pour passer d'une séquence à l'autre. Le calcul de cette distance est relativement simple lorsque la séquence est incluse de manière exacte dans la référence. La complexité augmente cependant lorsque la séquence n'y apparaît que partiellement ou avec des différences. Dans ce cas, l'objectif est de trouver l'alignement qui minimise cette distance d'édition, afin d'identifier l'origine la plus probable de la séquence sur la référence tout en tenant compte des variations.

Plus généralement, les algorithmes d'alignement reposent sur un système matriciel de scores, qui évalue la correspondance entre les séquences comparées. Ainsi, lorsqu'une correspondance parfaite (un match) est observée entre deux nucléotides ou deux protéines, le score augmente. À l'inverse, une différence (un mismatch) entre les deux séquences diminue ce score. La pénalité appliquée à un mismatch peut varier, notamment en fonction de la protéine concernée. En

outre, un gap, qui insère un espace artificiellement dans l'une des séquences pour mieux faire correspondre les autres éléments, peut soit ne pas affecter le score, soit réduire ce dernier selon la méthode d'alignement utilisée (voir Figure 1.3). L'une des méthodes de pénalité de gap la plus réaliste est la pénalité de gap affine, qui sanctionne plus sévèrement une ouverture de gap qu'une extension de gap. Plus le score obtenu est élevé, plus les séquences sont considérées comme similaires.

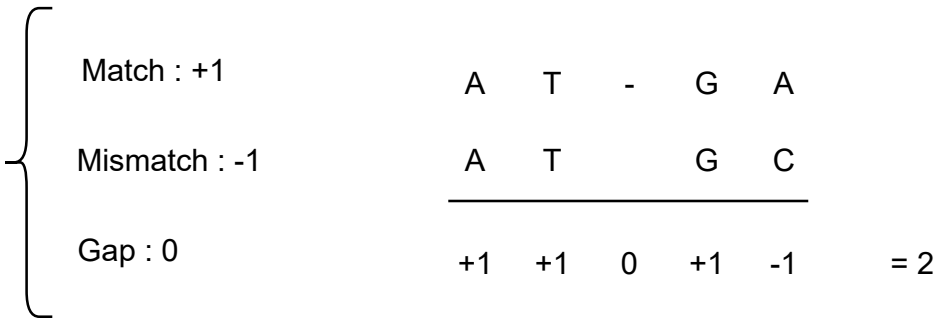


FIGURE 1.3 – *Exemple simple d'alignement illustrant le système de score. L'alignement aura ici un score de 2, grâce à 3 matchs (+3), un gap (0) et un mismatch (-1).*

En fonction de l'application voulue, deux grands types d'alignement sont utilisés. Le choix de leur utilisation va notamment varier en fonction de la longueur des séquences à comparer, ainsi que leur similarité attendue.

Alignement local Ce type d'alignement est particulièrement utilisé pour identifier des similarités entre des régions spécifiques de séquences, en particulier lorsque celles-ci sont très différentes. Il permet de localiser des régions homologues, c'est-à-dire des segments de séquences qui partagent une origine évolutive commune, malgré des différences globales. L'alignement local, qui se concentre sur des sous-séquences plutôt que sur l'ensemble des séquences, repose sur l'algorithme de Smith-Waterman (116). Cet algorithme fonctionne en remplissant une matrice de scores, où chaque cellule représente la meilleure correspondance possible entre des sous-séquences des deux séquences comparées. Une fois la matrice remplie, l'algorithme recherche le chemin optimal à travers cette matrice, afin d'identifier l'alignement local le plus significatif.

Théoriquement, nous avons A et B, deux séquences de longueur m et n respectivement. À l'initialisation, un système de score est fixé :

Soient $s(a, b)$ le score de similarité entre les éléments des deux séquences, et H_k la pénalité de gap appliquée. Une étape préliminaire est tout d'abord de remplir la première colonne ainsi que la première ligne de la matrice H de dimensions $(n+1) * (m+1)$ de 0 :

$H_{k,0} = H_{0,l} = 0 \quad \text{for} \quad 0 \leq k \leq n \quad \text{and} \quad 0 \leq l \leq m.$

Puis, H peut être remplie grâce au système :

$$H_{i,j} = \max \begin{cases} H_{i-1,j-1} + s(a_i, b_j), \\ \max_{k \geq 1}(H_{i-k,j} - W_k), \\ \max_{l \geq 1}(H_{i,j-l} - W_l), \\ 0 \end{cases}$$

où $H_{i-1,j-1} + s(a_i, b_j)$ est le score de l'alignement de a_i et b_j,
$H_{i-k,j} - W_k$ est le score si a_i est à la fin d'un gap de longueur k,
$H_{i,j-l} - W_l$ est le score si b_j est à la fin d'un gap de longueur l,
0 signifie qu'il n'y a aucune similarité entre a_i et b_j.

Une fois la matrice complète, le chemin est reconstitué en partant de la plus grande valeur, puis en remontant jusqu'au premier 0 rencontré.

Un exemple concret est présenté en Figure 1.4. Nous observons sur cette matrice qu'une seule solution optimale peut être trouvée, ce qui n'est pas obligatoirement le cas lorsque plusieurs scores optimaux apparaissent. C'est un algorithme déterministe dans le sens où la matrice, en fonction du système de score proposé, sera toujours créée de la même manière, cependant le chemin d'alignement peut différer lorsque plusieurs solutions optimales apparaissent.

Local alignment

Match : +2 ; Mismatch : -1 ; gap : -1

		A	C	A	C	T	G	A	T
	0	0	0	0	0	0	0	0	0
T	0	0	0	0	0	2	1	0	2
A	0	2	1	3	2	1	1	3	2
C	0	1	4	3	5	4	3	2	2
A	0	3	3	6	5	4	3	5	4
C	0	2	5	4	8	7	6	5	4
T	0	1	4	4	7	10	9	8	10
G	0	0	3	3	6	9	12	11	10

A C A C T G
| | | | | | |
A C A C T G

FIGURE 1.4 – *Exemple d'application de l'algorithme de Smith-Waterman. Le système de score utilisé ici est : match (+2), mismatch (-1) et gap (-1). Le parcours optimal, en bleu, est obtenu en partant de la valeur la plus élevée, ici 12, puis en remontant le chemin ayant à chaque étape le score le plus élevé, et en s'arrêtant au premier 0 rencontré. En rouge sont indiqués le point de départ ainsi que les valeurs qui ne sont pas prises en compte, car elles apparaissent à la suite de la valeur la plus élevée indiquant la fin du chemin optimal.*

Alignement global Contrairement à l'alignement local, qui se concentre sur des régions spécifiques des séquences, l'alignement global est conçu pour comparer des séquences dans leur intégralité. Il repose sur l'algorithme de Needleman-Wunsch (96), qui diffère de celui de Smith-Waterman car il aligne les deux séquences dans leur ensemble, même si leurs extrémités sont inégales ou présentent des divergences.

Le principe de cet algorithme est similaire à celui de Smith-Waterman : un système de matrice de scores est rempli pour évaluer les correspondances entre les séquences, mais la différence

réside dans la manière de rechercher le chemin optimal. Dans l'alignement global, le chemin optimal est retrouvé en partant de la dernière cellule de la matrice et en remontant jusqu'à la première cellule. Comme l'alignement local, l'algorithme de Needleman-Wunsch est exact et garantit le chemin optimal. Ce processus est illustré sur la Figure 1.5.

Global alignment

Match : +2 ; Mismatch : -1 ; gap : -1

		A	C	A	C	T	G	A	T
	0	-1	-2	-3	-4	-5	-6	-7	-8
T	-1	-1	-2	-3	-4	-2	-3	-4	-2
A	-2	1	0	2	1	0	-1	1	0
C	-3	0	3	2	4	3	2	1	0
A	-4	2	1	5	4	3	2	4	3
C	-5	1	4	3	7	6	5	4	3
T	-6	0	3	3	6	9	8	7	9
G	-7	-1	2	2	5	8	11	10	9

```
- A C A C T G A T
  | | | | | |
T A C A C T G - -
```

FIGURE 1.5 – *Exemple d'application de l'algorithme de Needleman-Wunsch. Le système de score utilisé ici est : match (+2), mismatch (-1) et gap (-1). Le principe est similaire à l'algorithme de Smith-Waterman en remontant le chemin optimal, à la différence que les cases de départ et d'arrivée sont obligatoirement les cases en haut à gauche et en bas à droite, et symbolisent l'alignement des deux premiers caractères et des deux derniers.*

La précision de ces algorithmes exacts a un coût extrêmement élevé. Bien qu'ils ne testent pas toutes les combinaisons possibles, ils reposent sur la programmation dynamique, qui entraîne une complexité quadratique. Les complexités en temps des algorithmes de Smith-Waterman et de Needleman-Wunsch sont de $O(mn)$, où m représente la longueur de la première séquence et n celle de la deuxième. Cela limite leur application directe à de longues séquences, comme celles issues du séquençage de troisième génération. Par exemple, pour deux séquences de 10.000 bases, la matrice de scores requise atteindrait déjà 10^8 cellules, qu'il faudrait à la fois calculer, stocker et parcourir, ce qui devient rapidement prohibitif.

Il existe en pratique des alternatives plus efficaces comme le doublage de bande, qui s'exécute en complexité $O(ms)$ pour le temps, proportionnellement à la longueur m de la plus petite séquence et a la distance d'édition s ou le Wave Front Algorithm (WFA) et ces variantes qui s'exécutent en temps $O(n + s^2)$. Même des méthodes plus récentes comme WFA, bien que plus efficaces, voient leurs performances chuter lorsque les séquences sont très divergentes, ce qui est courant avec des séquences longues et bruitées. C'est le cas de A*PA2 (50), capable d'aligner des séquences d'une longueur supérieure à 500.000 paires de bases ayant environ 6% de divergence

en un dixième de seconde en moyenne. C'est 19 fois plus rapide que les outils d'alignement exact de l'état de l'art comme EDLIB (119).

Pour pallier ce problème, il devient nécessaire de recourir à des algorithmes heuristiques. Ce type d'algorithme vise à optimiser le passage à l'échelle et la vitesse d'exécution, au détriment de l'obtention d'une solution exacte. Pour ce faire, c'est uniquement un sous-ensemble des solutions qui est analysé. Seules les solutions les plus probables sont traitées, à l'image par exemple d'un algorithme glouton qui fait des choix locaux optimaux dans l'objectif de construire rapidement une solution satisfaisante. Bien que ces heuristiques ne garantissent pas des solutions exactes, elles offrent l'avantage de calculer des résultats beaucoup plus rapidement, ce qui permet de traiter de grands ensembles de données tout en maintenant un compromis raisonnable entre précision et performance.

Mapping

Contrairement à l'alignement, qui cherche avant tout à mesurer la similarité entre deux séquences, le mapping consiste à projeter des séquences génomiques sur une séquence de référence, dans le but, par exemple, de déterminer leur position dans le génome. Une étape optionnelle d'alignement peut ensuite être faite entre la séquence mappée et la sous-séquence de la référence lui correspondant.

Plusieurs algorithmes permettent d'effectuer cette tâche avec une précision satisfaisante, tout en maintenant une complexité algorithmique raisonnable. Le choix de l'approche de mapping dépend non seulement de la longueur des séquences, mais aussi du taux de divergence attendu avec la référence. Des séquences courtes et peu divergentes permettent un traitement local et précis, tandis que des séquences longues ou présentant une forte divergence exigent des stratégies plus globales et tolérantes aux variations.

Comme nous l'avons vu, l'alignement exact de génomes entiers est particulièrement coûteux en ressources et peu réaliste dans un cadre pratique. Pour pallier cette limite, des algorithmes heuristiques ont été développés. Pour le mapping de séquences plus courtes comme des gènes sur un génome de référence, on privilégiera des approches de type seed-and-extend. À l'inverse, l'anchor chaining sera une solution plus adaptée au mapping de longues séquences comme des genomes complet.

Les deux méthodes seront présentées plus en détails dans cette section.

Seed-and-extend Le principe du seed-and-entend permet de réduire la complexité d'une recherche exacte, en évitant d'être aussi exhaustif que les algorithmes présentés précédemment (Smith-Waterman ou Needleman-Wunsch) qui nécessitaient la construction d'une matrice de scores complète. L'idée principale repose sur deux étapes :

- **Seed** : rechercher un match entre une sous-séquence de la requête sur la référence. Cette étape permet de limiter les recherches infructueuses en prenant pour point de départ une région déjà similaire.
- **Extend** : à partir du match trouvé en première étape, on étend l'alignement des deux côtés avec un système de score à l'image de l'alignement exact. Lorsque le score n'est plus acceptable, l'alignement est stoppé.

Bien que cette méthode soit techniquement applicable à tout type de séquences, son usage en pratique se limite principalement aux séquences courtes. En effet, sur des séquences plus

longues, le coût de l'alignement des extensions deviendrait excessif. De plus, un taux d'erreur élevé dans les séquences complique cette approche, car les correspondances (les matchs) peuvent devenir trop rares ou manquer de précision. Cela rend la méthode peu sensible ou excessivement coûteuse.

L'outil pionnier de cette méthodes est BLAST (Basic Local Alignment Search Tool) (7), devenu un standard de la recherche de similarité entre séquences, et dont les méthodes heuristiques ont amené par la suite d'autres concurrents. On retrouve notamment par la suite Bowtie (68), SOAP2 (75) ou encore BWA (Burrows-Wheeler Aligner) (74) qui utilisent également ces principes. Tous les trois ont été conçus afin de mapper un ensemble de séquences sur de longs génomes, tels que le génome humain. Bowtie a été amélioré en une version Bowtie2 (69), plus efficace sur des séquences légèrement plus longues, et permettant aussi un nombre de gaps plus important.

Anchor chaining L'anchor chaining est une deuxième méthode répandue pour réaliser un alignement de manière non exacte, mais plus adaptée aux séquences longues.

Son principe repose sur trois étapes :

— **Recherche d'ancres** : recherche de matchs entre la requête et la référence.

— **Chaîne d'ancres optimale** : une chaîne maximale d'ancres apparaissant dans le même ordre dans les deux séquences, appelée LCS (Longest Common Sequence) est sélectionnée.

— **Alignement** : une fois les ancres trouvées, des alignements optimaux sont cherchés entre elles.

Bien que le terme d'anchor chaining ne soit pas encore couramment utilisé à l'époque, MUMmer (37), développé en 1999, utilisait déjà ce principe. Les ancres étaient ici des MUM (Maximal Unique Matches) pour ensuite rechercher des sous-séquences communes maximales. Il est capable d'aligner des séquences longues telles que des chromosomes humains (millions de bases), en considérant que les séquences sont très proches.

La comparaison des deux méthodes est schématisée en Figure 1.6.

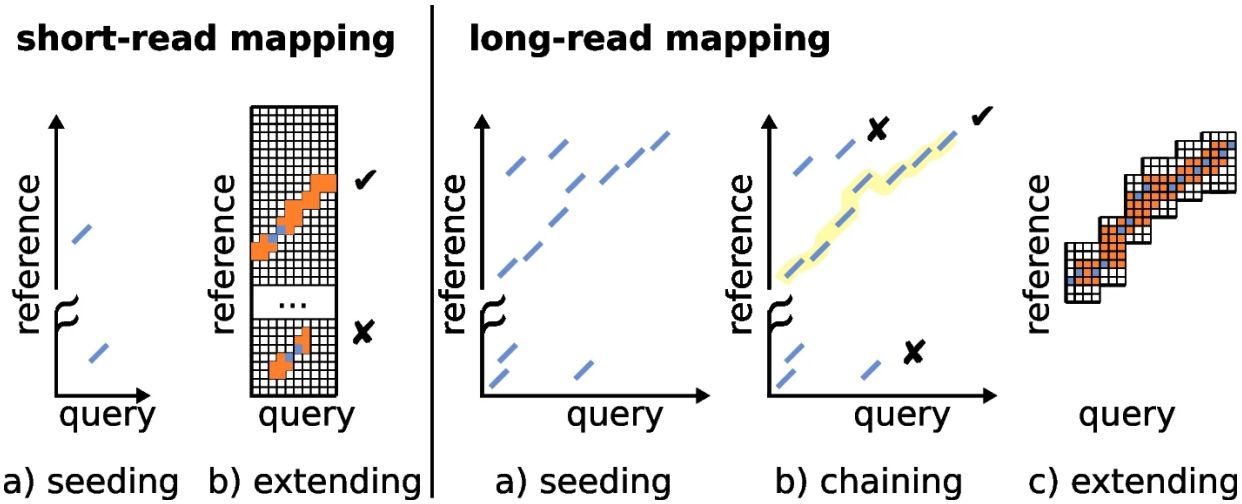


FIGURE 1.6 – *Différences de principe entre seed-and-extend (gauche) et anchor chaining (droite). Dans le cas des séquences courtes, l'unique ancre est indiquée en bleu entre la séquence et la requête, puis est étendue en orange. Pour les longues séquences, plusieurs ancres sont trouvées en bleu, puis le chemin optimal est surligné en jaune avant d'être étendu. Figure reprise de (111).*

Les différentes méthodes et algorithmes d'analyse basés sur une référence ont été présentés, avec pour objectif la comparaison de séquence entre elles, afin d'identifier des particularités

recherchées. Nous avons notamment évoqué l'alignement qui permet d'indiquer la ressemblance entre deux séquences, ainsi que le mapping qui concerne plus le positionnement d'une séquence sur une autre. Dans les deux cas, plusieurs méthodes existent en fonction de l'application, de la longueur des séquences à comparer mais aussi de leur ressemblance supposée.

Bien que ces approches offrent un bon compromis entre qualité et temps d'exécution dans certains cas, elles présentent néanmoins certaines limitations. Les méthodes exactes, bien que précises, sont extrêmement coûteuses en temps et en ressources, rendant leur application impraticable sur des grandes séquences divergentes. À l'inverse, les heuristiques permettent un traitement plus rapide et efficace, mais au prix d'une légère perte de précision. À titre d'exemple, minimap2 s'exécute en $nlog(n)$ en temps, tandis que l'algorithme de Smith-Waterman s'exécute en n^2. Pour les séquences courtes, BWA s'exécute quant à lui en temps linéaire, mais demande de comparer des séquences très similaires. En termes de précision, Minimap2, utilisé sur des longues séquences, est capable d'avoir jusqu'à 97.9% de ses alignements mapQ10 (Mapping Quality, 90% de chances ou plus que l'alignement soit correct) (73). À noter que ce pourcentage dépend tout de même aussi des données en entrées, un taux de répétitions important impactera obligatoirement la qualité de l'alignement.

L'alignement suppose l'existence de génomes de référence, qui servent de support à la comparaison. Toutefois, comme nous le verrons dans les sections suivantes, leur production, par séquençage puis assemblage, représente un défi complexe, pouvant constituer un frein important à ces approches.

1.1.3 Séquençage

La numérisation des séquences d'ADN, plus communément appelée séquençage du génome, a constitué une avancée révolutionnaire qui a transformé de nombreux domaines scientifiques. Au-delà de son rôle central en recherche biomédicale pour l'identification de maladies génétiques ou le développement de thérapies ciblées en oncologie, ses applications sont vastes et touchent de multiples secteurs. En microbiologie, le séquençage permet d'identifier rapidement des agents pathogènes lors d'épidémies, d'étudier la résistance aux antibiotiques et d'analyser les écosystèmes complexes du microbiome. En sciences de l'environnement, il est utilisé pour le suivi de la biodiversité, la détection d'espèces envahissantes et l'étude de l'adaptation des organismes au changement climatique. Le séquençage a également révolutionné l'agronomie, en accélérant la sélection de variétés de plantes plus résistantes ou productives. Il est devenu un outil indispensable en criminalistique pour l'identification de suspects à partir de traces d'ADN, ainsi qu'en paléogénétique, où il permet de reconstituer le génome d'espèces disparues et de mieux comprendre l'histoire évolutive du vivant. L'essor de ces technologies a ainsi ouvert la voie à de nouvelles approches pour comprendre la complexité du vivant, de la cellule à l'écosystème

L'extraction et la numérisation de l'information génétique s'appuient sur des dispositifs spécialisés appelés **séquenceurs**. Le processus commence par la lyse des cellules pour en libérer l'ADN. Cependant la molécule d'ADN est fragile et se fragmente naturellement lors de cette étape. Ainsi les séquenceurs actuels ne peuvent lire que de courts segments d'ADN à la fois. L'information génétique est donc traitée sous forme de fragments, dont la séquence est lue individuellement. Le résultat de chaque lecture est une courte séquence de nucléotides, appelée "read", qui correspond à une portion du génome dont l'emplacement d'origine est inconnu.

Le séquençage fournit les fragments d'ADN nécessaires pour reconstruire un génome via l'opération d'assemblage. Il permet d'identifier des variations génétiques (génotypage), de caractériser des gènes spécifiques, de détecter des similarités entre différents ensembles de données, et d'extraire des informations biologiques essentielles pour de nombreuses applications. Des avancées majeures ont transformé les méthodologies de séquençage depuis les premières expériences réalisées dans les années 1970, et les améliorations se poursuivent avec le développement de nouvelles technologies.

Trois générations de séquençage se sont ainsi succédées :

1. **Séquençage de Sanger** (113) : Cette méthode pionnière, inaugurée en Grande-Bretagne en 1977, est principalement utilisée pour séquencer des régions de taille limitée. Malgré son excellente précision (environ 0.1% d'erreurs), son coût élevé ($500 par mégabase (Mb)) et sa capacité de séquençage limitée (seulement quelques kilobases par run) restreignent son utilité.
2. **Deuxième génération (Next-Generation Sequencing, NGS)** (15) : Dévoilée en 2005, cette ère est largement dominée par Illumina. En tant que technologie de séquençage à haut débit, elle représente une révolution en termes de rentabilité ($0,4 par Mb) par rapport au séquençage de Sanger, avec la capacité de séquencer plusieurs centaines de gigabases (Gb) par run. Cependant, cette efficacité est contrebalancée par des longueurs de reads plus courtes, généralement entre 50 et 300 paires de bases.
3. **Troisième génération (Third-Generation Sequencing, TGS)** (100) : Apparue en 2009, cette ère récente marque l'avènement des reads longs (jusqu'à plusieurs centaines de kilobases), bien que présentant un taux d'erreur potentiellement plus élevé (de 0.1% à 10%). Elle est caractérisée par des technologies innovantes telles que Oxford Nanopore Technologies (ONT) (81) et PacBio (110).

Avec l'apparition des deuxième et troisième générations de séquençage, qui ont permis une réduction drastique des coûts, la quantité de données de séquençage a augmenté de façon constante depuis les années 80 (Figure 1.7). Les technologies de seconde génération ont conduit à la production d'un grand nombre de génomes fragmentés, à cause de la courte longueur des reads. En effet, comme nous le verrons plus loin, ces reads courts peinent à résoudre les nombreuses séquences répétées, qui sont la cause principale de la fragmentation des assemblages. En revanche, les technologies de troisième génération génèrent des reads beaucoup plus longs, capables de couvrir ces régions répétitives et permettant ainsi d'obtenir des assemblages beaucoup moins fragmentés voire complets. Les deux approches se distinguent également par leur profil d'erreur : le séquençage de seconde génération est sujet aux substitutions de bases, tandis que celui de troisième génération produit principalement des insertions et des délétions (indels).

Avec l'explosion des volumes de données génomiques, les méthodes d'analyse basées sur un génome de référence ont rapidement évolué. Elles se sont adaptées aux spécificités des nouvelles générations de séquenceurs en intégrant des algorithmes de plus en plus performants pour gérer et interpréter ces données massives. L'un des défis majeurs est de positionner, ou "mapper", les millions de reads sur ce génome de référence afin d'identifier leur origine, de détecter des variations génétiques (SNPs, variants structuraux) ou de quantifier leur abondance.

Les stratégies d'alignement diffèrent principalement selon la longueur des reads. Pour les reads courts, typiques de la technologie Illumina, l'approche "seed-and-extend" est privilégiée. Elle offre un équilibre entre vitesse et précision, et a été implémentée dans des outils de référence comme BWA (74) et Bowtie (69), optimisés pour ce type de données.

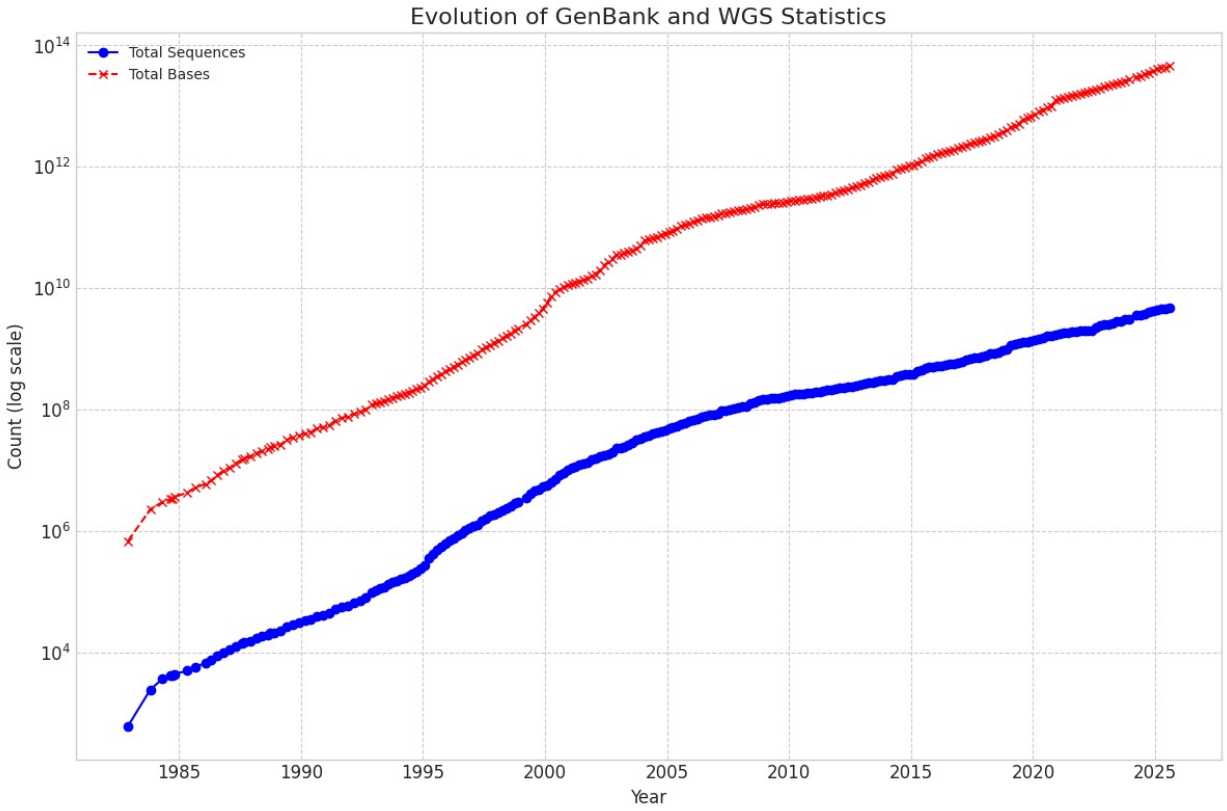


FIGURE 1.7 – *Évolution de la quantité de données de GenBank et Whole Genome Sequencing en fonction du temps. L'augmentation est presque constante, le total de séquences ayant été multiplié par* 10^6 *et le total de bases par un peu moins de* 10^8.

Pour les séquences plus longues, issues des technologies de troisième génération et jusqu'aux génomes entiers, une autre stratégie, l'"anchor chaining", est plus efficace. L'outil de référence dans ce domaine est Minimap2 (72), qui se distingue par sa polyvalence. Il est capable de mapper aussi bien des reads de quelques centaines de paires de bases que des chromosomes entiers et peut gérer efficacement des séquences contenant un taux d'erreur élevé, allant jusqu'à 15%.

Alors que les longs reads offrent une vue d'ensemble du génome et facilitent l'identification de variants structuraux grâce à leur taille plus importante, les reads courts, plus précis en raison de leur faible taux d'erreur, sont plus adaptés aux tâches telles que le génotypage, où une haute précision est cruciale. Cependant, ces approches nécessitent souvent une phase d'assemblage, visant à reconstruire la séquence d'origine à partir des fragments produits par le séquenceur. Cette étape cruciale, mais complexe, fera l'objet de la section suivante. Nous y présenterons les principales méthodes d'assemblage, en mettant en évidence leurs principes, leurs atouts, ainsi que les limites qu'elles peuvent présenter.

1.2 Assemblage

L'assemblage génomique est l'étape qui consiste à reconstituer une séquence d'ADN la plus continue et complète possible à partir des reads produits par les séquenceurs. Le principe repose sur l'identification des régions de chevauchement entre ces reads pour les ordonner et les fusionner, comme l'illustre la Figure 1.8.

La qualité de la reconstruction d'un génome dépend fortement des technologies de séquençage, qui déterminent des caractéristiques essentielles comme la longueur des reads et leur taux

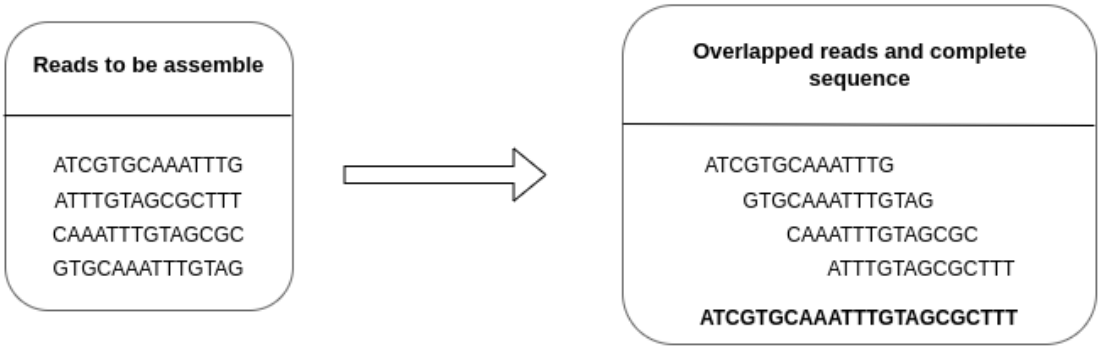


FIGURE 1.8 – *Exemple d'assemblage. Quatre reads (à gauche) sont alignés grâce à leurs régions de chevauchement pour reconstituer une séquence unique plus longue (à droite).*

d'erreur. La longueur de ces fragments est un facteur critique qui influe directement sur la complexité et la continuité de l'assemblage final.

Les reads courts, en raison de leurs chevauchements réduits, génèrent des ambiguïtés. Si une séquence de chevauchement est présente à plusieurs endroits du génome, de multiples possibilités d'assemblage émergent, ce qui fragmente la reconstruction et compromet sa fiabilité. Ce problème, illustré par la Figure 1.9, est particulièrement prononcé dans les régions répétées, qui constituent un défi majeur en génomique.

À l'inverse, les technologies plus récentes produisant des longs reads génèrent des chevauchements plus étendus et souvent uniques. Un seul read long peut couvrir une répétition dans son intégralité ainsi que ses régions flanquantes, levant ainsi toute ambiguïté sur sa position. Cet avantage est décisif pour résoudre les régions complexes et obtenir un assemblage de génome beaucoup plus complet et continu.

Cependant, ce gain en longueur s'accompagne parfois d'un compromis sur la précision. Certaines technologies de troisième génération, comme celles d'Oxford Nanopore (ONT), peuvent présenter un taux d'erreur élevé. D'autres, comme les reads HiFi, combinent longueur et haute fidélité avec un taux d'erreur inférieur à 1%. Un faible taux d'erreur est non seulement crucial pour la justesse de la séquence, mais il est également indispensable pour distinguer avec certitude des régions génomiques très similaires mais non identiques, comme des gènes dupliqués ou des allèles différents. La gestion de ces erreurs est donc primordiale, car elles peuvent perturber l'identification des chevauchements et introduire des défauts dans l'assemblage final.

L'exemple le plus emblématique de ces défis est sans doute l'assemblage du génome humain. Avec une taille d'environ 3 milliards de paires de bases (3 Gb), la construction d'un génome de référence complet représente un défi majeur. Pendant près de deux décennies, les chercheurs ont dû travailler à partir d'un génome partiel, comportant des régions incomplètes et difficiles à assembler. Ce n'est qu'en 2022, grâce aux avancées technologiques et aux nouvelles stratégies d'assemblage, que le Telomere-to-Telomere (T2T) Consortium a réussi à produire un génome humain entièrement séquencé (53, 98). Ce travail a permis de combler les lacunes des versions précédentes et d'obtenir une séquence complète, de télomère à télomère. La difficulté de cette tâche provenait principalement des très grandes répétitions, des répétitions inexactes, des séquences de faible complexité comme les télomères et les séquences centromériques, qui sont elles-mêmes des répétitions imbriquées à plusieurs niveaux.

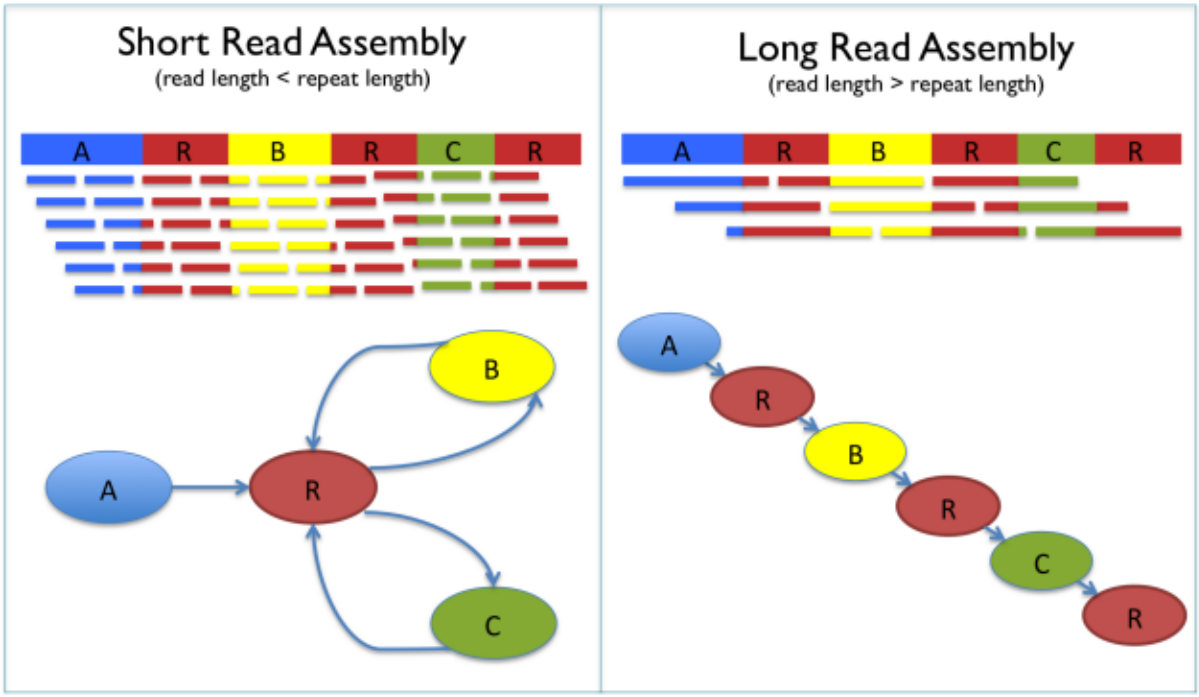


FIGURE 1.9 – *Illustration de l'assemblage appliqué aux reads courts (gauche) et long reads (droite). On observe que les longs reads permettent moins d'ambiguïtés, couvrant plus facilement les répétitions. Cette figure est reprise de (70).*

Afin de mieux comprendre les défis liés à l'assemblage et les solutions mises en place, les sections suivantes présentent les principales méthodes d'assemblage.

1.2.1 Méthodes d'assemblage de novo

Pour reconstruire un génome à partir de fragments de reads, deux stratégies s'opposent : **l'assemblage de novo** et **l'assemblage guidé par référence**. L'approche de novo est la plus fondamentale, puisqu'elle reconstitue la séquence originale en se basant uniquement sur les chevauchements entre les reads bruts, sans aucune connaissance préalable du génome. À l'inverse, il est aussi possible d'utiliser un génome de référence, si celui-ci existe, pour guider l'assemblage en y alignant les reads. En effet, deux individus d'une même espèce auront des génomes extrêmement ressemblants. Cependant, cette méthode souffre de limitations critiques. Le problème le plus évident est la disponibilité : un génome de référence de haute qualité n'existe tout simplement pas pour la grande majorité des espèces. Plus fondamentalement encore, l'utilisation d'une référence introduit un biais majeur. L'assemblage est contraint par la structure de la référence, ce qui empêche la découverte de variations structurelles importantes (comme de grandes insertions ou délétions) ou de gènes nouveaux présents dans l'échantillon mais absents de la référence. De plus, toutes les erreurs, les régions manquantes ou les approximations présentes dans le génome de référence seront inévitablement propagées au nouvel assemblage. Face à ces inconvénients qui risquent de masquer une part importante de la réalité biologique, l'assemblage de novo représente la seule approche capable de construire une image complète et non biaisée d'un génome. C'est pour cette raison que nous nous concentrerons exclusivement sur cette méthode dans cette section.

Ces méthodes reposent principalement sur des modèles basés sur des graphes, qui permettent de représenter les relations entre les reads et de guider l'assemblage. Trois principaux

paradigmes existent : le greedy assembly ou "assemblage glouton" (56), OLC (Overlap Layout Consensus) (94), ainsi que les graphes de de Bruijn (28).

Greedy assembly L'approche d'assemblage glouton (greedy assembly) repose sur une heuristique simple : à chaque étape, l'algorithme fait le choix qui semble localement optimal. Le principe est que deux reads partageant un long chevauchement ont une forte probabilité d'être adjacents dans le génome. L'algorithme démarre donc avec une séquence initiale (un contig) et l'étend itérativement en y fusionnant le read non encore utilisé qui présente le plus grand et meilleur chevauchement. Ce processus se répète jusqu'à ce qu'aucune extension ne soit possible.

Si cette méthode est rapide, son principal défaut est son incapacité à revenir en arrière pour corriger une décision. Cette faiblesse la rend particulièrement vulnérable aux séquences répétées du génome. En présence d'une répétition, plusieurs reads issus de régions distinctes peuvent présenter des chevauchements de taille similaire avec le contig en cours d'élongation. L'algorithme glouton risque alors de choisir le mauvais read, intégrant un fragment d'une autre partie du génome et créant une erreur d'assemblage irréversible.

En raison de ces limitations, l'approche gloutonne a été largement supplantée par des méthodes plus robustes, comme celles basées sur les graphes (Overlap-Layout-Consensus ou de Bruijn). Elle n'est plus utilisée que pour des génomes très simples ou comme étape préliminaire dans certains anciens outils, à l'image de CAP3 (56), avant des phases de raffinement.

Overlap Layout Consensus (OLC) À la différence des approches gloutonnes qui opèrent par décisions locales successives, le paradigme OLC (Overlap-Layout-Consensus) adopte une stratégie globale. En construisant d'abord un graphe de tous les chevauchements, cette méthode est plus robuste pour résoudre les ambiguïtés structurelles du génome, notamment celles dues aux séquences répétées.

Cette approche se décompose en trois étapes fondamentales, illustrées sur la Figure 1.10 :

1. **Overlap (Chevauchement)** : Cette première phase consiste à comparer l'ensemble des reads entre eux afin d'identifier tous les chevauchements significatifs. Ces relations sont ensuite matérialisées par un graphe de chevauchements, où les nœuds sont les reads et les arêtes représentent les alignements possibles entre eux.
2. **Layout (Agencement)** : L'objectif est de simplifier le graphe, souvent très complexe, pour en déduire l'ordre et l'orientation des reads le long du génome. Cette étape cruciale recherche les chemins non ambigus et linéaires qui correspondent aux contigs. Le processus implique généralement le "nettoyage" du graphe pour retirer les arêtes parasites et résoudre les variations simples, afin de ne conserver que les chemins les plus plausibles.
3. **Consensus** : Pour chaque chemin défini à l'étape précédente, une séquence finale de haute fidélité est générée. Cette étape de consensus produit un alignement multiple de tous les reads couvrant le contig. La séquence finale est alors déduite en choisissant, pour chaque position, la base la plus fréquente parmi tous les reads alignés. Ce vote majoritaire permet de corriger efficacement les erreurs de séquençage et d'améliorer la qualité de l'assemblage.

De nombreux outils implémentent ce paradigme, comme Miniasm (71), Celera (38) ou encore Canu (66). En réalité, il en existe plusieurs dizaines.

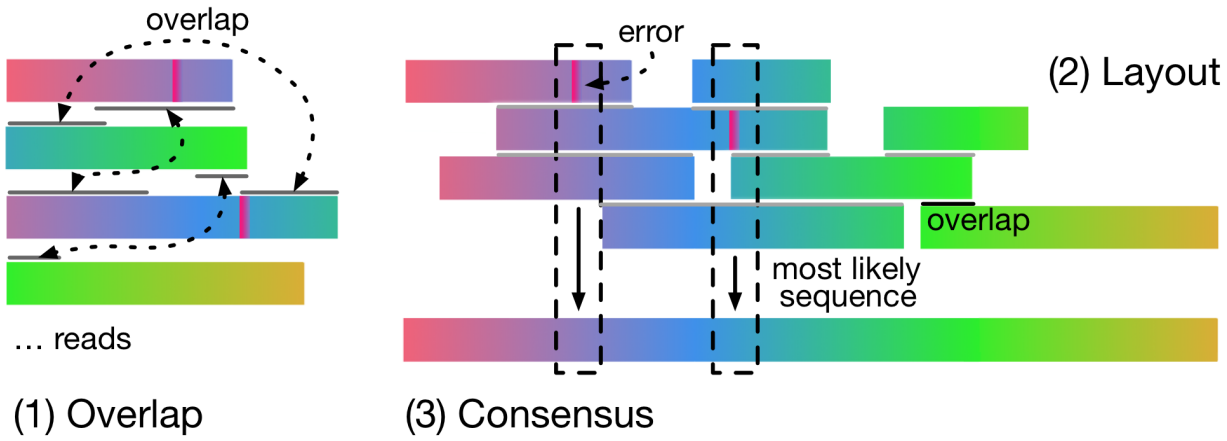


FIGURE 1.10 – *Illustration du paradigme d'overlap layout consensus. Figure reprise de* `https://github.com/UCLA-VAST/minimap2-acceleration`.

Ce paradigme est cependant très coûteux en temps comme en mémoire, principalement à cause de la première étape qui consiste à identifier tous les chevauchements possibles entre les reads. Cette phase nécessite de comparer chaque read avec tous les autres, une approche qui, dans le pire des cas, a une complexité temporelle quadratique $O(n^2)$, où n est le nombre de reads. Pour un projet de séquençage générant des millions, voire des milliards de reads, le nombre de comparaisons à effectuer devient astronomique, constituant un goulot d'étranglement.

Sur le plan de la mémoire, la construction du graphe de chevauchement (overlap graph) est également très exigeante. Ce graphe, où chaque read est un noeud et chaque chevauchement un arc, peut devenir immense et complexe, en particulier pour certains génomes de grande taille. Le stockage de cette structure de données, ainsi que de l'ensemble des reads, requiert une quantité de mémoire vive considérable, limitant souvent l'application de cette méthode aux seules machines disposant de très grandes ressources.

Graphes de de Bruijn Le graphe de de Bruijn (Figure 1.11) est un graphe orienté qui représente les chevauchements entre des séquences de symboles, dans notre cas, des nucléotides. Ses principales caractéristiques sont :

— Les séquences dans ce graphe ont une longueur de k, et sont appelées k-mers.
— Le chevauchement entre ces k-mers couvre une longueur de k-1.
— Les sommets du graphe représentent les k-mers.
— Les arcs illustrent les chevauchements entre les k-mers.

Dans le cadre des graphes de De Bruijn, un read correspond alors à un chemin traversant une suite de noeuds dans le graphe. Lorsque plusieurs reads partagent des k-mers, leurs chemins fusionnent, permettant de connecter les différentes séquences et de reconstruire des segments plus longs du génome en suivant ces parcours.

Cependant, la présence de régions répétées dans le génome complexifie le graphe. Un k-mer issu d'une séquence répétée agit comme un carrefour, possédant plusieurs arêtes entrantes et sortantes qui ne peuvent être résolues sans informations supplémentaires. Cette ambiguïté structurelle empêche de déterminer le chemin correct, menant à des assemblages fragmentés qui s'interrompent aux extrémités de ces répétitions.

Cette structure est difficilement adaptable pour les données longs reads. En effet, trouver un chemin pour réaliser des assemblages dans ce type de données reconstituant fidèlement la séquence se révèle être extrêmement difficile dans de nombreux cas :

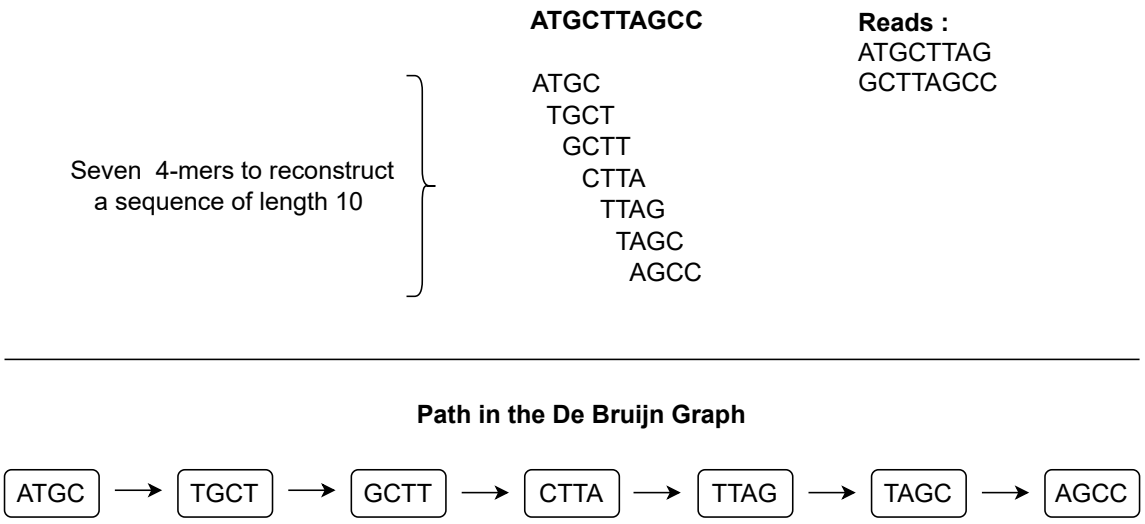
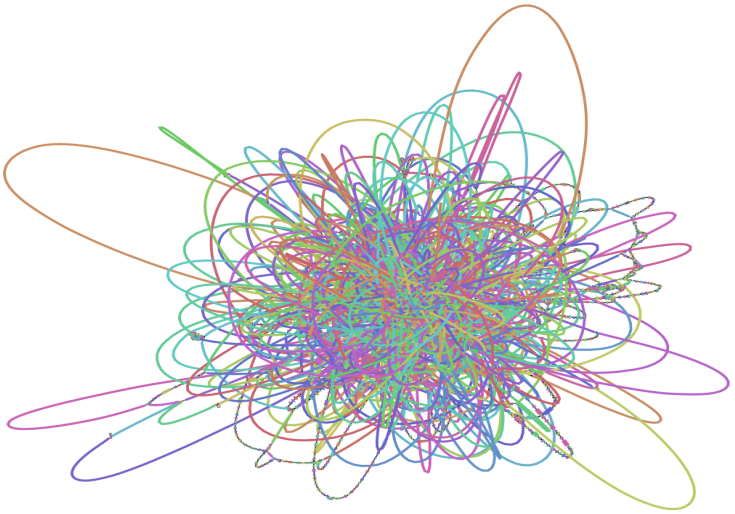
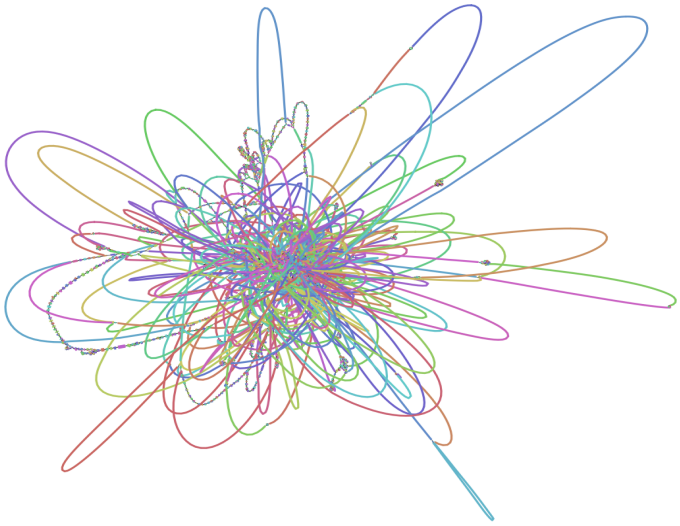


FIGURE 1.11 – *Cas simple de graphe de De Bruijn représentant un jeu de données de 2 reads de longueur 8, découpés en 7 k-mers de longueur 4. Le graphe de de Bruijn est présenté en dessous.*

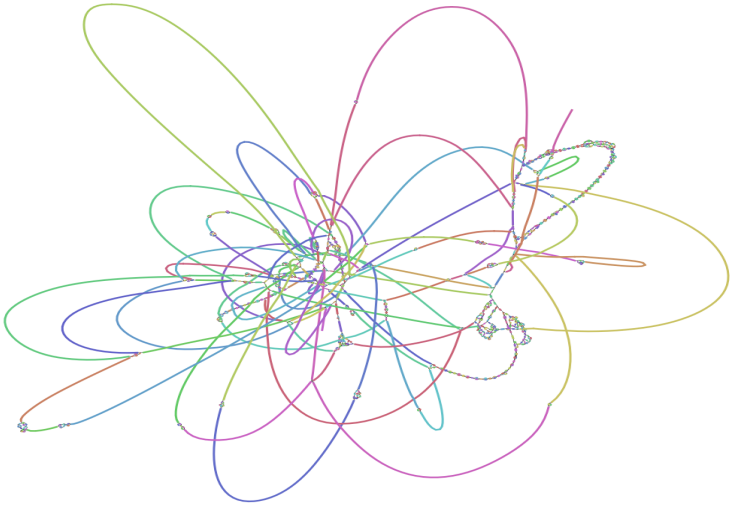
— Enraison du taux d'erreur parfois élevé, la probabilité que les k-mers se chevauchant entre les reads soient parfaitement identiques devient très faible, ce qui pulvérise le graphe en une multitude de noeuds déconnectés.
— Plus le taux d'erreur est élevé, plus le nombre de k-mers différents sera important et plus le graphe sera complexe. Pour une erreur dans un read découpé en k-mers de taille 31, 31 d'entre eux comporteront donc une base erronée et risqueront de fausser et complexifier le graphe.
— La construction du graphe dépend aussi beaucoup de son paramètre k, qu'il est impératif de bien choisir au préalable. En effet, une valeur de k trop petite amènera trop d'ambiguïtés (répétitions), tandis qu'une valeur de k trop grande fragmentera le graphe par manque d'overlaps.
— Ces inconvénients peuvent entraîner une perte d'informations au niveau des variations génétiques, en particulier les variations rares ou dans les régions complexes.



(a)



(b)



(c)

FIGURE 1.12 – *Comparaison des graphes de de Bruijn créés à partir de k-mers de différentes longueurs issues du génome de E. coli : 1.12a longueur 21 (haut), 1.12b longueur 31 (centre), 1.12c longueur 63 (bas).*

Malgré des avancées significatives, l'assemblage fait face à des défis majeurs qui limitent encore notre capacité à obtenir des séquences complètes et parfaites. Le passage à l'échelle reste un obstacle, l'assemblage de grands génomes ou de métagénomes complexes restant coûteux en ressources, tant en temps qu'en mémoire. L'objectif d'un assemblage de type "telomere-to-telomere"

se heurte à la complexité des génomes eucaryotes, notamment les grandes répétitions en tandem, les répétitions mosaïques et les séquences de faible complexité qui empêchent la résolution complète des chromosomes. De plus, l'assemblage de génomes diploïdes ou polyploïdes, comme celui de l'humain, présente une difficulté considérable pour la séparation correcte des haplotypes, chaque région homozygote agissant comme une répétition. Enfin, la précision à l'échelle de la base unique n'est pas encore garantie. Les technologies spécialisées longs reads conservent des biais, particulièrement sur les homopolymères, qui peuvent persister après l'étape de consensus, tandis que les reads courts souffrent d'un biais GC qui entraîne une profondeur inégale de certaines régions.

Par conséquent, l'obtention d'un génome exact à la base près reste, à ce jour, un objectif largement hors de portée. Seuls les génomes bactériens font exception, leur structure étant moins complexe et leur taille réduite par rapport à des génomes eucaryotes. L'assemblage constitue donc un réel challenge et représente une opération destructrice fortement dépendante de la qualité et des caractéristiques des données d'entrée (taux d'erreurs, profondeur de séquençage, répétitions) ainsi que des paramètres sélectionnés (128). Même dans des circonstances idéales, des éléments structurels peuvent être incorrects (mauvais assemblages), et certaines régions à faible profondeur ou répétitives peuvent être exclues de l'assemblage. Par conséquent, générer des séquences de référence de haute qualité, représentant des exemplaires de séquences nucléiques assemblées, devient une tâche redoutable.

Face à ces contraintes, une approche alternative consiste à travailler **directement avec les données brutes issues des séquenceurs**, en contournant les étapes de prétraitement et sans recourir à l'assemblage. Cette stratégie de novo permet d'exploiter pleinement l'information génomique sans introduire de biais liés à l'assemblage.

1.3 Les méthodes de novo sans assemblage

L'assemblage de novo, bien qu'indispensable pour l'étude de nouveaux génomes, reste une opération complexe et coûteuse en ressources. Il impose de résoudre les ambiguïtés entre les reads qui se chevauchent, un processus où des choix difficiles peuvent mener à des erreurs d'assemblage. Une alternative efficace consiste à travailler directement sur l'ensemble des k-mers extraits des reads, ce qui est simple à générer à partir des données brutes. Un ensemble de k-mers peut être vu implicitement comme une définition d'un graphe de de Bruijn, car il contient les mêmes informations. Cette structure de données représente l'information génomique de manière efficace car la redondance présente dans les reads est abstraite, un k-mer partagé par de nombreux reads ne sera représenté qu'une seule fois dans le graphe. Cette approche offre deux autres avantages majeurs :

— Elle intègre un mécanisme de correction d'erreurs robuste : les k-mers issus d'erreurs de séquençage sont statistiquement rares et peuvent être écartés en fixant un seuil d'abondance minimale. Cela permet de "nettoyer" les données brutes et de simplifier drastiquement le graphe avant même l'assemblage.
— En se basant sur les k-mers plutôt que sur les reads entiers, cette méthode évite l'étape coûteuse de comparaison de tous les reads entre eux, permettant des analyses rapides "sans alignement".

Le principal compromis de cette méthode est la perte de l'information contenue dans le read individuel. Cependant, cette perte peut être contrôlée par le choix de la valeur de k. Une grande valeur de k préserve davantage le contexte de la séquence, ce qui est crucial pour résoudre les zones répétitives et limiter les ambiguïtés dans le graphe. En contrepartie, un k trop grand devient plus sensible aux erreurs de séquençage, car une seule erreur dans un read corrompra un plus grand nombre de k-mers. Le choix de k est donc un paramètre fondamental pour trouver le juste équilibre entre la simplification des données et la préservation de l'information biologique nécessaire à une reconstruction fidèle du génome.

À quelles applications est adaptée l'analyse de novo des séquences génomiques ?

Génotypage Les graphes de de Bruijn sont largement utilisés en génotypage de novo, pour la détection de variations. Une approche courante repose sur la détection des "bulles" dans le graphe, une méthode exploitée par des outils comme KISSNP2 (121). Lorsqu'une variation isolée apparaît dans une séquence, le graphe de de Bruijn génère deux chemins distincts, correspondant aux deux versions alternatives du segment génomique. Comme illustré sur la Figure 1.13, ces chemins forment visuellement une bulle, qui peut ensuite être identifiée et analysée pour détecter la présence d'un variant. Cette approche est particulièrement efficace pour traiter les grands volumes de données issues du séquençage haut débit (reads courts), où les méthodes basées sur les graphes de de Bruijn permettent une détection rapide et robuste des variations génétiques.

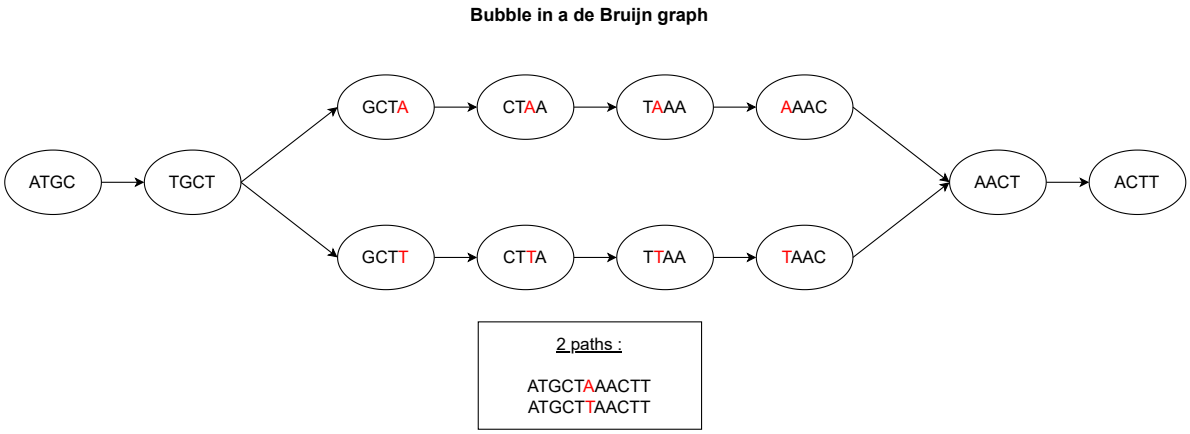


FIGURE 1.13 – *Exemple de bulle dans un graphe de de Bruijn, provoquant l'apparition de 2 chemins distincts dû à une base de différence (A dans le chemin du haut, T dans celui du bas). Deux contigs, GCTAAAC et GCTTAAC, sont créés.*

Des outils tels que discoSNP (121), et ses extensions DiscoSNP++ (105) ou DiscoSnp-RAD (47), permettent cette application. Dans le même objectif, on peut également citer LAVA (Lightweight Assignment of Variant Alleles) (114) ou encore Kmer2SNP (76), plus récent. Ils sont conçus pour être utilisés sur des reads courts.

Quantification L'utilisation de l'abondance des k-mers est d'abord une étape cruciale pour la gestion des erreurs de séquençage. Les k-mers à très faible abondance sont souvent le signe d'erreurs de séquençage et peuvent être filtrés. Cette correction améliore significativement la fiabilité des analyses en aval, comme l'assemblage de génome.

En génomique, une fois les données nettoyées, l'analyse de la distribution des fréquences de k-mers permet d'estimer des caractéristiques fondamentales d'un génome, comme sa taille, son

hétérozygotie ou sa ploïdie. Cette approche est également efficace pour différencier les variations génétiques d'erreur de séquençage ou de répétitions inexactes.

En métagénomique, l'abondance permet d'analyser la composition d'écosystèmes. En comparant les *k*-mers d'un échantillon à des bases de données de génomes connus, il est possible d'estimer l'abondance relative des différentes espèces présentes (profilage taxonomique) et d'évaluer la diversité microbienne.

Enfin, en transcriptomique (RNA-seq), le comptage de *k*-mers offre une alternative rapide et sans alignement pour quantifier l'expression des gènes. L'abondance de *k*-mers uniques à un transcrit sert d'indicateur direct de la quantité d'ARN de ce gène, facilitant la comparaison des niveaux d'expression entre différentes conditions expérimentales.

Des outils comme Kallisto (23) et Reindeer (85) permettent de quantifier ces *k*-mers sur un ou plusieurs jeux de données. Kallisto construit un graphe de de Bruijn coloré où chaque *k*-mer est associé à un transcriptome via une table de hachage. Le graphe permet ensuite de générer des chemins, qui sont quantifiés individuellement. Un processus de pseudo-alignement identifie les transcrits compatibles avec chaque read, puis un modèle probabiliste permet d'estimer les abondances relatives. Reindeer, quant à lui, est conçu pour passer à l'échelle et analyser de très grandes cohortes de jeux de données. Pour ce faire, il s'appuie sur des outils performants comme BCALM2 (29) ou GGCAT (33) pour construire les graphes de De Bruijn individuels, pour ensuite construire un graphe global avec les informations de présence et d'abondance de chaque *k*-mer dans chaque échantillon. Cette représentation compacte permet des analyses comparatives efficaces sur des milliers de jeux de données.

Classification L'analyse de novo est également adaptée aux tâches de classification, par exemple en métagénomique, où il s'agit d'identifier rapidement l'origine taxonomique des séquences issues d'un échantillon. Des outils comme Kraken (129) ont été développés pour répondre à ce besoin. Kraken repose sur une base de données contenant les *k*-mers issus de génomes de référence, chacun associé à son Lowest Common Ancestor (LCA, noeud le plus bas étant ancêtre commun aux deux *k*-mers). Lors de l'analyse, chaque read est décomposé en *k*-mers qui sont comparés à cette base. Une classification est ensuite effectuée en attribuant à chaque read le plus petit ancêtre commun dans l'arbre taxonomique des identifiants rencontrés.

Comparaison / clustering La comparaison entre jeux de données génomiques est également possible, notamment lorsqu'il s'agit d'évaluer la similarité en termes de kmer partagés entre datasets. Des outils comme Commet (82) comparent les métagénomes via une indexation exhaustive de type "all-versus-all". Pour réduire les coûts de calcul, Mash (99) utilise le sketching. Il crée une empreinte de chaque jeu de données en ne gardant que les *k*-mers ayant les plus faibles valeurs de hachage (MinHash). La comparaison de ces sketches estime efficacement la similarité des données, ce qui est utile pour le clustering de génomes ou la détection de contamination. Des outils dérivés ont suivi des objectifs différents : Dashing (136) et BinHash (136) visent une accélération maximale du processus, tandis que Sourmash (106) ajoute des fonctionnalités et une robustesse accrues, au détriment de la performance.

Correction Les graphes de de Bruijn peuvent aussi être employés pour corriger des reads (77) (112) en utilisant l'abondance des *k*-mers dans le graphe. Cette analyse d'abondance permet de distinguer les séquences correctes des erreurs de séquençage. Dans un jeu de données, un *k*-mer "solide", c'est-à-dire sans erreur, apparaîtra un nombre de fois proche de la profondeur de sé-

quençage moyenne. À l'inverse, un *k*-mer contenant une erreur de séquençage sera beaucoup plus rare, souvent unique. L'analyse de la distribution des fréquences de tous les *k*-mers (le spectre de *k*-mers) révèle ainsi typiquement deux pics : un premier à très faible abondance correspondant aux erreurs, et un second centré autour de la profondeur *p* correspondant aux *k*-mers solides (voir Figure 1.14). Cette distinction est exploitée pour la correction. Un read contenant une erreur se traduira dans le graphe de de Bruijn par un chemin qui inclut un ou plusieurs *k*-mers de faible abondance. Ce "chemin erroné" forme généralement une "bulle" ou une "branche morte" qui diverge du chemin principal, lequel est composé de *k*-mers solides à haute fréquence. Les algorithmes de correction parcourent alors le graphe. Lorsqu'ils rencontrent un *k*-mer de faible abondance dans un read, ils recherchent un chemin alternatif et proche, composé uniquement de *k*-mers solides. S'il existe un tel chemin, l'algorithme peut remplacer la base erronée dans le read par celle suggérée par le chemin solide, corrigeant ainsi l'erreur.

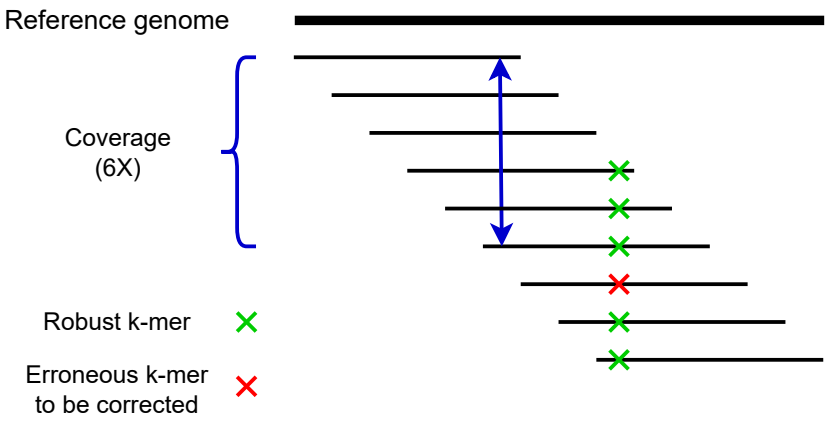


FIGURE 1.14 – *Jeu de données de reads, ayant une profondeur de 6X. À une même position ici symbolisée par les croix, si un k-mer est différent de tous les autres (une seule occurrence sur 6 minimum possibles), on peut supposer qu'il est erroné et à corriger.*

Compression Enfin, compte tenu des volumes massifs produits par les séquenceurs, la compression des données devient cruciale. Plusieurs outils se basent sur le contenu en *k*-mers des reads pour améliorer le taux de compression.

Certains réalisent un pré-assemblage pour y mapper les reads et les réordonner par rapport à une référence implicite : Quip (60), HARC (27) et Minicom (80). Dans le même esprit, LEON encode les reads comme des chemins dans un graphe de Bruijn probabiliste servant de référence (16).

Une autre famille d'approches réordonne les reads en regroupant les séquences similaires, selon leur proximité en *k*-mers, afin d'exploiter des motifs locaux pour une compression plus efficace. COIL (127) et ReCoil (130) recherchent des recouvrements suffixe–préfixe. BEETL (32), quant à lui, optimise l'ordre des reads pour des compressions basées sur la BWT. ORCOM (49) et MINCE (103) s'appuient sur les minimizers pour regrouper les reads. BdBG (124) suit une logique proche, en combinant ce réordonnancement avec une exploration de chemins à la manière de LEON.

En conclusion, ces approches de novo permettent donc d'extraire ou de réaliser des opérations essentielles sur les séquences, sans nécessiter leur positionnement explicite sur un génome.

C'est précisément dans ce cadre que s'inscrit mon projet de thèse, qui explore l'étude des séquences génomiques à partir des données brutes, en s'appuyant notamment sur des techniques d'indexation. Le chapitre suivant présente un état de l'art des approches existantes, et introduit les contributions principales développées au cours de cette thèse.

Chapitre 2

Indexation

L'indexation s'est rapidement imposée comme une étape clé pour éviter de parcourir plusieurs fois les mêmes données, notamment lorsqu'un grand nombre de requêtes doit être effectué au cours de l'analyse. On distingue généralement deux phases : la construction de l'index, qui organise les données dans une structure adaptée, et la phase de requête, qui exploite cette structure pour répondre efficacement à des interrogations multiples. L'objectif principal est de garantir des temps de réponse sous-linéaires, idéalement logarithmiques ou constants, tout en maintenant une empreinte mémoire maîtrisée.

Comme vu précédemment, l'explosion des volumes de données génomiques accessibles pose des défis majeurs en termes de passage à l'échelle. L'analyse comparative de ces données repose souvent sur la disponibilité d'un génome de référence, qui n'est pas toujours existant ou pertinent. Dans ces cas, une étape d'assemblage est nécessaire, mais elle reste coûteuse et approximative, car un assemblage parfait est difficile à obtenir. Par ailleurs, les opérations d'alignement exact entre séquences, bien qu'essentielles, sont notoirement gourmandes en temps de calcul ; leur efficacité repose donc sur l'emploi d'heuristiques et de structures d'indexation adaptées.

L'un des objectifs centraux dans l'étude des séquences est de retrouver rapidement une sous-séquence donnée dans un long texte, par exemple l'occurrence d'un gène particulier dans un génome complet. L'indexation permet d'éviter un recalcul systématique des alignements, de réduire le coût en mémoire et en temps de traitement, et d'optimiser l'espace de stockage grâce à des représentations compactes.

Au-delà de la recherche exacte, la recherche inexacte (ou recherche approchée) visant à identifier des séquences similaires mais non identiques, en tolérant un certain nombre d'erreurs jouent un rôle fondamental dans le traitement de séquences biologiques, où les mutations, insertions et délétions sont fréquentes mais se situent en dehors du cadre de cette thèse.

On distingue trois grands types de requêtes, classés du plus précis au plus général :

— **Recherche des positions** d'une séquence dans un jeu de données.
— **Recherche de l'abondance** d'une séquence, obtenue en comptant ses occurrences, ce qui peut être déduit de la recherche de positions.
— **Recherche de la présence** d'une séquence, consistant à déterminer si elle apparaît au moins une fois, ce qui peut être dérivé de la recherche d'abondance.

Chaque type de requête englobe le suivant : connaître les positions permet d'estimer l'abondance, et l'abondance permet de déduire la présence.

Nous détaillerons le principe de ces requêtes dans les points ci-dessous.

Présence L'information de présence/absence est utile pour filtrer les données avant des opérations coûteuses, par exemple avant un alignement. Lors de la recherche d'une séquence dans un dataset, si tous les k-mers de la séquence sont retrouvés, alors la séquence est probablement présente. Toutefois, la présence des k-mers est une condition nécessaire mais non suffisante : des k-mers peuvent provenir de régions distinctes ou de répétitions et ne pas former la séquence cible.

Abondance L'abondance d'une séquence est centrale pour la correction de reads, la détection de SNPs et, plus largement, pour la caractérisation de la variabilité. À partir des abondances de k-mers, on déduit ensuite l'abondance d'une séquence. Cette information sert aussi de filtre : un k-mer de très faible abondance sera souvent ignoré car probablement erroné, tandis qu'un k-mer dont l'abondance s'écarte des distributions attendues peut signaler une variation ou une répétition. Au-delà de la simple distinction erreur/hétérozygotie/homozygotie ou répétition, ces abondances alimentent de nombreuses applications en génomique. En transcriptomique, elles sont également cruciales pour la quantification des transcrits et, plus généralement, des génomes présents dans l'échantillon.

Positions Il est souvent requis d'identifier la séquence d'origine d'un élément, voire sa position exacte, notamment pour les techniques d'alignement et de mapping. La seule présence des k-mers ne suffit pas : il faut vérifier la cohérence de leurs positions relatives, c'est-à-dire qu'ils se situent globalement dans la même région, dans le même ordre, avec des décalages compatibles avec la longueur de k-mer et l'orientation attendue. Cette étape permet de distinguer les occurrences réelles des artefacts liés aux répétitions ou aux collisions, et prépare un éventuel affinage par alignement local.

Deux principales problématiques apparaissent ici : le stockage des données à étudier et leur utilisation. Ces données doivent être stockées dans des structures qui seront utilisées pour répondre à des problématiques par la suite. Au vu de la quantité de données, les accès aux structures sont coûteux (nombre d'accès élevé, les accès ne sont pas toujours directs). La complexité algorithmique des opérations de recherche dans ces structures se doit d'être la plus faible possible. On peut penser par exemple à celles-ci :

- Chercher un motif dans un texte est linéaire en fonction de la taille du texte
- Chercher un élément dans une liste triée est logarithmique
- Chercher une association dans une table de hachage est constant

Choisir une structure adaptée permet donc de réduire significativement le temps de requête, au prix, parfois, d'une consommation mémoire augmentée.

Par exemple, construire une table de hachage sur tous les k-mers de taille k dans un texte de taille N a un coût en mémoire de $O(kN)$. Stocker ces mêmes k-mers dans une liste triée aurait la même complexité. Par contre, en termes de requêtes, l'accès à une table de hachage est constant, une recherche dans une liste triée est logarithmique. C'est donc sur cet aspect que le gain est visible.

Deux grandes familles d'index existent et diffèrent notamment de par les informations qu'elles stockent et les requêtes possibles :

- **Les index basés sur le hachage**, qui établissent une correspondance explicite entre des éléments du jeu de données (par exemple les k-mers) et des informations associées (abondance, position, appartenance, etc.), selon une structure de type clé-valeur. Ces index sont

généralement très performants en termes de temps de requête, et ne stockent pas obligatoirement tout le texte.

— **L'indexation full-text**, qui est une méthode entièrement textuelle qui ne découpera pas l'entrée, mais la transformera de manière à y effectuer des requêtes. Leur développement vise principalement à devenir de plus en plus compressible. Contrairement aux index basés sur le hachage, tout le texte est indexé, sans filtre.

Les premières approches d'indexation reposaient surtout sur le hachage. Les tables de hachage offraient une construction simple et rapide et permettaient d'associer aux k-mers diverses informations (abondance, origine, etc.). En revanche, leur empreinte mémoire était élevée. À titre d'exemple, un génome humain contient de l'ordre de 3×10^9 k-mers, stockés sur 8 octets (64 bits), leur stockage « naïf » requiert déjà plus de 24 Go, auxquels s'ajoutent en pratique le surcoût des pointeurs, des métadonnées et du facteur de charge, portant souvent l'empreinte à plusieurs dizaines de gigaoctets. Ces exigences, incompatibles avec les capacités matérielles disponibles il y a encore peu, limitaient l'usage à de petits jeux de données ou à des serveurs très dotés en mémoire, freinant leur adoption.

Ces contraintes ont conduit à l'essor des méthodes d'indexation full-text, mieux adaptées à la compression et capables de stocker un index souvent plus petit que le texte lui-même, en exploitant sa redondance. Elles ont marqué une avancée majeure en mémoire, avec seulement quelques gigaoctets de RAM pour indexer le génome humain (67). Surtout, ces index acceptent des requêtes de longueur arbitraire, ce qui les rend plus polyvalents que les approches de type dictionnaire de k-mers qui imposent un k fixe. Cette flexibilité s'accompagne d'une localisation précise des motifs dans le texte, permettant l'alignement direct des reads sur le génome. Ils se sont révélés particulièrement efficaces pour les reads courts Illumina, courts, nombreux et peu erronés. Des outils comme BWA (74) et Bowtie (68) ont largement popularisé cette approche.

L'arrivée de la troisième génération de séquençage et l'augmentation de la RAM ont mis en évidence les limites des index full-text pour les reads longs. D'une part, la mémoire n'étant plus le principal goulot, et les schémas de mapping par chaînage d'ancres exigeant de très nombreuses requêtes, les approches fondées sur des dictionnaires de k-mers sont revenues au premier plan. Minimap2, en particulier, a montré qu'indexer une fraction seulement des k-mers (via des minimizers, un échantillonnage parcimonieux) permet de mapper puis d'aligner des long reads ou des génomes entiers avec précision et efficacité.

D'autre part, pour des jeux de longs reads au taux d'erreur élevé (jusqu'à 15% selon les technologies et générations), les index full-text perdent de l'intérêt : le bruit réduit la compressibilité du texte tandis qu'indexer des erreurs gaspille de la mémoire. La correction des erreurs de séquençage en amont est possible mais est coûteuse, complexifie la chaîne d'analyse et peut introduire des biais. À l'inverse, les approches k-mers contournent ce problème en n'indexant pas les k-mers rares et en tolérant naturellement substitutions et indels grâce à des mécanismes de vote par recouvrement et à des critères approximatifs (pourcentage de k-mers partagés, similarité de Jaccard, chaînage cohérent). Dans ce contexte, les méthodes de hachage, adaptées pour la TGS, se révèlent plus efficaces et plus robustes au bruit. La hausse générale des capacités mémoire, y compris sur des machines modestes, a levé le principal frein historique à leur adoption.

La section suivante présentera l'état de l'art actuel de l'indexation, en détaillant les deux

grandes familles évoquées ci-dessus, en montrant leurs caractéristiques mais aussi leurs limitations, ce qui nous amènera par la suite au coeur du sujet de cette thèse.

2.1 Recherche exacte de séquence : indexation full-text et BWT

Dans de nombreuses applications, il est indispensable de pouvoir rechercher des séquences au sein de jeux de données potentiellement volumineux. Les index full-text sont une solution à ce besoin en permettant une recherche exacte, sans perte d'information, grâce à une transformation complète du texte d'origine en une structure indexée. C'est le cas récemment de U-index (universal index) (10).

Ils permettent de rechercher efficacement des mots ou des séquences de caractères dans un texte complet, sans se limiter à des mots-clés prédéfinis. Ces structures de données facilitent l'accès rapide à toutes les occurrences d'un motif donné dans le texte. Cette technique est largement utilisée pour les analyses génomiques, notamment la recherche de motifs dans des séquences. Hors de ce contexte, ce type d'indexation est aussi très utilisé dans les moteurs de recherche, afin de cibler des mots-clés entrés par les utilisateurs (123).

Dans ce cadre, il n'y a aucune utilisation de k-mers, le texte dans sa globalité est transformé afin de pouvoir y réaliser des recherches efficaces. Le plus gros point fort de ces méthodes sera la compression, facilitée notamment par un réordonnancement des caractères, qui permet de stocker le texte sous une forme de manière générale plus légère que le texte initial.

Cette section visera à présenter les différentes approches principales. Nous commencerons par présenter des structures simples, non compressées, pour arriver par la suite aux structures compressées full-text.

Arbre des suffixes

Un arbre des suffixes est une structure de données qui représente tous les suffixes d'une chaîne de caractères dans un arbre compacté. Chaque suffixe peut être lu en parcourant les arcs de la racine vers les feuilles. Un arbre des suffixes, dénoté T, est un arbre orienté tel que :

— T a n feuilles, n étant la taille du texte à indexer
— chaque noeud dispose d'au minimum 2 noeuds fils
— chaque arc est étiquetté par une sous-chaîne de T
— le chemin de la racine à une feuille i est dénoté $T[i..n]$

Une illustration de ce type d'arbre est donnée en Figure 2.1.

L'application la plus naturelle est la recherche exacte de sous-chaîne, et donc de motifs. Mais les répétitions peuvent aussi être retrouvées facilement : en effet, chaque noeud ayant au minimum deux fils, on sait que cette chaîne est répétée au moins deux fois.

L'arbre peut être construit en temps linéaire par rapport au texte à indexer ($O(n)$ pour un texte à indexer de taille n), grâce à plusieurs algorithmes comme ceux de Ukkonen (120), McCreight (89), ou Weiner (126) qui utilisent les liens suffixes (liens entre les noeuds internes qui permet la construction linéaire). La recherche d'un motif de longueur m dans l'arbre dépend de la taille de ce motif et est en $O(m)$.

Une extension de l'algorithme peut permettre d'indexer un ensemble de séquences distinctes. On parle d'arbre des suffixes généralisé (17). Dans tous les cas, dans cette extension ou l'algorithme d'origine, il est indispensable de garder en mémoire la séquence d'origine en supplément

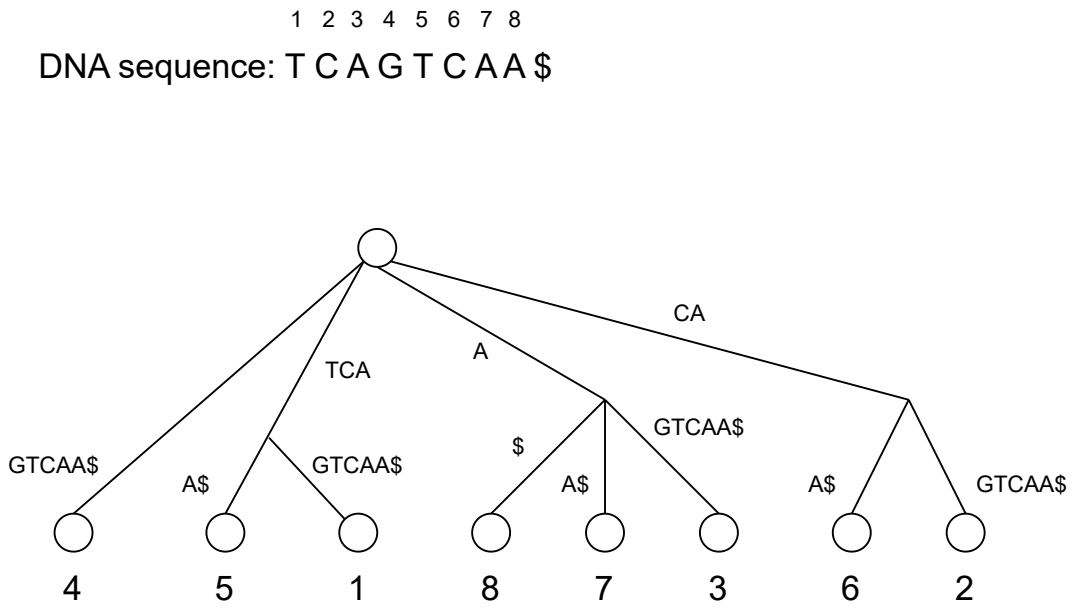


FIGURE 2.1 – *Exemple d'arbre des suffixes, pour la séquence ADN TCAGTCAA. Dans cet exemple, les liens suffixes ne sont pas représentés.*

du suffixe lui-même, ce qui complexifie légèrement la mécanique, tout en gardant une construction en temps linéaire. Cette extension permet notamment l'analyse comparative de séquences, ou encore la recherche de motifs communs.

L'arbre des suffixes reste une structure très lourde en mémoire, notamment pour de longues séquences ou de grands ensembles de séquences. Ce coût en mémoire est notamment dû aux pointeurs nécessaires à la structure d'arbre pour garder les informations de noeuds père/ noeuds fils. Un pointeur étant stocké sur 64 bits, la complexité est de $O(64n)$, n étant la longueur du texte. Bien qu'il peut avoir théoriquement une complexité linéaire en mémoire, en pratique la RAM utilisée peut rapidement exploser. Une solution, engendrant une perte d'informations, est l'arbre des suffixes tronqué (95). Il permet de limiter la taille des suffixes stockés, en fixant une longueur maximale. Dans notre cas, nous limitons généralement à une taille de k-mer fixée. Il peut donc être suffisant pour certaines applications, mais ne permet pas la recherche de motifs longs (de longueur supérieure à k). La construction restant toujours linéaire, c'est la taille du paramètre k qui jouera sur le gain en complexité. Les informations perdues peuvent représenter des erreurs, et donc ne pas supprimer uniquement des éléments informatifs.

Un lien direct entre l'arbre des suffixes tronqué et le graphe de de Bruijn peut être fait (voir Figure 2.2). Le graphe de de Bruijn peut être vu comme un arbre tronqué, ne stockant que les k-mers (sans les suffixes plus petits). Chaque k-mer stocké dans un graphe est un chemin de longueur k dans l'arbre. On peut également voir l'arbre des suffixes généralisé comme une structure colorée, chaque séquence représentant une couleur différente.

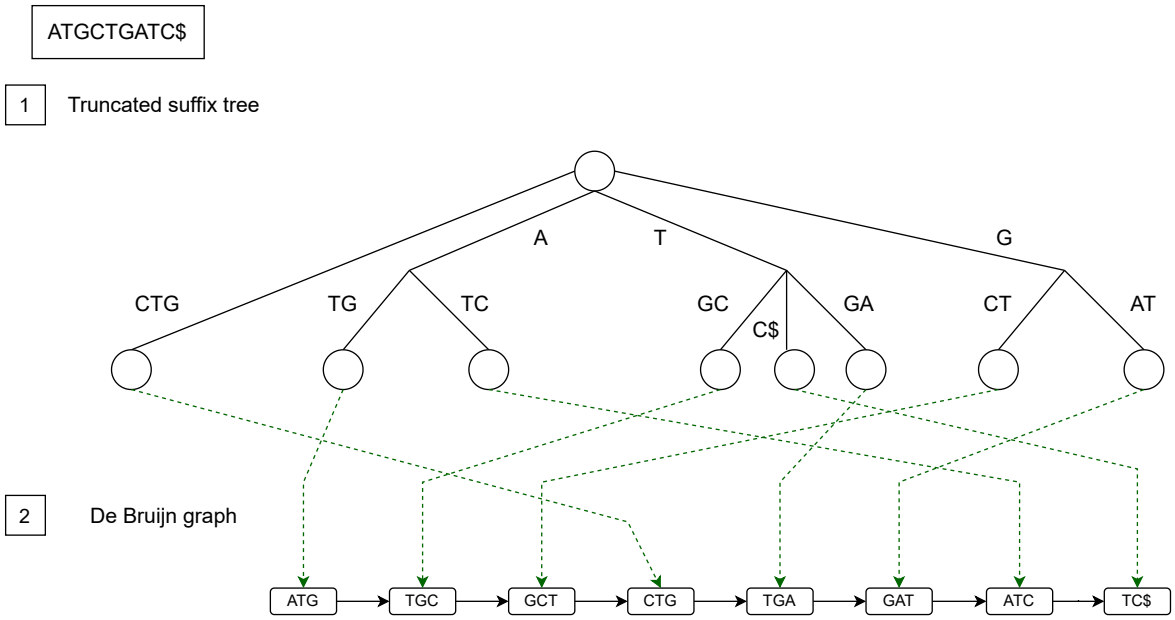


FIGURE 2.2 – *Lien entre arbre des suffixes tronqué et graphe de de Bruijn. Les k-mers de taille 3 représentés sur l'arbre sont directement liés au graphe par les flèches.*

Une autre solution, permettant d'alléger le coût en mémoire tout en restant exacte, consiste à stocker les suffixes tout simplement dans une table.

Table des suffixes

La table des suffixes (83) est une version linéarisée de l'arbre des suffixes. Cette table contient tous les suffixes de la chaîne donnée, triés dans l'ordre lexicographique. C'est une structure beaucoup plus compacte que l'arbre, permettant des recherches de motif en $O(nlog(n))$ et pouvant être construit en temps linéaire. Il est possible de passer de l'arbre au tableau, et inversement (voir Figure 2.3). Pour passer de l'arbre au tableau, un parcours des feuilles tout en gardant en mémoire la position dans la séquence d'origine suffit, tandis que pour l'opération inverse une structure supplémentaire est nécessaire afin de reconstituer les arêtes intermédiaires : le LCP (Longest Common Prefix). Cette opération est linéaire.

Sequence : TCAGTCAA$

Suffix array (SA)

$	9
A$	8
AA$	7
AGTCAA$	3
CAA$	6
CAGTCAA$	2
GTCAA$	4
TCAA$	5
TCAGTCAA$	1

LCP

$	X
A$	0
AA$	1
AGTCAA$	1
CAA$	0
CAGTCAA$	2
GTCAA$	0
TCAA$	0
TCAGTCAA$	3

Suffix tree

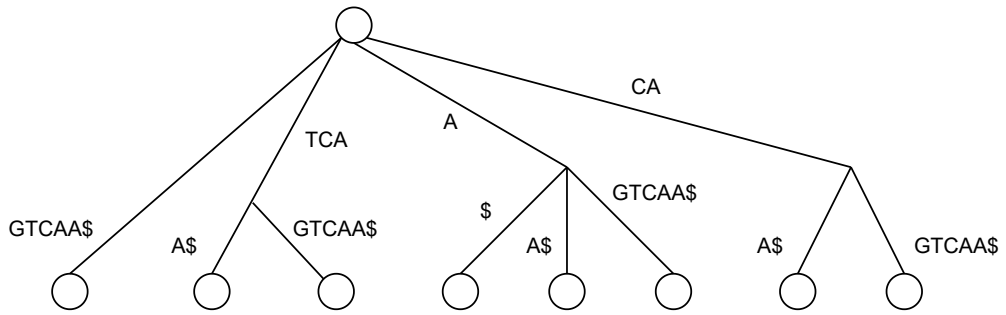


FIGURE 2.3 – *Table des suffixes, LCP et arbre des suffixes. Pour chaque préfixe, la valeur donnée dans la table des LCP correspond à la longueur du préfixe à chercher à la racine de l'arbre. Par exemple, pour le suffixe $CAGTCAA$\$ qui a la valeur 2 en LCP, on cherchera CA à partir de la racine.*

Dans la continuité de cette approche, la transformée de Burrows-Wheeler (BWT) exploite directement l'ordre lexicographique des suffixes pour produire une version permutée du texte original, qui présente des propriétés particulièrement avantageuses en termes de compressibilité et de structuration des données.

Transformée de Burrows-Wheeler

La transformée de Burrows-Wheeler (26) est dans certains cas un prérequis aux méthodes d'indexation full-text.

Cette transformée est une méthode permettant d'optimiser la compression et la recherche dans un texte. Elle ne compresse pas en tant que tel, mais réorganise les caractères de manière à ce que d'autres outils puissent compresser plus efficacement. L'input est le texte à indexer, complété par un unique caractère de fin de chaîne (généralement $), l'output est le texte réorganisé.

Le caractère de fin de texte est indispensable, afin d'obtenir des rotations uniques et identifiables. En utilisant les arbres des suffixes (126), la transformée est calculable en temps linéaire. En partant de la table des suffixes, en prenant à chaque étape la lettre qui précède la position de chaque suffixe trié, n étapes constantes sont nécessaires, où n est la taille du texte. Quatre étapes sont nécessaires, illustrées en Figure 2.4 :

1. Ajout du caractère de fin de texte : *ATGCTAGCT* $\rightarrow$ *ATGCTAGCT$*
2. Création de toutes les rotations circulaires du texte
3. Tri des rotations, par ordre lexicographique
4. Extraction de la dernière colonne

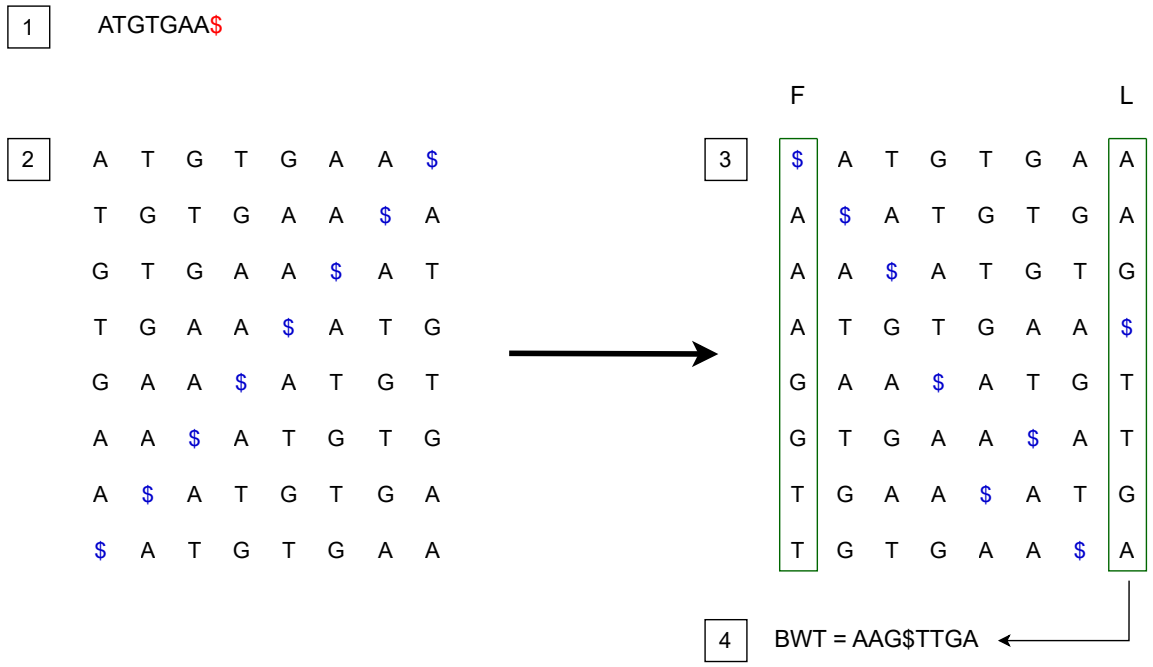


FIGURE 2.4 – *Étapes de la transformée de Burrows-Wheeler. (1) : ajout du caractère de fin de ligne $, (2) création de toutes les rotations circulaires du texte d'origine, (3) tri des rotations par ordre lexicographique, (4) extraction de la dernière colonne, qui est la BWT.*

Le texte d'origine peut être retrouvé uniquement à partir de la dernière colonne de la transformée (voir Figure 2.5), il n'y a donc aucune perte d'information.

Pour validation de méthode, et notamment en preuve d'efficacité de compression, la BWT a été testée en compressant les 14 fichiers les plus utilisés du Calgary Compression Corpus (9, 26). La BWT a permis en moyenne de diviser par 3.5 la taille de fichiers, avec pour certains une réduction de taille par 10, utilisant entre 0.83 et 4.45 bits par caractères. Aussi, là où la table des suffixes est en $O(nlog(n))$, la BWT est de complexité $O(nlog(\sigma))$, où σ représente la taille de l'alphabet.

On peut retrouver à partir de la matrice finale la liste de tous les suffixes du texte d'origine, triée par ordre lexicographique. La liste correspondant à la Figure 2.5 est : [$, A$, AA$, ATGTGAA$, GAA$, GTGAA$, TGAA$, TGTGAA$]. Ce lien entre la matrice de BWT et la table des suffixes est crucial dans la construction de certains index, notamment le **FM-index** (44).

FM-index Cet index est utilisé pour trouver de façon efficace le nombre d'occurrences d'une sous-chaîne dans le texte compressé, ainsi que pour localiser la position de chaque occurrence

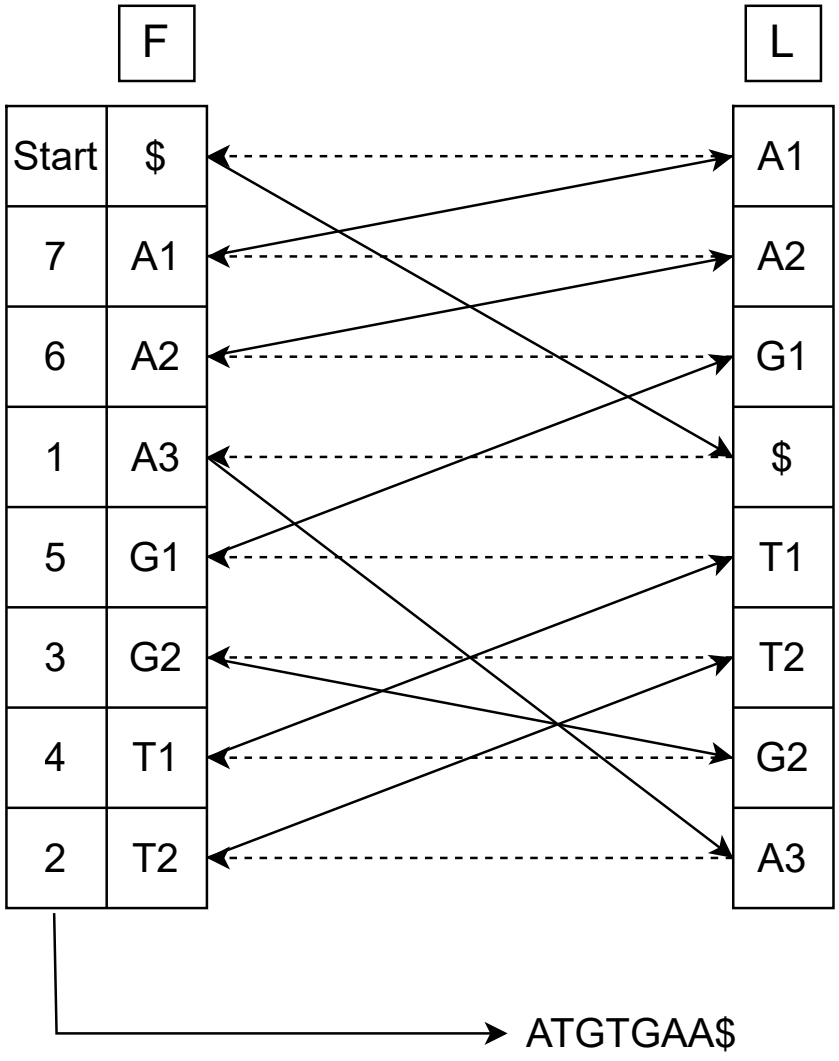


FIGURE 2.5 – *Opération inverse du calcul de BWT, pour retrouver le texte d'origine. À chaque étape, en commençant par le caractère de fin de texte, on recherche le même caractère en colonne Last. L'élément en même position dans la colonne First est le prochain caractère à intégrer au texte. La première colonne, commençant par Start, indique l'ordre dans lequel les caractères sont ajoutés.*

de la chaîne recherchée. En plus de la BWT, deux structures sont nécessaires (illustrées en Tables 2.1 et 2.2 sur l'exemple présenté ci-dessus) :

— Une première appelée C qui compte le nombre de caractères plus petits qu'un caractère c dans le texte. Cette table permettra de retrouver facilement dans la matrice de BWT la position de départ de c dans la première colonne triée.

— Un tableau d'occurrences ($Occ(c, i)$), qui compte le nombre de fois où le caractère c apparaît dans les i premiers caractères de la transformée.

c	$	A	G	T
C[c]	0	1	4	6

TABLE 2.1 – *Table présentant le nombre de caractères plus petits qu'un caractère c, dans le cas de l'exemple de texte T = ATGCTAGCT$*

BWT	A	A	G	$	T	T	G	A
Position	1	2	3	4	5	6	7	8
$	0	0	0	1	1	1	1	1
A	1	2	2	2	2	2	2	3
G	0	0	1	1	1	1	2	2
T	0	0	0	0	1	2	2	2

TABLE 2.2 – *Table présentant le nombre de fois qu'un caractère c apparaît dans les i premiers caractères de la transformée AAG$TTGA.*

Le tableau de correspondance entre la première et la dernière colonne (LF-mapping, ou last-to-first mapping) peut être calculé grâce à ces deux tables avec la formule suivante :

$$\mathrm{LF}(i) = C[\mathrm{L}[i]] + \mathrm{Occ}(\mathrm{L}[i], i)$$

Prenons un exemple, toujours tiré des figures 2.4 :

$$\mathrm{LF}(2) = C[\mathrm{L}[2]] + \mathrm{Occ}(\mathrm{L}[2], 2)$$

$$\mathrm{LF}(2) = C[\mathrm{G}] + \mathrm{Occ}(\mathrm{G}, 2)$$

$$\mathrm{LF}(2) = 4 + 0 = 4$$

On peut déduire que le caractère G en position 2 de la BWT correspond à la 4e position dans la première colonne.

Pour rechercher un motif dans un texte, il suffit de réaliser une recherche inversée en propageant le LF-mapping, afin de trouver une plage correspondante (plage d'indices indiquant où retrouver le préfixe dans la table des suffixes).

Un des principaux avantages des structures telles que le FM-index est la compression. De nombreuses optimisations ont pu être mises en place telles que les wavelet trees (51) (arbre binaire où chaque nœud correspond à une sous-partition de l'alphabet) ou le Run-length Encoding (RLE) (107). Ce dernier permet de compresser des textes ayant des caractères identiques successifs, ce qui est notre cas du fait du tri lexicographique des caractères. Chaque caractère est

associé à son nombre d'occurrences successif. Par exemple, $AAAAACCCTGGG$ serait encodé $A5C3T1G3$.

Plusieurs autres index sont des variantes directes du FM-index. On peut surtout lister le r-index (93), ayant une complexité en espace de $\mathcal{O}(r)$, où r est le RLE associé à la BWT à calculer pour un texte de taille n. Le temps nécessaire passe de $\Omega(n/r)$ (pour le FM-index classique) à $\mathcal{O}(\log(n/r))$. Plus récemment, une extension du r-index, br-index (8) (bi-directional index) a été développé, offrant la possibilité d'extensions bidirectionnelles lors des recherches de pattern.

Movi (133), basé sur le move index (97), a à la fois une complexité de $\mathcal{O}(r)$ en mémoire et $\mathcal{O}(1)$ en temps de requête. Néanmoins, ces structures sont limitées dans la compression pour des données répétées, souvent bruitées qui réduisent la redondance et introduisent de nouvelles séquences non pertinentes dans l'index.

D'autres structures, provenant de l'idée initiale de la BWT, permettent de stocker différentes informations comme plusieurs séquences, à l'image de l'extended Burrows-Wheeler Transform, ou une version tronquée de la BWT à l'image de BOSS (21) et de la spectralBWT (2).

Concernant les index colorés, permettant d'associer les éléments à leur source, VARI (92) associe la représentation BOSS (topologie succincte du graphe de de Bruijn) à une représentation compressée des couleurs. Chaque k-mer y est relié à un bitvector qui indique les échantillons ou sources dans lesquels il est présent. Cependant, cette méthode souffre d'une redondance importante, car les couleurs sont calculées de manière indépendante pour chaque k-mer. Ainsi, plusieurs k-mers peuvent partager le même bitvector, ce qui entraîne un coût mémoire inutile.

Pour résoudre ce problème, Rainbowfish (5) a été proposé. Il observe que de nombreux bitvectors sont identiques, et évite donc de les stocker plusieurs fois. À la place, seuls les bitvectors distincts sont conservés, et chaque k-mer est simplement associé à un identifiant représentant le bitvector correspondant. Cette optimisation permet à Rainbowfish d'être jusqu'à 20 fois plus économe en mémoire. En contrepartie, la construction de l'index est plus lente : jusqu'à trois fois plus de temps est nécessaire pour de très grands jeux de données (par exemple humains), même si cet impact est négligeable sur de petits ensembles, comme ceux de type *E. coli*.

Nous avons vu que les structures basées sur la BWT présentent l'avantage d'être hautement compressibles, ce qui en fait des solutions efficaces en termes d'utilisation mémoire. Cette propriété en a fait un composant central de nombreux index full-text, tels que le FM-index, utilisés notamment pour la recherche rapide de motifs.

Cependant, ces structures présentent certaines limitations. En particulier, le temps de requête peut devenir un facteur limitant, notamment dans le cas de recherches sur de très grands jeux de données ou lorsqu'un accès fréquent à l'index est requis. De plus, leur nature globale, conservant l'intégralité de la séquence d'origine dans une forme transformée, les rend peu adaptées au filtrage d'éléments erronés. En effet, toute l'information est conservée, ce qui empêche un certain contrôle sur le contenu final de l'index. Ce principe reste acceptable sur les génomes entiers, bien assemblés. Les jeux de reads compliquent la tâche, car bien que la redondance puisse être gérée, chaque erreur est conservée, cette méthode devient bien moins pertinente.

À l'inverse, les approches basées sur le découpage des séquences en k-mers offrent une plus grande flexibilité. Elles permettent notamment de sélectionner les k-mers à indexer en fonction de critères définis, comme leur abondance, facilitant ainsi le filtrage des erreurs et l'adaptation de

l'index aux objectifs spécifiques de l'analyse. En pratique, presque la totalité des k-mers erronés peuvent être filtrés et supprimés, tout en gardant l'information génomique.

La section suivante est donc consacrée aux structures de données fondées sur les k-mers. Nous en présenterons les principes, les avantages en termes de performance et de flexibilité, ainsi que les limites potentielles qu'elles peuvent rencontrer selon les cas d'usage.

2.2 Méthodes basées sur les k-mers

Dans ce chapitre, nous considérerons les graphes de de Bruijn comme une forme de table d'association, où chaque k-mer joue le rôle d'une clé associée à diverses informations, telles que son abondance, son appartenance à un ou plusieurs jeux de données (dans le cas des structures colorées), ou encore d'autres métadonnées. Cette analogie est justifiée par le fait que les relations de chevauchement, au cœur de la structure des graphes de de Bruijn, peuvent être déterminées efficacement par un petit nombre de requêtes locales. Deux nœuds sont voisins si le suffixe de longueur $k-1$ de l'un correspond au préfixe de longueur $k-1$ de l'autre. À partir d'un k-mer donné, l'ensemble des k-mers adjacents (successeurs ou prédécesseurs) s'obtient par décalage d'une base, offrant jusqu'à quatre possibilités de chaque côté, calculables efficacement à partir d'un accès clé–valeur.

Cette section sera structurée en fonction de l'information recherchée : nous aborderons dans un premier temps les index indiquant la présence/absence d'une séquence, puis nous détaillerons les index fournissant l'abondance et enfin nous parlerons des index étant capables de donner des positions.

2.2.1 Index de présence

En termes de complexité, ce type d'index est le plus simple à mettre en place. Il se résume souvent à la création d'un ensemble de k-mers, dont on peut ensuite vérifier la présence, afin de confirmer ou non la présence d'une séquence.

Ensembles de k-mers

Un premier type d'index est spécifiquement conçu pour cette tâche : le set de k-mers, qui permet de tester la présence ou l'absence d'un k-mer, sans lui associer d'information supplémentaire.

Une approche naïve pour représenter un ensemble de k-mers consisterait à les stocker dans une liste triée selon l'ordre lexicographique. Cependant, cette méthode présente rapidement des limites en termes de mémoire et de performances. La complexité en espace d'une telle structure est de l'ordre de $O(nk)$, où n est le nombre de k-mers et k leur longueur, puisque chaque k-mer est stocké explicitement sous forme de chaîne ou d'entier codé sur plusieurs octets. Pour un génome humain contenant environ 3×10^9 k-mers, même un encodage compact sur 8 octets par k-mer nécessite déjà près de 24 Go. Les requêtes s'effectuent en $O(\log n)$ à l'aide d'une recherche dichotomique, ce qui reste correct pour de petits ensembles, mais peu efficace pour des jeux de données massifs, d'autant que les insertions et suppressions nécessitent des réorganisations coûteuses de la structure.

Les tables de hachage offrent une alternative plus performante. Chaque k-mer est transformé par une fonction de hachage et inséré à la position correspondante dans une structure (Figure 2.6). L'accès, l'insertion et la suppression s'effectuent en moyenne en temps constant $O(1)$, rendant cette structure très efficace pour de grands ensembles de k-mers. Cependant, cette rapidité a un coût mémoire significatif. En effet, chaque entrée doit stocker à la fois la clé hachée, la valeur associée, et souvent des métadonnées de contrôle (par exemple un drapeau d'occupation ou un pointeur). En pratique, la taille mémoire d'une table de hachage dynamique dépasse largement le volume théorique minimal : selon le facteur de charge α (rapport entre le nombre d'éléments et la taille du tableau), il faut souvent allouer entre 1.5 et 4 fois plus d'espace que la taille totale des données stockées. Ainsi, indexer les k-mers d'un génome humain peut nécessiter plusieurs dizaines de gigaoctets de mémoire vive.

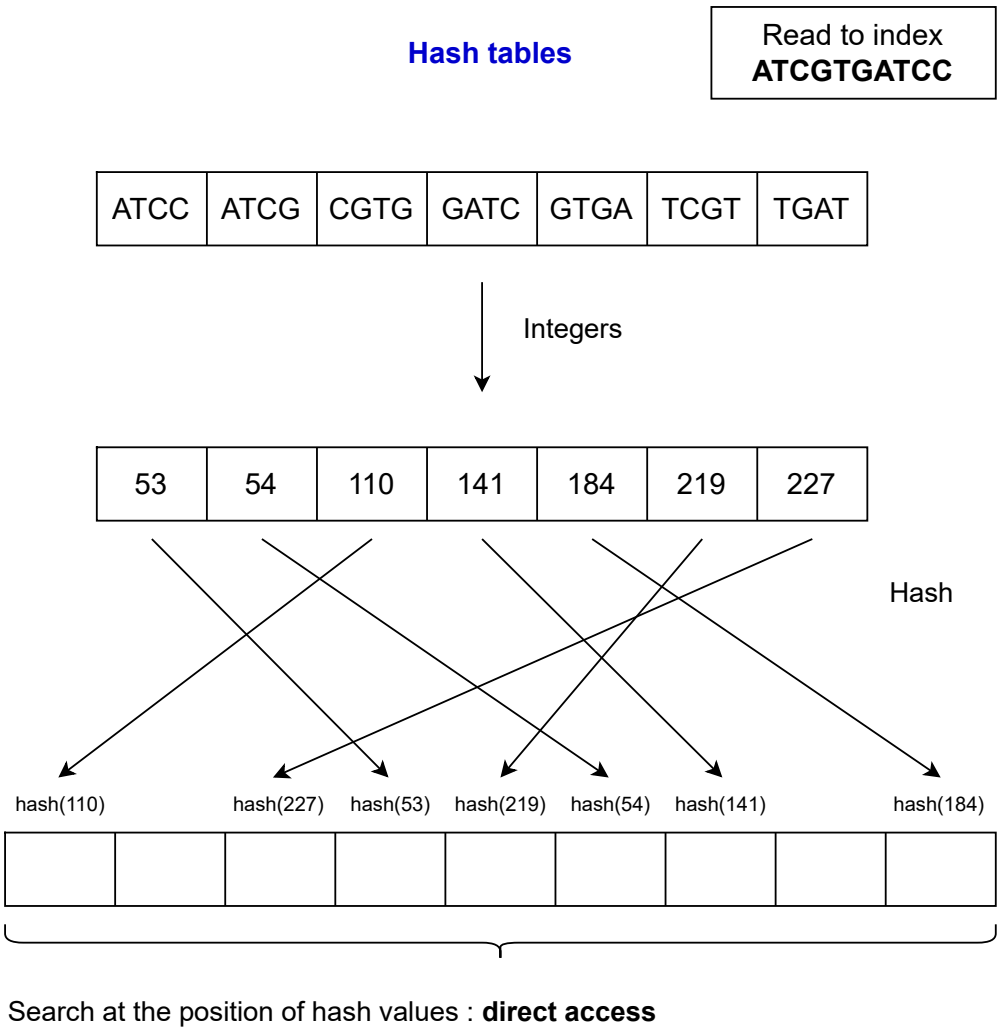


FIGURE 2.6 – *K-mer set représenté par une table de hachage. Les valeurs des k-mers sont hachées et peuvent être retrouvées directement à la position correspondante dans le tableau.*

Suivant la stratégie de hachage utilisée, des collisions peuvent survenir lorsque plusieurs k-mers obtiennent la même valeur de hash. Plusieurs stratégies permettent de les gérer :

— **Chaînage** : chaque position du tableau pointe vers une liste chaînée contenant les k-mers ayant produit le même hash. Cette approche augmente la flexibilité mais dégrade les temps d'accès proportionnellement au nombre de collisions.

— **Adressage ouvert** : en cas de collision, on recherche la prochaine case libre selon une stratégie déterministe (linéaire, quadratique, double hachage, etc.). Cela évite les pointeurs supplémentaires et préserve une bonne localité mémoire, mais la performance dépend fortement du facteur de charge.
— **Hachage parfait minimal** : la fonction de hachage est construite pour associer une clé unique à chaque k-mer, garantissant l'absence de collisions. Cette solution minimise l'espace et offre des requêtes en temps constant, mais elle impose un ensemble statique de k-mers (aucune insertion ou suppression après construction).

Ainsi, bien que les tables de hachage constituent une solution rapide et conceptuellement simple, leur consommation mémoire importante reste un obstacle majeur pour les très grands jeux de données biologiques.

Divers index sont directement basés sur une ou plusieurs tables de hachage, à l'exemple de Pufferfish (6), BLight (86) ou encore Bifrost (54).

Ces trois exemples utilisent, afin de limiter au maximum l'utilisation mémoire, une fonction de hachage minimale parfaite (MPHF) ou un dérivé, en l'occurrence BBHash (78) pour Pufferfish et Blight. Une fonction de hachage parfaite est une fonction qui associe les clés à insérer à un ensemble d'entiers, sans aucune collision. Il s'agit d'une structure statique, qui est adaptée à un ensemble de clés fixe. Lorsqu'elle est minimale, les entiers contenus dans l'ensemble sont consécutifs (généralement de 0 à $n-1$, où n est le nombre de clés), et la table qui les contient a une taille égale à n. On peut observer les différences entre les fonctions de hachage sur la Figure 2.7.

Une propriété intéressante de ces approches est qu'il n'est pas nécessaire de stocker explicitement les k-mers : la fonction de hachage permet de les représenter implicitement par leurs positions dans la table. Cela réduit la mémoire utilisée. En revanche, comme la MPHF ne reconnaît que les clés qui ont servi à sa construction, il est nécessaire de détecter et filtrer les k-mers qui ne faisaient pas partie de l'ensemble initial. C'est pour cette raison qu'un filtre est utilisé au préalable de l'accès à la table, afin de considérer le k-mer courant ou non en fonction de sa présence.

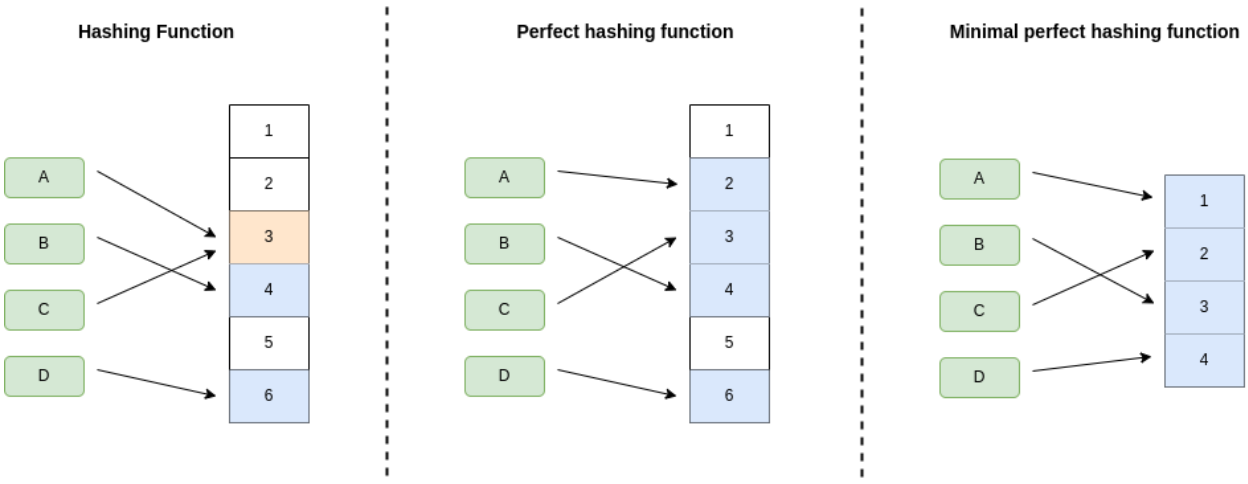


FIGURE 2.7 – *Différences entre la fonction de hachage classique, la fonction de hachage parfaite et la fonction de hachage minimale parfaite.*

Enfin, une dernière manière courante de hacher consiste à associer à chaque k-mer sa plus petite sous-chaîne de longueur m (parmi ses $k-m+1$ sous-chaînes), selon un ordre total fixé.

Cette sous-chaîne s'appelle un minimizer. Plusieurs k-mers distincts peuvent partager le même minimizer, ce qui réduit la mémoire (on ne stocke que $\leq 4^m$ clefs possibles), mais introduit des collisions. Cette fonction n'est pas une fonction de hachage cryptographique : elle n'est ni uniforme, ni résistante aux collisions, ni à sens unique. Elle relève au contraire du hachage sensible à la similarité (LSH) : des k-mers proches ont une probabilité élevée de partager le même minimizer.

Au-delà de leur rôle en assemblage, les graphes de de Bruijn présentés plus haut peuvent être vus comme une représentation d'un ensemble de k-mers. En compactant le graphe, on obtient des chemins dont les séquences, souvent appelées *tigs (par exemple unitigs), offrent une description textuelle compacte de cet ensemble.

Ces séquences constituent un SPSS (109) (Spectrum-Preserving String Set) du jeu de k-mers considéré : un ensemble de chaînes dont le spectre en k-mers est exactement celui du jeu initial, sans k-mer supplémentaire ni manquant. Un SPSS sert donc de substitut textuel au set de k-mers, dans le but de minimiser l'espace de leur représentation.

Tigs Nous présentons ici quatre variantes de graphes de de Bruijn, appelées ici tigs : les unitigs, simplitigs, matchtigs et eulertigs. La figure 2.8 illustre les deux premières catégories.

La forme la plus commune est celle des **unitigs**. Leur but principal est de compacter le graphe en réduisant le nombre de nœuds, tout en évitant de construire des séquences chimériques (non vues dans le jeu original). Un unitig correspond à un chemin maximal non ambigu dans le graphe, c'est-à-dire une suite de k-mers contigus telle que chaque nœud, à l'exception des extrémités, possède exactement un prédécesseur et un successeur. Cette stratégie de compaction présente plusieurs avantages, notamment en matière de mémoire et de détection des erreurs, qui apparaissent souvent sous forme de branches courtes ou de bulles facilement filtrables. Malgré cette réduction, une redondance persiste aux extrémités des unitigs, correspondant aux préfixes et suffixes de longueur $k-1$ reliant les unitigs entre eux. Pour pallier ce problème, la notion de simplitig (24) a été introduite.

L'idée fondamentale est que les unitigs peuvent eux-mêmes être compactés afin de produire des séquences plus longues en minimisant la redondance des préfixes/suffixes. Le principe de cette approche est de réaliser des compactions sans se soucier de la production de séquences chimériques : lorsque plusieurs prolongements sont possibles, au lieu de s'arrêter comme le ferait l'algorithme de construction d'unitigs, on effectue une compaction dont l'optimalité n'est pas garantie. Cette approche est de nature gloutonne : à chaque nœud visité, le chemin est prolongé aussi loin que possible, sans rétro-analyse, même si plusieurs prolongements sont envisageables. Ce processus est répété jusqu'à ce que tous les nœuds soient couverts.

Pour réduire encore la taille de la SPSS, les eulertigs cherchent à trouver un parcours minimal couvrant tous les k-mers en s'appuyant sur des cycles eulériens. Un exemple est présenté dans la figure 2.9. Enfin, les matchtigs généralisent encore cette idée en autorisant la réutilisation d'un k-mer dans plusieurs chemins, ce qui permet d'obtenir une compression plus efficace, au prix d'une redondance qui peut, selon le cas, gêner ou non les analyses en aval.

Au-delà, les masked superstrings autorisent des compactions avec des chevauchements de longueur strictement inférieure à $k-1$. Cet assouplissement de la contrainte peut créer des k-mers "aliens" absents du jeu initial. Il est alors nécessaire de les marquer explicitement via une structure auxiliaire (par exemple un bitset positionnel aligné sur les chaînes, une table associant les intervalles invalides, ou un index des k-mers aliens) afin de préserver des requêtes exactes sur le spectre d'origine et d'éviter toute contamination lors des analyses.

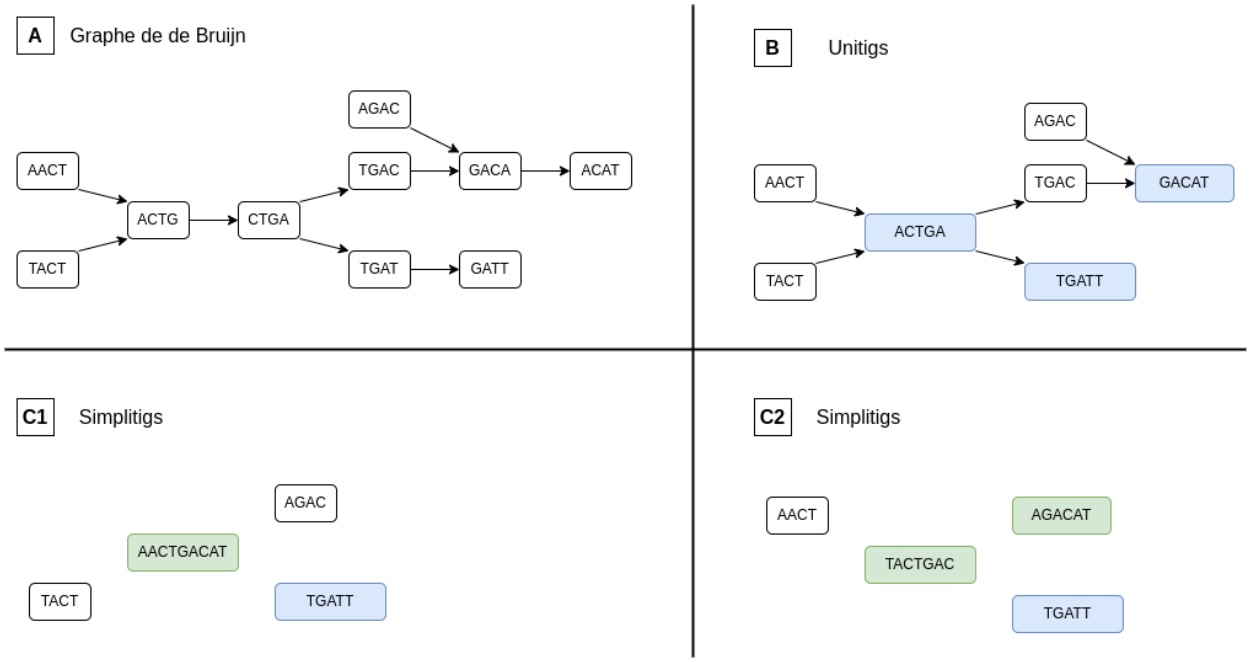


FIGURE 2.8 – *Exemple d'unitigs (B) et de simplitigs (C1 et C2) à partir d'un graphe de de Bruijn (A). Les unitigs sont créés en regroupant les k-mers successifs dont le chemin n'a pas d'ambiguïté. Ici 3 regroupements apparaissent pour former 3 unitigs dont la longueur est supérieure à k (affichés en bleu). Deux possibilités de simplitigs sont montrés ici, avec en vert les éléments rassemblés.*

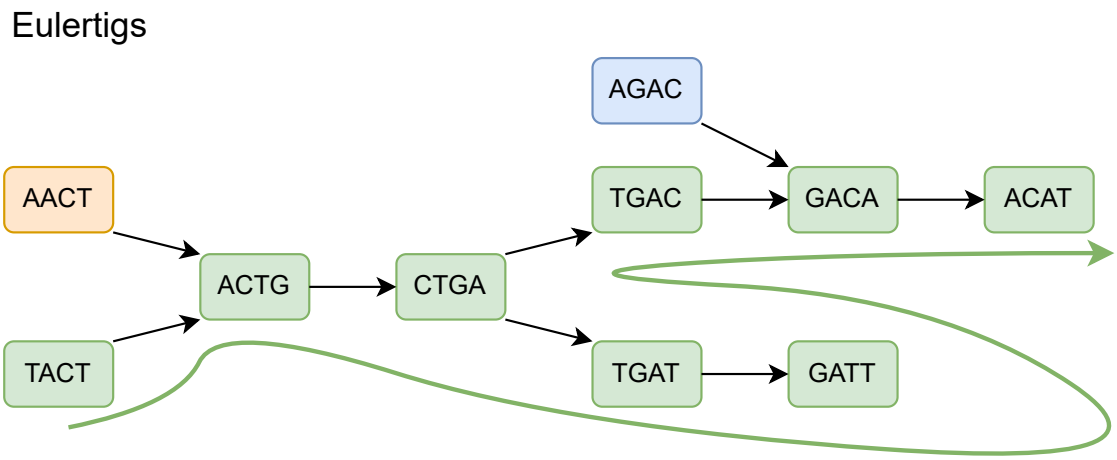


FIGURE 2.9 – *Exemple d'eulertig (flèche verte) à partir du graphe de de Bruijn donné.*

Les tigs sont utilisés dans divers types d'index, notamment dans le cadre de la compression de graphes. BCALM et sa version finale BCALM2 (29) permettent de compacter efficacement un graphe de de Bruijn en produisant un ensemble d'unitigs maximaux. La méthode repose sur une phase de compaction en plusieurs étapes : les k-mers sont d'abord répartis en groupes (buckets) selon leurs minimizers respectifs, chaque bucket est ensuite compacté localement en formant des unitigs, qui sont finalement étendus entre buckets pour obtenir des unitigs maximaux. Pour un objectif similaire, Cuttlefish (62) calcule les unitigs maximaux sans nécessiter le stockage explicite du graphe de de Bruijn. Cette méthode repose sur l'hypothèse que les k-mers terminaux des séquences en entrée correspondent également aux extrémités des unitigs, ce qui permet d'en simplifier la construction tout en limitant les besoins en mémoire intermédiaire. GGCAT (33) poursuit cette logique en proposant une méthode efficace et adaptée à de très grands ensembles de données. Ces index sont exacts donc ne produisent pas de faux positifs lors des requêtes.

Les structures présentées ci-dessus réduisent fortement l'espace, passant d'environ $2k$ bits par k-mer à quelques bits par k-mer pour une SPSS. Elles sont cependant les plus efficaces quand le graphe est proche d'une chaîne, avec peu de variations et de branchements. Plus le graphe est ramifié, plus il subsiste de chevauchements et plus la SPSS s'allonge. De plus, leur construction est globale et statique, ce qui ne convient pas à tous les usages.

Dans ce contexte, les structures probabilistes, en particulier les filtres de Bloom, offrent une alternative très économe en mémoire et rapide. Un filtre de Bloom est un tableau de m bits initialisés à 0. On choisit f fonctions de hachage indépendantes. Pour insérer un k-mer, on calcule ses f hachages et on met à 1 les bits aux positions obtenues. Pour une requête, on recalcule les f positions et on répond présent si tous les bits sont à 1. Le coût d'insertion et de requête est $O(f)$ en temps, l'espace du filtre étant m bits. Le filtre ne stocke pas les clés, seulement des bits partagés, ce qui explique sa compacité.

La structure est approximative : elle ne produit jamais de faux négatifs, mais peut produire des faux positifs lorsque des clés différentes activent les mêmes bits (voir Figure 2.10). La probabilité de faux positif vaut

$$p \approx \left(1 - e^{-fn/m}\right)^f$$

pour n éléments insérés, m bits et f hachages. Le choix optimal du nombre de hachages est

$$f^\star = \frac{m}{n} \ln 2,$$

auquel cas

$$p \approx \left(0.6185\right)^{m/n}.$$

Ainsi, pour une cible p fixée, le budget en bits par k-mer est $m/n \approx \log_{0.6185}(p)$, souvent inférieur à 10 bits par k-mer en pratique. Les collisions ne sont pas «résolues» individuellement ; on contrôle leur fréquence en augmentant m ou en ajustant f. Des variantes (partitioned/blocked Bloom, filtres en cascade) améliorent la localité mémoire et les performances cache.

Les filtres peuvent être combinés à d'autres structures pour obtenir des comportements exacts sur le graphe. Minia (31) assemble des contigs en utilisant un filtre de Bloom et stocke séparément les faux positifs critiques (cFP) qui impacteraient la topologie, rendant les requêtes sur les arêtes exactes. Dans ces deux cas, la mémoire reste très basse : Minia assemble plus de 2.7 milliards de k-mers d'un génome humain complet avec 5.7 Go de RAM, pour un temps de l'ordre de 23 heures.

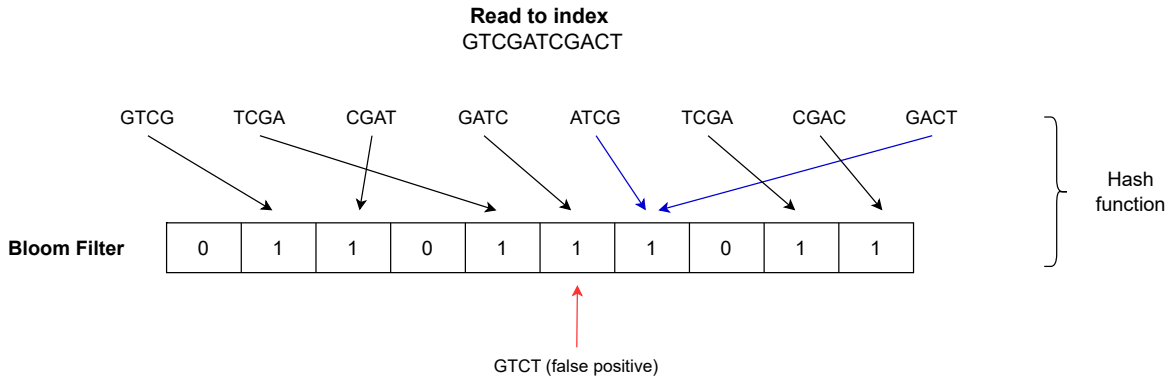


FIGURE 2.10 – *Exemple de read dont les 8 k-mers sont insérés dans un filtre de Bloom. Un k-mer supplémentaire, GTCT, n'appartenant pas au read, est toutefois déclaré présent : c'est un faux positif.*

Les structures présentées ci-dessus ne stockent qu'un unique jeu de données. Or, dans de nombreuses applications (pangenomique, métagénomique, transcriptomique), on souhaite en indexer plusieurs simultanément afin d'identifier la provenance d'une séquence ou de comparer des profils. Pour cela, les structures colorées étendent l'index en associant à chaque élément un ensemble de couleurs représentant les jeux de données où il apparaît.

k-mers colorés

Nous cherchons à retrouver l'origine de chaque k-mer, c'est-à-dire l'ensemble des séquences (génomes, reads ou jeux de données) dans lesquelles il apparaît. On associe donc à chaque k-mer un sous-ensemble de jeux de données, souvent représenté par un ensemble de couleurs.

Un moyen direct consiste à stocker l'information d'origine dans une table de hachage : la clé est le k-mer, la valeur est une structure codant l'ensemble des jeux de données où il est présent. On parle alors de graphe de de Bruijn coloré. Le choix du codage des couleurs dépend du nombre de jeux de données D et de la parcimonie des présences :

— vecteur binaire dense de longueur D (accès et opérations logiques très rapides, coût $\approx D$ bits par k-mer) ;
— liste triée d'identifiants de jeux de données pour les ensembles clairsemés (coût $\approx \bar{c} \log_2 D$ bits par k-mer, où $\bar{c}$ est la cardinalité moyenne) ;
— bitmaps compressés (par exemple Roaring) ou encodages succincts de bitsets (RRR, Elias–Fano), qui offrent un compromis entre compacité et opérations en temps presque constant.

Les requêtes de présence par jeu de données, d'énumération des couleurs, et les opérations ensemblistes (union, intersection) se traduisent en accès $O(1)$ amorti au conteneur du k-mer puis en opérations sur la représentation des couleurs. Le coût mémoire total s'approxime par

$$\text{Espace} \;\approx\; \text{index des } k\text{-mers} \;+\; \sum_{i=1}^{n} \text{coût}(\text{couleurs}(k_i)),$$

avec n le nombre de k-mers.

Mantis (101) illustre une conception exacte et compacte fondée sur le Counting Quotient Filter (CQF). Chaque k-mer est inséré dans le CQF qui renvoie un identifiant de classe de couleur. Les ensembles de couleurs identiques sont factorisés dans une table partagée de classes (color-class table) et référencés par ces identifiants. Cette factorisation réduit fortement la redondance lorsque de nombreux k-mers coexistent dans des combinaisons de jeux de données récurrentes.

Dans Mantis, la valeur associée à un *k*-mer n'est donc pas un bitset complet mais un pointeur vers la classe, ce qui améliore la localité mémoire. À l'échelle, Mantis est plus compact que les SSBT et accélère sensiblement les requêtes, tout en restant exact. La figure 2.11 illustre le principe : le CQF indexe les *k*-mers et la table des classes stocke les vecteurs de bits couleurs.

Bifrost (54) vise un objectif similaire sur un graphe compacté et coloré, avec une dimension dynamique. L'index structure les unitigs et maintient, pour chacun, un conteneur de couleurs compressé. Des opérations d'ajout et de mise à jour de jeux de données sont supportées sans reconstruction complète, ce qui est utile pour des scénarios évolutifs. En pratique, Bifrost combine une indexation rapide des *k*-mers par minimizers avec des bitmaps compressés par unitig. Les requêtes s'effectuent en deux temps : localisation du *k*-mer dans l'unitig, puis interrogation du conteneur de couleurs. Le coût mémoire par *k*-mer dépend alors de la longueur des unitigs et de la compressibilité des ensembles de couleurs.

Fulgor (43) prolonge cette logique en combinant deux concepts : les SPSS pour regrouper les *k*-mers contigus et les MPHF pour indexer ces chaînes. Inspiré du paradigme introduit par Blight (86), Fulgor stocke les SPSS issues d'un graphe de de Bruijn compacté et assigne un identifiant unique à chaque *k*-mer par MPHF. L'accès aux couleurs est alors indirect : l'identifiant obtenu via la MPHF sert d'index dans une structure parallèle contenant les couleurs. Cette approche présente deux avantages majeurs :

1. **Mémoire minimale.** Les MPHFs garantissent un adressage parfait sans collision et sans pointeurs, réduisant l'espace à quelques bits par *k*-mer.
2. **Accès direct et compressé.** Les SPSS réduisent le nombre d'entrées uniques, et la table de couleurs, compressée par classes, permet des requêtes rapides tout en restant exacte.

Le résultat est un index coloré extrêmement compact, scalable et exact, adapté à des collections pangénomiques de très grande taille.

Themisto (3) est un index de *k*-mers colorés à grande échelle fondé sur la Spectral BWT (variante de BOSS) pour l'accès succinct aux *k*-mers et sur une structure de couleurs factorisée via des *k*-mers clefs. Les ensembles de couleurs sont encodés selon leur densité (listes triées, bitmaps, Roaring), ce qui réduit fortement la redondance.

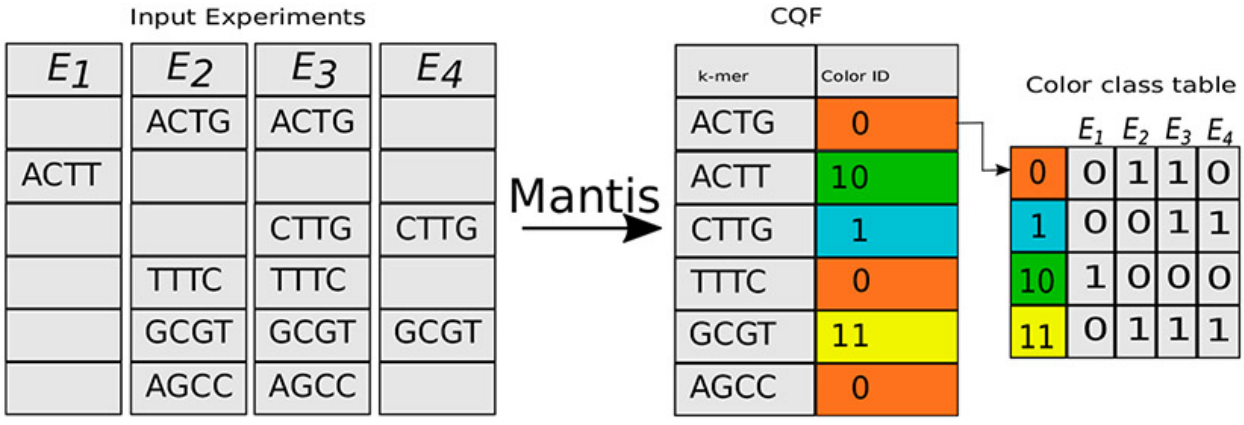


FIGURE 2.11 – *Schéma représentant la structure de Mantis (101). À gauche sont représentés les 4 éléments (E1, E2, E3, E4) à indexer. À droite, la structure finale associant à chaque k-mer un identifiant de couleur, lui-même pointant vers une liste de bits, indiquant la présence ou non dans un élément.*

Une autre approche pour représenter les graphes de de Bruijn colorés consiste à employer des structures probabilistes, en particulier des filtres de Bloom, pour réduire le coût mémoire. L'idée la plus simple consiste à construire un filtre de Bloom par jeu de données. Chaque filtre

encode la présence ou l'absence des k-mers d'un dataset donné. Le coût mémoire total est alors proportionnel au nombre de jeux de données, chaque filtre nécessitant un bitset de taille adaptée à la précision souhaitée. Cette approche reste compacte, mais pour déterminer les couleurs (présences) d'un k-mer, il faut interroger chaque filtre indépendamment, soit un coût en $O(N)$ pour N jeux de données, ce qui devient prohibitif à grande échelle.

Matrice de filtres de Bloom Pour éviter ce coût linéaire, une solution consiste à regrouper tous les filtres dans une **matrice de filtres de Bloom** (22). Dans cette structure, chaque colonne correspond à une position de bit dans les filtres (c'est-à-dire à une valeur de hachage donnée). Au lieu d'interroger chaque filtre séparément, on interroge directement la colonne correspondant à la position du bit associée au k-mer recherché. Ainsi, en accédant à f positions (où f est le nombre de fonctions de hachage), on obtient immédiatement la présence du k-mer dans l'ensemble des jeux de données, avec seulement $O(f)$ accès mémoire, contre $O(fD)$ pour une approche naïve. Ce principe, illustré dans BIGSI (22), permet d'indexer efficacement plusieurs millions de génomes microbiens tout en conservant un temps de requête constant. BIGSI a notamment été utilisé pour la détection de gènes de résistance aux antibiotiques à l'échelle mondiale, démontrant la capacité de cette approche à traiter des collections massives.

Sequence Bloom Trees (SBT) Une autre structure issue du même paradigme est celle des Sequence Bloom Trees (SBT) (117). Chaque nœud de l'arbre contient un filtre de Bloom représentant l'union des k-mers de ses descendants. Lors d'une requête, les k-mers sont testés successivement depuis la racine : si un k-mer est absent du filtre d'un nœud, toute la sous-arborescence peut être ignorée. À l'inverse, si le k-mer est présent, la recherche continue dans les noeuds enfants jusqu'à atteindre les feuilles, correspondant aux jeux de données individuels. Ce processus hiérarchique réduit le nombre de filtres à interroger et limite les accès mémoire (Figure 2.12).

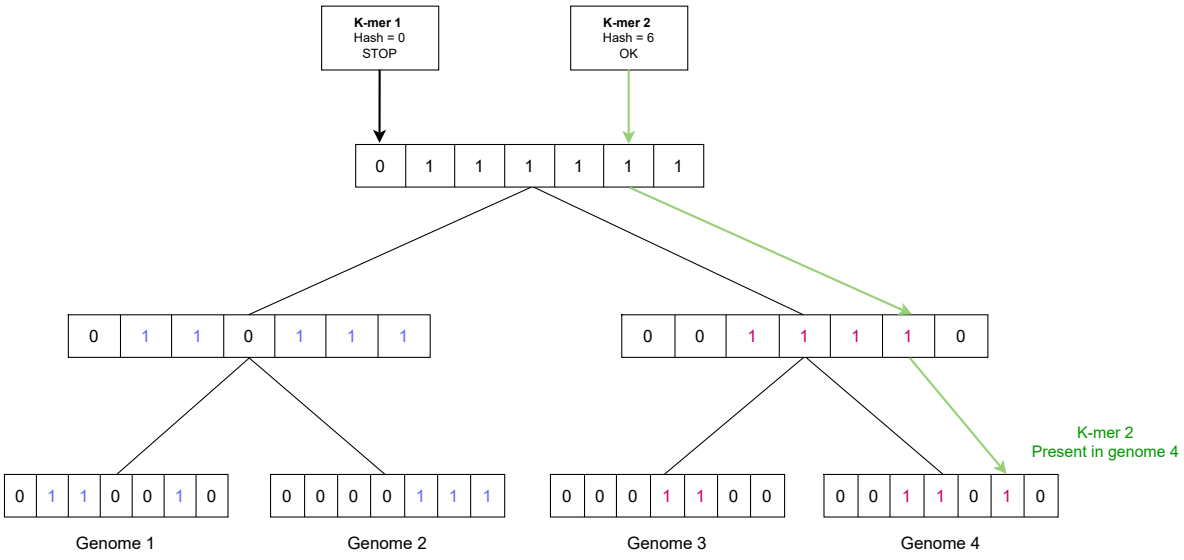


FIGURE 2.12 – *Exemple simplifié de SBT pour quatre génomes. Dans le premier cas de requête (k-mer 1), sa valeur de hachage n'est pas trouvée à la racine, le processus s'arrête. Dans le second cas (k-mer 2), la valeur est retrouvée jusqu'à la feuille correspondant au génome 4, indiquant une probabilité élevée de présence.*

Extensions et variantes. Plusieurs outils reposent sur ce principe, avec des optimisations diverses. Le **SSBT** (*Split Sequence Bloom Trees*) (118) subdivise les filtres pour mieux gérer des bases atteignant le téraoctet, tandis que **HowDe-SBT** (52) réduit l'utilisation de la mémoire et du temps

de calcul respectivement de 39% et 36%. **SeqOthello** (132) adopte un paradigme proche mais remplace les filtres de Bloom par une fonction de hachage minimale parfaite (**Othello**) (131), garantissant un accès en temps constant. Comme Othello est coûteux en mémoire pour des ensembles RNA-seq contenant des milliards de *k*-mers, SeqOthello répartit les *k*-mers en **buckets** indépendants, permettant une construction parallélisée et une meilleure gestion mémoire.

Dans des applications telles que la recherche de variants ou la correction d'erreurs, nous avons montré que l'estimation de l'abondance des *k*-mers joue un rôle central. Des index spécifiques ont été développés dans ce but et seront abordés dans la section suivante.

2.2.2 Index d'abondance

L'intégration de l'information d'abondance dans les structures d'indexation repose sur l'idée d'étendre les représentations de présence/absence à des comptages discrets. Le **counting Bloom filter** (Figure 2.13) en constitue la version la plus directe : chaque case du filtre stocke un entier représentant le nombre d'occurrences d'un *k*-mer. Cette structure permet de quantifier sans stocker explicitement les séquences, au prix d'un léger surcoût mémoire et d'un risque accru de collisions. Une alternative plus efficace est le **Counting Quotient Filter (CQF)** (102), qui réduit les collisions tout en accélérant les insertions et requêtes, surpassant les Bloom filters classiques par un facteur pouvant atteindre quatre.

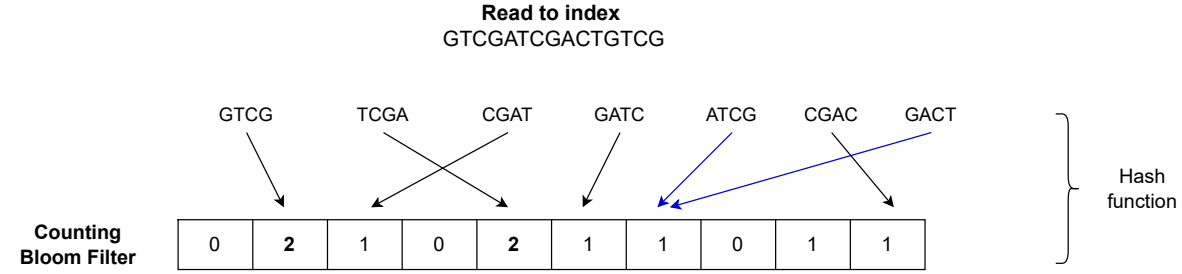


FIGURE 2.13 – *Exemple d'un read dont les 7 k-mers sont insérés dans un counting Bloom filter. Chaque cellule contient un entier représentant le nombre d'occurrences du k-mer.*

Applications et limites En génomique, les abondances de *k*-mers sont souvent homogènes, reflétant la couverture de séquençage. Leur estimation n'apporte donc que peu d'information additionnelle. En revanche, en **métagénomique** et **transcriptomique**, l'abondance des *k*-mers traduit respectivement la composition relative des espèces et l'expression différentielle des gènes. L'indexation d'abondances devient alors indispensable pour estimer proportions, profils ou régulations à partir de grands ensembles de données.

REINDEER et REINDEER2 REINDEER (85) a introduit la première structure d'index coloré exacte, associant à chaque *k*-mer leur abondance exacte. Les *k*-mers sont regroupés par monotig (unitigs dont les *k*-mers ont la même abondance), dont l'abondance moyenne est stockée à l'aide d'une MPHF. REINDEER2 (84) prolonge cette approche avec un modèle hybride, évolutif et paramétrable. Pour cette version, les abondances sont discrétisées. Il combine des filtres de Bloom par niveaux d'abondance et un index inversé partitionné, assurant des insertions dynamiques, un contrôle de la précision par quantification logarithmique, et un traitement différencié des *k*-mers fréquents. Cette conception permet des index de plusieurs milliers de jeux de données tout en conservant une corrélation inférieure à 1% d'erreur avec des quantificateurs de référence comme Kallisto.

Needle (36) adopte lui une approche de sous-échantillonage. Plutôt que d'indexer tous les *k*-mers, il ne conserve qu'un sous-ensemble sélectionné par un schéma d'échantillonnage (minimizers). Chaque *k*-mer indexé est associé à une estimation approximative de son abondance, permettant des requêtes rapides et peu coûteuses en mémoire basées, comme REINDEER2, sur une matrice de filtre. Cette approximation convient aux analyses globales, mais manque de finesse pour les études nécessitant une précision base par base, comme la localisation de variants.

Metagraph (42) étend le paradigme des graphes de de Bruijn colorés à la quantification. Il utilise un graphe compacté enrichi d'une table d'abondance, couplé à des bitmaps compressés pour représenter la présence et la multiplicité des *k*-mers. Metagraph privilégie la scalabilité : il peut indexer des bases pangénomiques à l'échelle du téraoctet grâce à une factorisation hiérarchique des chemins, au prix d'une forte consommation de ressources en construction et d'une indexation statique.

2.2.3 Index de position

Les index full-text (FM-index, BWT) offrent une localisation directe et précise des occurrences, restituant les positions exactes grâce à des mécanismes d'échantillonnage efficaces. À l'opposé, les index fondés sur les *k*-mers se sont principalement concentrés sur la présence, l'abondance ou le chaînage d'ancres, sans prise directe sur la localisation fine. Retrouver cette information nécessite alors des structures complémentaires telles que des listes d'occurrences (*k*-mers $\rightarrow$ reads) ou des stratégies d'échantillonnage, ce qui en fait aujourd'hui un axe d'optimisation majeur.

Dans le contexte des longs reads, cette question devient centrale. La "position" ne se limite plus à un simple offset dans un fichier FASTA, souvent dénué de sens biologique, mais correspond à l'identifiant précis du read porteur. Ces informations sont essentielles pour la détection de variations locales, la comparaison d'échantillons ou encore l'assemblage de novo.

Ainsi, l'association directe *k*-mers $\rightarrow$ reads s'impose comme une primitive clé des futurs index génomiques. Elle dépasse la simple présence ou abondance, tout en éliminant la dépendance à une référence. En redonnant accès aux reads d'origine depuis l'index lui-même, elle ouvre la voie à des approches plus flexibles, marquant une étape décisive vers une indexation beaucoup plus adaptée aux contraintes des longs reads.

C'est dans ce contexte qu'est née l'idée et le développement de **K2R** (122), la contribution majeure de cette thèse. Il s'agit d'un index *k*-mers-to-reads principalement conçu pour les données issues des technologies de séquençage de troisième génération. Les sections suivantes en présenteront les objectifs, les différentes étapes de conception, les difficultés rencontrées ainsi que les améliorations successives ayant conduit à sa version finale, avant d'en exposer les résultats.

Chapitre 3

K2R

L'idée d'un index *k*-mers → reads et son développement constituent l'apport principal de cette thèse. Historiquement, les index *k*-mers → reads étaient efficaces pour les reads courts, où la longueur de *k*-mer est comparable à celle du read, limitant la perte d'information. Avec les longs reads, cette perte relative augmente et a conduit à délaisser ces structures dans un premier temps. Toutefois, l'association explicite des *k*-mers à leurs reads restaure l'accès direct aux séquences sources, permet de localiser des motifs sans référence, facilite la détection de variations locales et fournit un levier utile pour l'assemblage. Dans ce contexte, la réévaluation et l'adaptation de ces index aux spécificités des longs reads s'avèrent pertinentes et constituent le fil conducteur de notre approche. Notre objectif est de concevoir un index qui passe à l'échelle sur de grands jeux de données (forte profondeur sur de grands genomes, cohortes), tout en restant adapté aux longs reads, pouvant atteindre plusieurs centaines de kilobases avec un taux d'erreur sensiblement plus élevé que les reads courts (jusqu'à 15%). L'index doit supporter des requêtes massives, conserver une empreinte mémoire maîtrisée et offrir des temps d'accès presque constants.

Le passage à l'échelle reste néanmoins l'obstacle le plus important, l'enjeu est donc de concevoir une structure aussi légère que possible, tout en maintenant une haute qualité, notamment en garantissant des requêtes exactes (aucun faux positif), localisant des séquences dans un jeu de données de reads. Plus précisément, on veut savoir dans quel(s) read(s) apparaissent des *k*-mers d'intérêt. Par la suite, localiser ces *k*-mers dans les reads devient très peu coûteux. K2R a donc une granularité intermédiaire, comprise entre la simple présence/absence et la position exacte dans une séquence, qui peut être trouvée facilement une fois l'index construit. Nous avons décidé de nommer notre type d'index *Graphe de de Bruijn teinté*, offrant visuellement une plus grande diversité de couleurs qu'un graphe de de Bruijn classique, car chaque couleur correspond pour nous à un read, au lieu d'un document.

Son principe consiste à comparer les *k*-mers des séquences d'intérêt, afin d'identifier les reads partageant une proportion significative de *k*-mers avec la séquence recherchée.

Nous avons donc, pour réaliser ce type de requête de manière efficace et optimisée en termes de performances, choisi d'associer chaque *k*-mer différent du jeu de données aux identifiants de reads dans lesquels il apparaît. Ainsi, le principe des requêtes consiste à découper la séquence recherchée en *k*-mers, de vérifier leur présence dans l'index construit au préalable, et de calculer pour finir le taux de *k*-mers communs. Ce principe est illustré en Figure 3.1, où l'on peut voir que malgré une substitution et une insertion dans une séquence relativement courte, deux tiers des *k*-mers restent communs.

Le fonctionnement général de K2R est quant à lui montré en Figure 3.2.

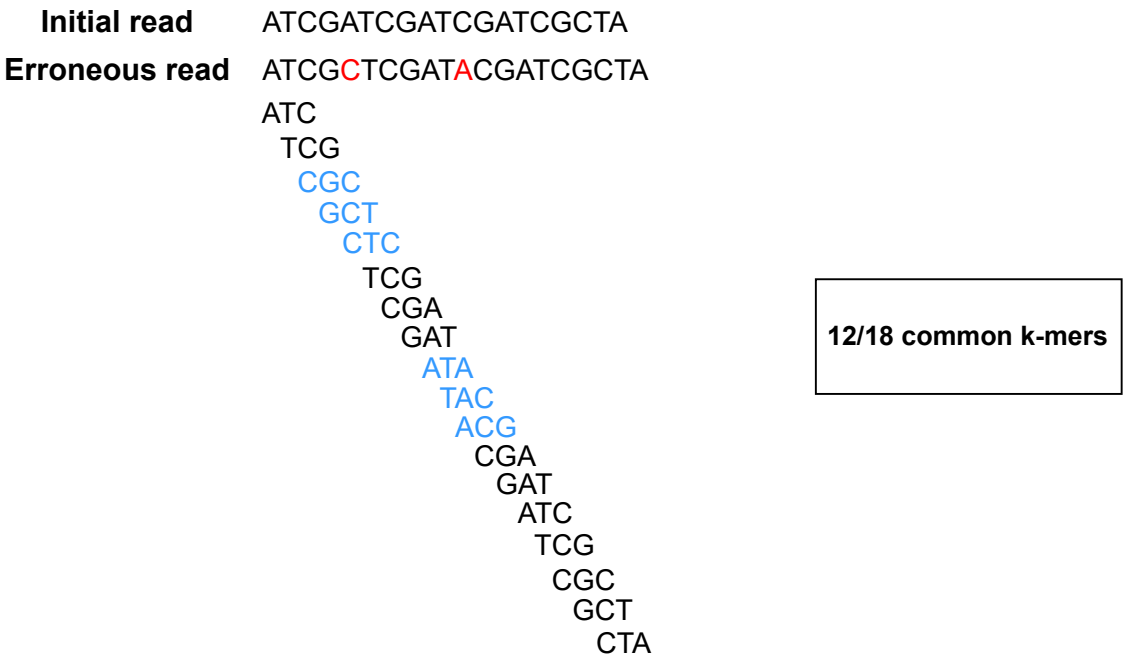


FIGURE 3.1 – *Illustration de la similarité entre séquences, par rapport aux k-mers communs. On observe ici que pour une séquence relativement courte (19 bases), l'ajout d'une substitution et d'une insertion n'a d'impact que sur un tiers des k-mers. Les deux séquences auront un taux de k-mers communs de 2/3.*

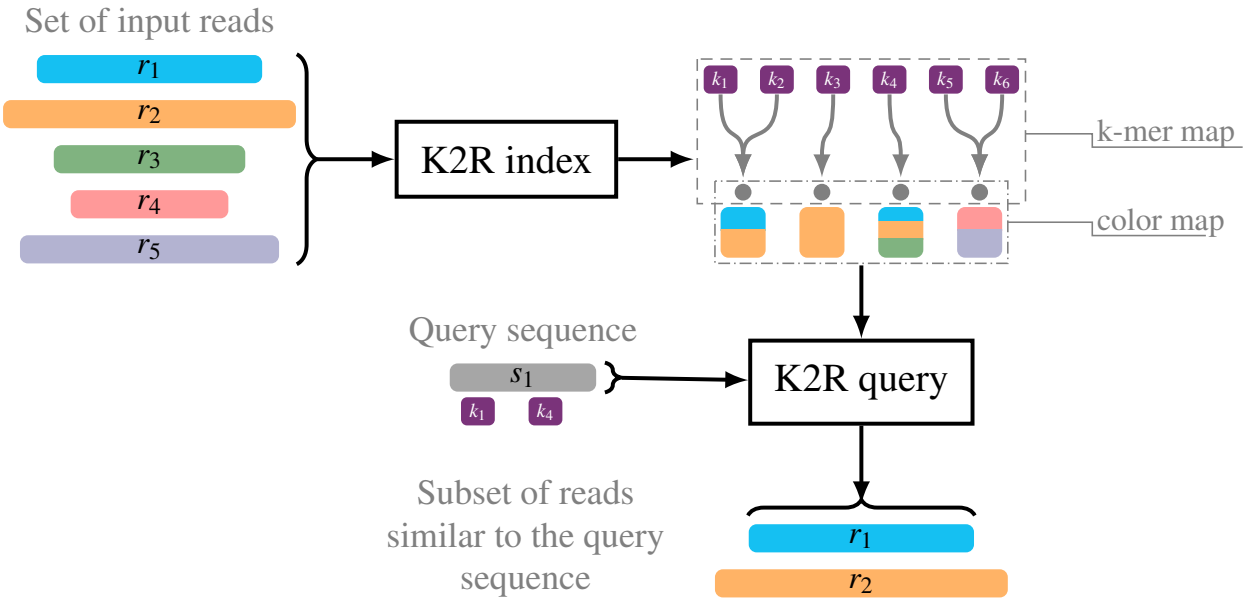


FIGURE 3.2 – *Principe général de l'idée de K2R. On cherche à identifier les reads dans lesquels la séquence s_1 apparaît (à un pourcentage de k-mers communs donné). Le jeu de reads (r1 à r5) est indexé, ici sont représentés 6 k-mers (k-mer map, de k1 à k6). Ceux-ci sont associés aux identifiants de reads dans lesquels ils apparaissent (ici des couleurs, color map). La séquence est elle-même découpée en k-mers, ici deux appartiennent à l'index (k_1 et k_4), et apparaissent tous les deux dans les reads r_1 et r_2.*

Pour atteindre ce résultat, plusieurs idées de structures se sont succédées et des optimisations ont été nécessaires. La section suivante (section 3.1) les expliquera de manière détaillée, avec pour chaque optimisation une présentation de la problématique, la méthode utilisée ainsi que des données chiffrées appuyant l'efficacité de ce qui a été mis en place.

3.1 Étapes de développement et méthodes

Les caractéristiques initiales que nous avions fixées pour notre index étaient les suivantes :

— Exploitation du hachage pour construire l'index.
— Association de chaque *k*-mer aux reads dans lesquels il apparaît. Cet ensemble de reads sera ici considéré comme une couleur.
— Maintien d'une structure légère en mémoire et simple à construire.
— Objectif de requête : recherche de reads contenant des séquences d'intérêt en identifiant les *k*-mers qu'ils partagent avec la requête.
— Garantie de requêtes exactes et d'un débit élevé lors de celles-ci.
— Gestion des erreurs de séquençage grâce au filtrage.

Nous prendrons comme point de départ pour remplir ces conditions la méthode la plus évidente, en montrant par la suite par quels moyens nous avons su l'améliorer.

3.1.1 Version initiale : table de hachage

Nous avons en premier lieu stocké les *k*-mers ainsi que les identifiants de reads associés dans une table de hachage, structure la plus intuitive pour ce type d'utilisation puisque clés et valeurs y sont directement liés (Figure 3.3). Les *k*-mers y sont représentés sous forme d'entiers.

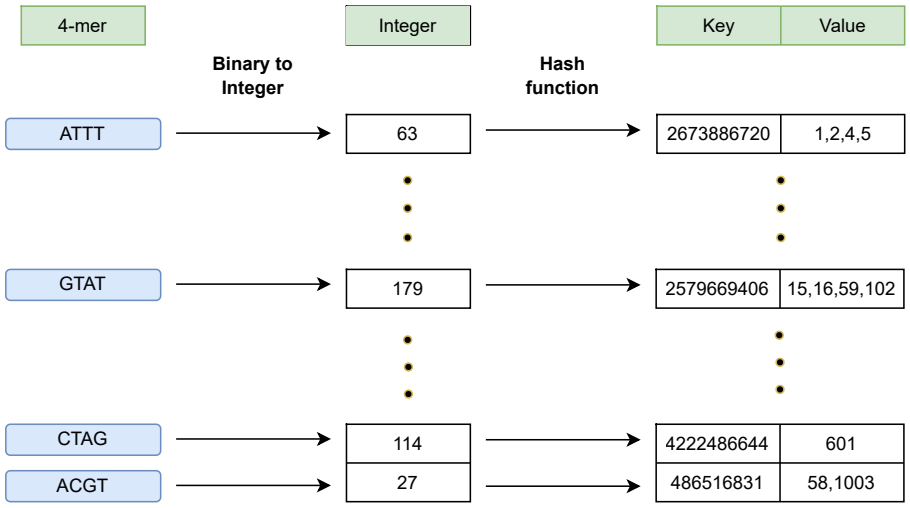


FIGURE 3.3 – *Table de hachage, première version de K2R. Chaque k-mer est haché puis stocké avec la liste des identifiants de reads correspondants.*

Nous avons concrètement avec cette structure un entier (64 bits) pour chaque *k*-mer, et un vecteur d'entiers (32 bits) pour les identifiants de reads. Ces représentations imposent des limites, qui resteront dans la structure finale de K2R : un *k*-mer représenté sur 64 bits devrait être de longueur inférieure ou égale à 32 (2 bits par base). Dans le cas de *k*-mers plus grands, son hachage sur 64 bits est possible, mais peut provoquer des collisions. Leur nombre dépend du nombre de *k*-mers différents, mais la possibilité que deux *k*-mers aient le même hash resterait en pratique faible même en considerant le paradoxe des anniversaires, on estime a 21.6% la probabilité d'une collision entre 3 milliard d'entier 64bits. Dans le cas où les collisions seraient un problème, l'utilisation d'entiers 128bits peut régler en pratique le régler. Sur le même principe, le nombre de reads doit être inférieur ou égal à 2^{32} ou utiliser de plus grands entiers.

À titre d'exemple et de confirmation, nous avons stocké un jeu de reads de longueur 10.000 paires de bases, générés à partir du génome d'*E. coli*, ayant 1% d'erreur et une profondeur de 10X. La taille de la table finale est de 538 Mo, dont 384 Mo pour la partie valeur, qui concerne les couleurs. Lorsque la profondeur est augmentée à 30X, cette fois-ci la taille totale passe à 1.295 Go, dont 926 Mo de couleurs (voir Figure 3.4). La taille du génome d'*E. coli* étant de 4.6 Mbp et celle du génome humain de 3.2 Gbp, on peut s'attendre à ce que l'indexation de plus gros génomes soit techniquement impossible via cette méthode naïve.

Cette solution s'avère donc très coûteuse en ressources, car la taille des vecteurs atteindra souvent environ la valeur de profondeur. Nous avons donc recherché une combinaison de structures, capable de stocker l'information sans avoir besoin d'un nombre si important de grands vecteurs, et en réduisant aussi le coût de stockage des *k*-mers.

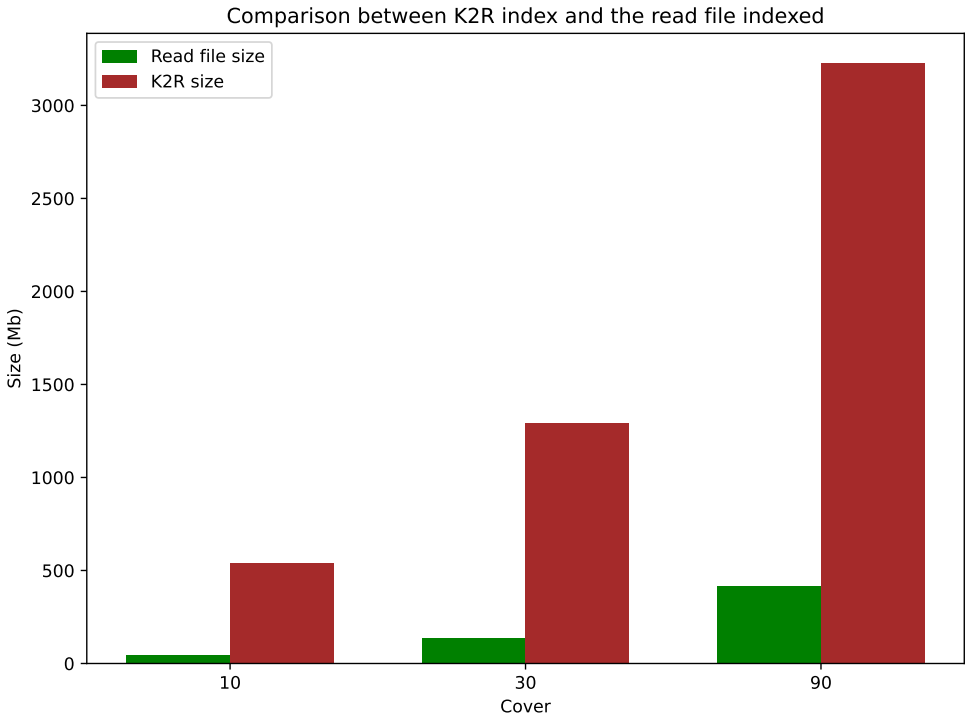


FIGURE 3.4 – *Comparaison entre la taille d'un fichier de reads générés à partir du génome d'E. coli, de longueur 10.000, ayant un taux d'erreur de 1% et la taille finale de l'index créé à partir de ce fichier, grâce à une table de hachage. La taille de l'index généré par cette version préliminaire de K2R est d'environ un ordre de grandeur supérieur à la taille du fichier de reads.*

3.1.2 Filtre de Bloom

Une structure efficace pour stocker des *k*-mers avec un impact mémoire limité est le filtre de Bloom. Chaque *k*-mer est haché par f fonctions et active f bits dans un tableau de m bits, la requête teste ces mêmes f positions. Le filtre ne stocke pas les clés, seulement des bits partagés : pas de faux négatifs, mais des faux positifs avec une probabilité de :

$$p \approx \left(1 - e^{-fn/m}\right)^f,$$

où n est le nombre d'éléments. Le choix optimal $f^\star = (m/n)\ln 2$ donne $p \approx (0.6185)^{m/n}$, soit un budget typique de quelques bits par *k*-mer pour un p visé. Comme m est fixé, les cases à 0 occupent de l'espace, il faut donc dimensionner m en amont pour équilibrer empreinte mémoire et taux de faux positifs. Dans notre cas, nous nous concentrons sur le cas particulier à une seule

fonction de hachage ($f = 1$) pour accélérer les insertions et les requêtes en flux. Chaque k-mer active un bit dans un tableau de m bits, la requête teste ce même bit. Ce choix simplifie le calcul, mais augmente le taux de faux positifs. Pour n éléments insérés, la probabilité qu'un bit soit positionné à 1 est

$$p_1 = 1 - \left(1 - \tfrac{1}{m}\right)^n \approx 1 - e^{-n/m},$$

et donc la probabilité de faux positif correspond à $p_{\text{fp}} = p_1$. Autrement dit, la probabilité qu'un élément non inséré soit déclaré présent est égale à la fraction de bits déjà activés. Ce taux croît rapidement avec n/m.

Notre second objectif était aussi de limiter le nombre de vecteurs à stocker. En effet, chaque vecteur à stocker alloue de la mémoire pour être stocké, et des pointeurs sont également nécessaires pour y accéder. Nous avons donc regroupé tous les identifiants de reads dans un unique vecteur, où les valeurs correspondant à un k-mer spécifique seraient stockées de manière séquentielle. La principale difficulté ici est de pouvoir retrouver les identifiants correspondant à un k-mer précis, étant donné qu'ils ne sont pas "séparés" dans le vecteur unique. Une façon de récupérer ces valeurs est d'avoir préalablement enregistré la position indiquant où effectuer la recherche dans le vecteur. Pour ce faire, nous avons donc une structure composée de quatre éléments, comme illustré dans la Figure 3.5 :

— Figure 3.5 (A) : D'un filtre de Bloom avec une fonction de hachage permettant de vérifier l'occurrence d'un k-mer. Si le bit est à 1 à la position du hash du k-mer, alors celui-ci est probablement présent.
— Figure 3.5 (B) : D'un counting Bloom filter donnant le nombre d'occurrences de chaque k-mer. Un counting Bloom filter va, au lieu d'indiquer la présence/absence d'un élément sur un bit, stocker son nombre d'occurrences. Concrètement, c'est un vecteur de taille fixe d'entiers, où pour chaque élément une fonction de hash va désigner l'indice du vecteur à incrémenter. Il est nécessaire à la création du vecteur suivant.
— Figure 3.5 (C) : D'un vecteur de positions, indiquant pour chaque k-mer où aller chercher les identifiants lui étant assignés dans le vecteur suivant. Il est créé en additionnant chaque nombre d'occurrences du counting bloom filter.
— Figure 3.5 (D) : D'un vecteur unique stockant tous les identifiants de reads.

Nous avons fait initialement le choix de réaliser un benchmark sur cette version en le comparant tout d'abord à Short read connector (SRC) (87) et au r-index (93), pour la construction des index ainsi que les requêtes. Short read connector (SRC) est une méthode utilisant une MPHF, à l'origine développé pour les reads courts. Le r-index, quant à lui, est basé sur la transformée de Burrows-Wheeler, et utilise le run-length encoding (RLE : répétitions de plusieurs bases identiques) à des fins de compression. Le taux d'erreurs des jeux de données, simulés à partir des génomes d'*E. coli*, *Mycoplasma* et *S. cerevisiae* varient entre 0.1% et 10% pour une profondeur de 50X. Les résultats sont présentés dans les Figures 3.7 et 3.8.

Bien que cette version provisoire de K2R implique un coût en mémoire et en temps plus élevé que celui du r-index pour de faibles taux d'erreur, en raison notamment de l'utilisation de plusieurs structures de données et de deux parcours complets du fichier d'entrée, l'approche s'avère déjà compétitive pour des taux d'erreur plus élevés, tant en phase de construction qu'en phase de requête.

La version précédente de K2R, appliquée au même jeu de données *E. coli* avec un taux d'erreur de 1%, nécessitait plus de 6Go de mémoire RAM pour l'indexation, contre un peu plus de

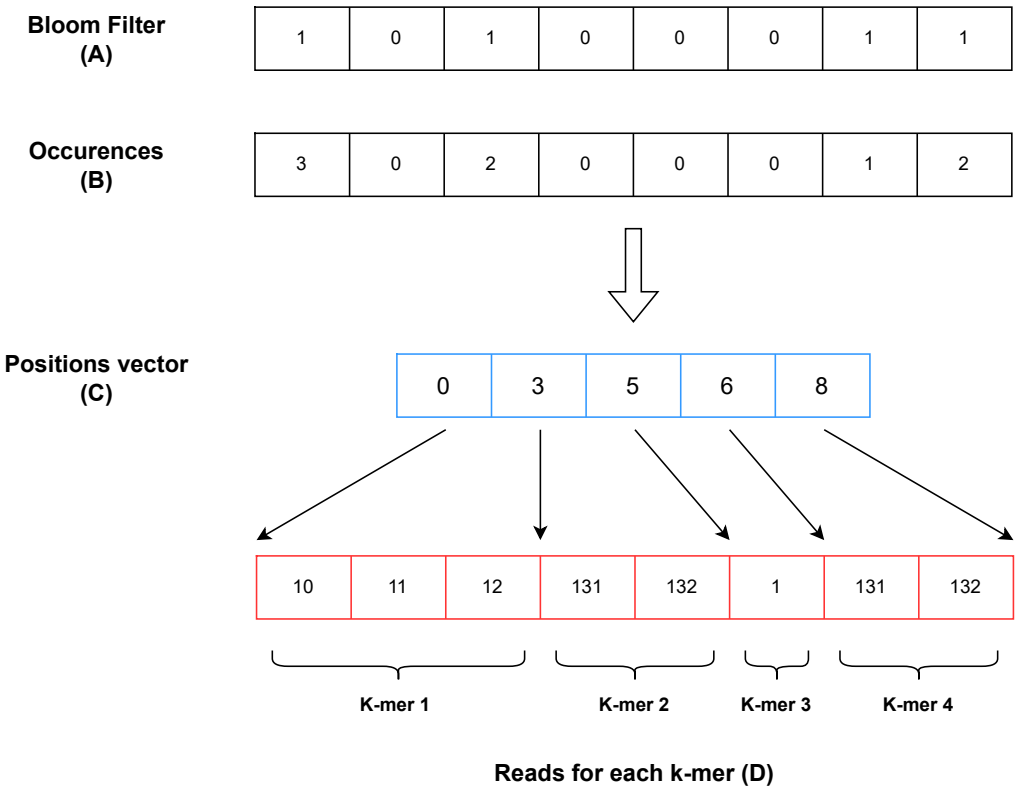


FIGURE 3.5 – *Filtre de Bloom (A) et Counting Bloom Filter (B) utilisés pour créer l'index. Le vecteur (C) est généré à partir de B en sommant les éléments. Une fois C créé, B peut être supprimé afin d'économiser de la mémoire. Le vecteur (D) contient les identifiants des reads.*

2Go dans la version actuelle, soit une réduction d'environ un facteur trois. Cette amélioration est encore plus marquée pour un taux d'erreur de 10%, où l'ancienne version nécessitait plus de 20Go de RAM, alors que la nouvelle n'utilise qu'environ 2Go. Cette différence est expliquée par l'utilisation du filtre de Bloom, plus léger que la table pour les *k*-mers. Quant au temps, il est légèrement plus élevé pour cette version-ci (environ deux fois supérieur), dû également à l'étape de création des filtres qui nécessite un parcours de fichier supplémentaire. On a donc un compromis temps/mémoire permettant d'envisager l'indexation de plus gros jeux de données.

L'analyse plus précise de l'utilisation mémoire de K2R au cours de l'exécution révèle que l'index lui-même (représenté par le vecteur de reads) constitue logiquement la principale source d'utilisation mémoire. Dans le cas de l'indexation d'un fichier de reads issus du génome d'*E. coli* et de profondeur 10X, on retrouve 176Mo pour le vecteur de reads, contre 39Mo pour celui de positions. Pour une profondeur de 90X le ratio reste à peu près le même, avec 1.58Go d'identifiants contre 150Mo de positions. Cela suggère que la mise en oeuvre de techniques de compression sur cette structure pourrait constituer un apport significatif. En revanche, en ce qui concerne les temps de requête, K2R surpasse d'ores et déjà les solutions de référence prises en compte dans cette étude.

Sur le même principe que la Figure 3.4 de la version précédente, nous avons voulu connaître la différence de taille entre la structure créée et le jeu de reads indexé (Figure3.6). La différence est légèrement moindre, l'index étant environ 4 fois plus lourd que le jeu de données de reads.

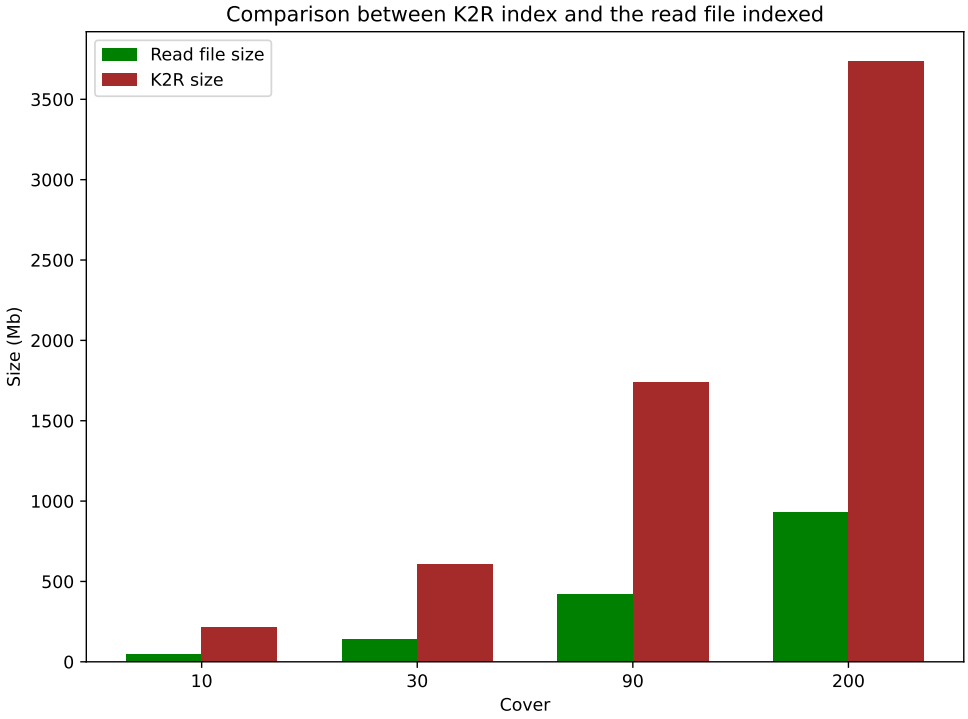


FIGURE 3.6 – *Comparaison entre la taille d'un fichier de reads générés à partir du génome d'E. coli, de longueur 10.000 et ayant un taux d'erreur de 1%, et la taille finale de l'index créé à partir de ce fichier, en utilisant des filtres de Bloom (version préliminaire de K2R). Plusieurs profondeurs sont présentées ici, allant de 10X à 200X. La taille de la structure d'indexation est environ 4 fois plus lourde que le jeu de reads.*

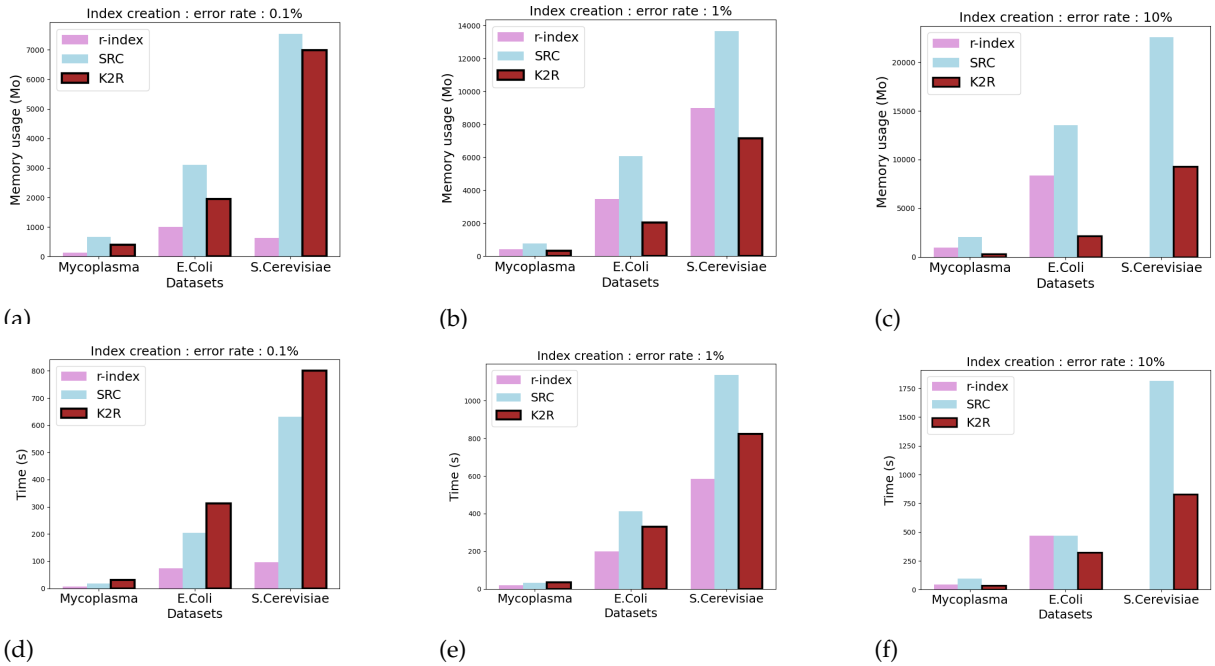


FIGURE 3.7 – *Résultats obtenus en termes de création d'index (sous-figures a,b et c pour la mémoire RAM et d,e et f pour le temps). Le temps CPU (durée pendant laquelle le processeur est effectivement occupé à exécuter un programme) est donné pour chaque outil et différents taux d'erreurs (de 0.1% à 10%). K2R est d'ores et déjà compétitif, tant en temps qu'en mémoire, notamment dans le cas de taux d'erreur importants, en étant plus efficace que SRC comme du r-index (jusqu'à 5 fois plus rapide en temps). Les taux d'erreur faibles (0.1%) restent cependant bloquants, K2R étant presque un ordre de grandeur plus lent que le r-index dans certains cas.*

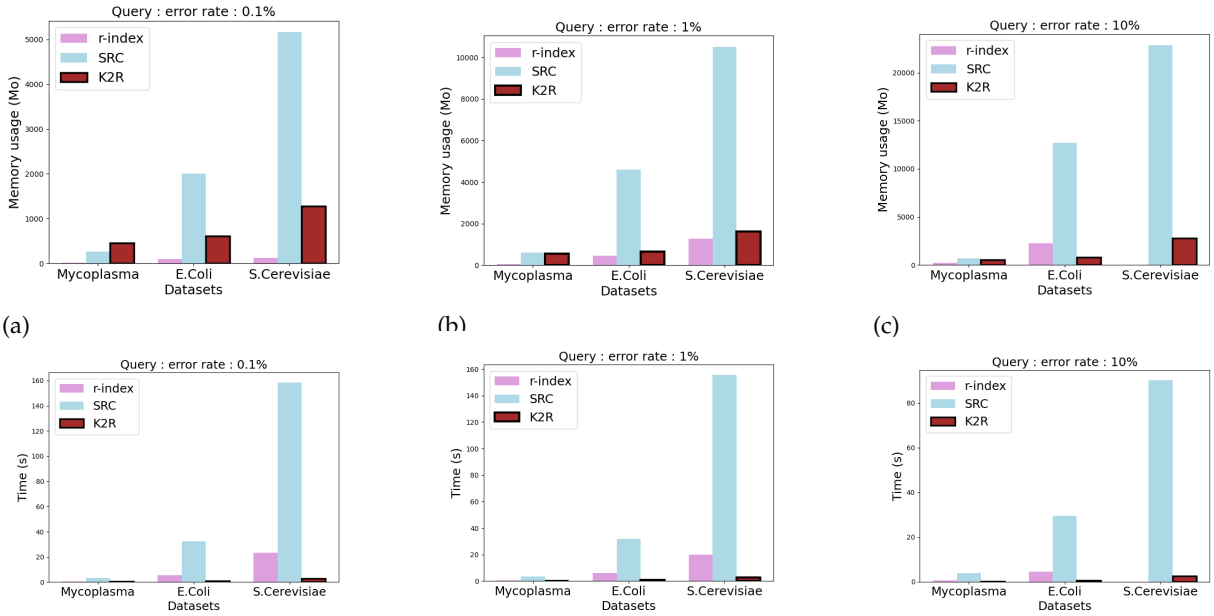


FIGURE 3.8 – *Résultat obtenus en termes de requête (sous-figures a,b et c pour la mémoire RAM et d,e et f pour le temps). Le temps CPU est donné pour chaque outils et différents taux d'erreurs (de 0.1% à 10%). À l'image de la construction de l'index, K2R surpasse SRC et le r-index pour un fort taux d'erreurs (10%). Pour un taux de 0.1%, K2R reste plus coûteux que le r-index.*

La priorité à ce stade était donc de réduire l'impact mémoire. Ici nous considérions le vecteur d'identifiants comme l'ensemble des couleurs du jeu de données. Deux idées principales ressortent : avoir la possibilité de compresser ces couleurs, mais aussi d'éviter les redondances. En effet, plusieurs k-mers, notamment dans le cas de reads chevauchants, peuvent être associés aux mêmes identifiants et donc à la même couleur.

3.1.3 Couleurs

Au cours des versions précédentes de K2R, nous avons constaté qu'un grand nombre de k-mers partageaient la même couleur, c'est-à-dire exactement le même ensemble d'identifiants de reads. Cette redondance découle de la profondeur de séquençage : plus la profondeur est élevée, plus les reads se chevauchent et plus de k-mers voisins sont portés par les mêmes reads. À titre d'ordre de grandeur, une profondeur de 10X induit typiquement des recouvrements d'environ 90 % de la longueur des reads, et 100X d'environ 99 %. Les couleurs observées correspondent donc majoritairement à des combinaisons de reads se chevauchant, donc à des région génomiques, ce qui borne fortement le nombre de couleurs distinctes malgré un nombre très élevé de k-mers. Il est alors pertinent de factoriser les couleurs : à l'instar de Mantis (101) ou Rainbowfish (5), qui ne stockent chaque ensemble de reads qu'une seule fois dans une table de classes de couleurs, et chaque k-mer ne garde qu'un identifiant de classe. Seules les erreurs de séquençage peuvent impacter ce principe et créer de nouvelles couleurs, comme illustré en Figure 3.9.

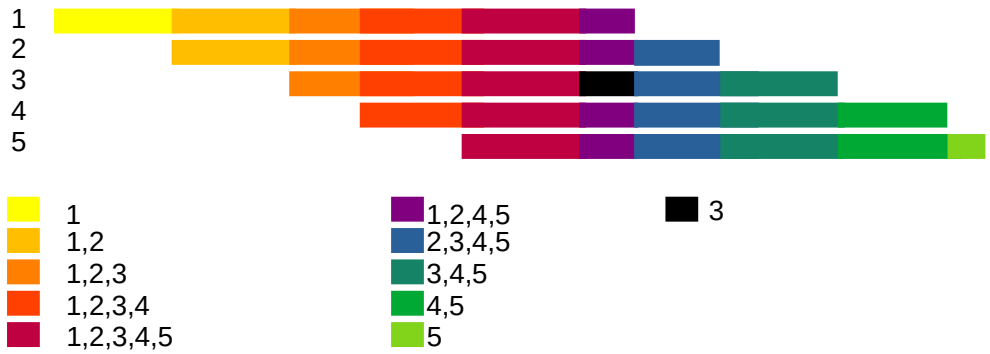


FIGURE 3.9 – *Exemple illustrant comment les reads successifs créent des sous-séquences dont les k-mers constituent une même couleur. Une erreur de séquençage est montrée en noir, et nécessite donc de créer une couleur supplémentaire.*

L'objectif à accomplir est à la fois de ne stocker qu'une seule fois chaque couleur, mais aussi de permettre de les compresser pour un gain complémentaire.

Pour ne stocker que des couleurs uniques, nous avons divisé la structure en deux parties, à l'image de Mantis (101), pour stocker d'un côté les *k*-mers, et de l'autre les couleurs. Deux tables de hachage sont donc créées, associant chaque *k*-mer à un identifiant de couleur, et chaque identifiant de couleur à la couleur elle-même, comme illustré en Figure 3.10.

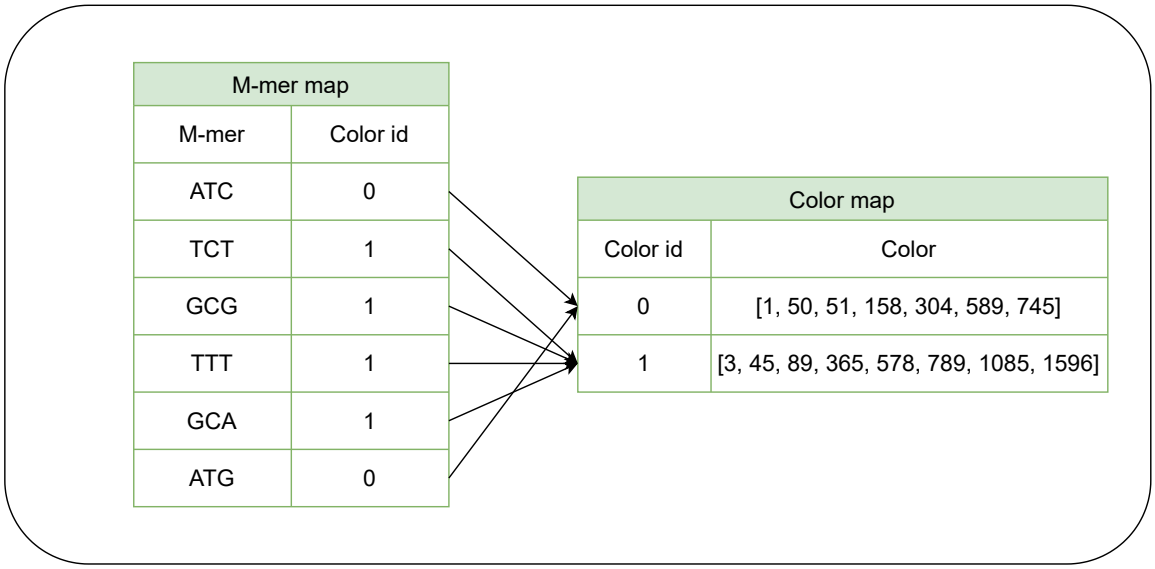


FIGURE 3.10 – *Structure de K2R utilisant deux tables, une pour les k-mers, la seconde pour les couleurs. Les identifiants de reads sont ici écrits sous forme de liste, qui est en réalité compressée.*

Nous avons voulu mesurer l'impact en termes de nombre d'éléments stockés de cette modification. Pour cela, nous avons indexé un jeu de données *E. coli*, ayant un taux d'erreur de 1%. Le nombre de *k*-mers indexés est de 4.553.982 contre uniquement 88.345 couleurs. Le nombre de couleurs créées est donc divisé par 51 par rapport au nombre de *k*-mers.

Même si le nombre de couleurs est considérablement réduit, stocker un nombre de vecteurs d'entiers important reste extrêmement coûteux. Nous avons cette fois-ci cherché non pas à modifier la structure, mais plutôt à la compresser.

3.1.4 Compression des couleurs

Nous souhaitons donc ici compresser les couleurs, c'est-à-dire compresser des listes d'entiers représentant les identifiants de reads. Nous avons fait le choix dans ce projet d'utiliser **TurboPFor** (108), outil permettant de compresser des listes d'entiers croissants efficacement en utilisant les instructions SIMD disponibles, pour diminuer le coût en mémoire des couleurs. Afin de garantir que les listes d'identifiants soient croissantes, le fichier de reads est parcouru une unique fois en attribuant un entier à chacun, allant de 0 au nombre de reads total. TurboPFor utilise le delta encoding qui, au lieu de stocker chaque entier d'une liste, va préférer stocker la différence entre chaque entier comme montré en Figure 3.11.

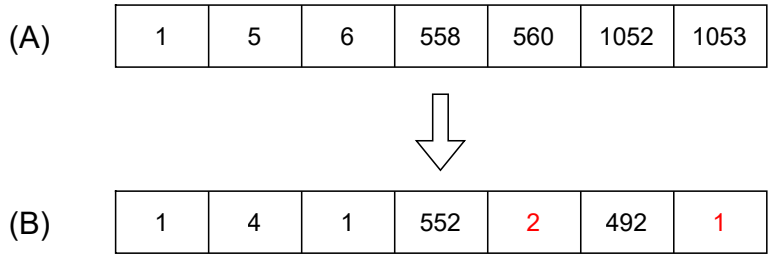


FIGURE 3.11 – *Exemple de fonctionnement du delta encoding. Seules les différences entre les entiers du vecteur (A) sont stockées dans le vecteur (B). En rouge sont indiqués les gains les plus significatifs, c'est-à-dire une différence faible entre deux entiers élevés.*

Plus les entiers sont proches, plus la compression est efficace car la différence est plus faible.

TurboPFor offre des niveaux de compression proches de l'état de l'art pour un coût de calcul très faible, illustré en Figure 3.12. Par contre, ce coût dépend directement du nombre d'éléments à compresser : plus la liste est longue, plus le temps de traitement augmente. Pour éviter le stockage en RAM de listes entières d'entiers et limiter le surcoût lié à la décompression/recompression lors de chaque modification, nous avons adopté une approche de compression progressive. Un système de buffer de taille fixe (32 éléments) est utilisé : les entiers y sont insérés au fur et à mesure, et dès que le buffer est plein, son contenu est ajouté à la liste principale puis compressé, avant d'être vidé. À la fin de l'indexation, si le buffer n'est pas vide, les éléments restants sont ajoutés et compressés dans la structure finale. Le choix de la taille de buffer est un compromis temps/mémoire : un petit buffer sera léger en RAM, mais réduira très peu le temps utilisé par la compression. Un plus grand buffer aura l'impact inverse. Ce mécanisme réduit le nombre d'éléments traités simultanément et garantit un temps de compression raisonnable.

Chaque liste d'identifiants de read sera considérée comme compressée dans la suite de ce manuscrit.

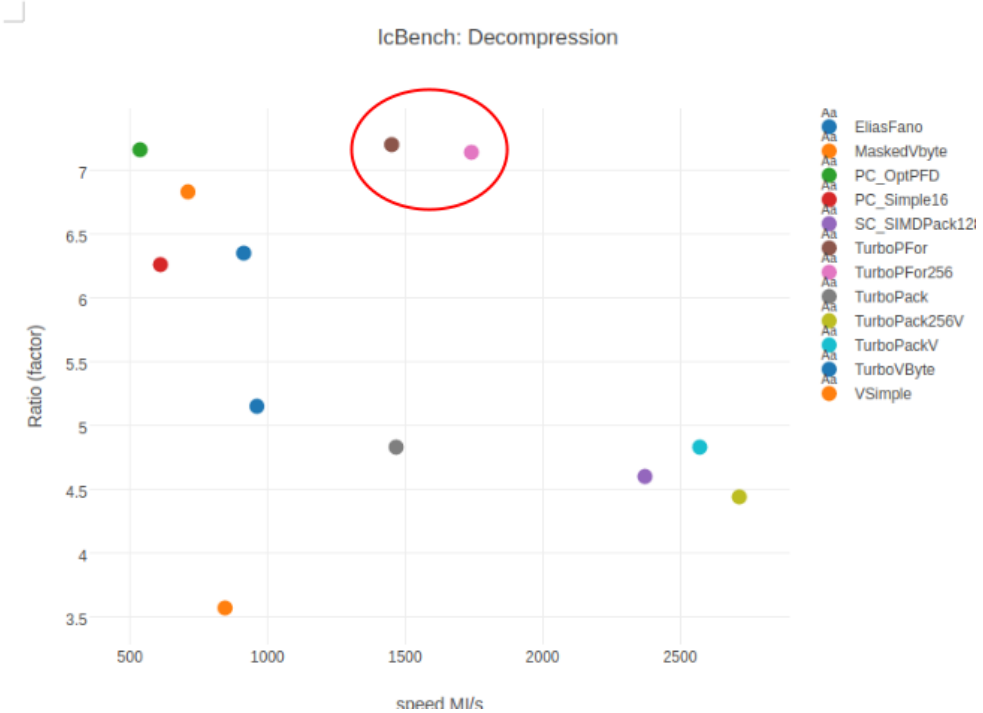


FIGURE 3.12 – *Performances de TurboPFor, en comparaison avec d'autres méthodes. Ce graphique a été pris sur le dépôt GitHub de TurboPFor :* `https://github.com/powturbo/TurboPFor-Integer-Compression`.

En pratique, en l'intégrant à K2R, le gain en mémoire augmente en fonction de la profondeur du jeu de données traité (Figure 3.13). Un facteur 3 peut être atteint en indexant un jeu de reads générés à partir du génome *E. coli*, pour une profondeur de 200X.

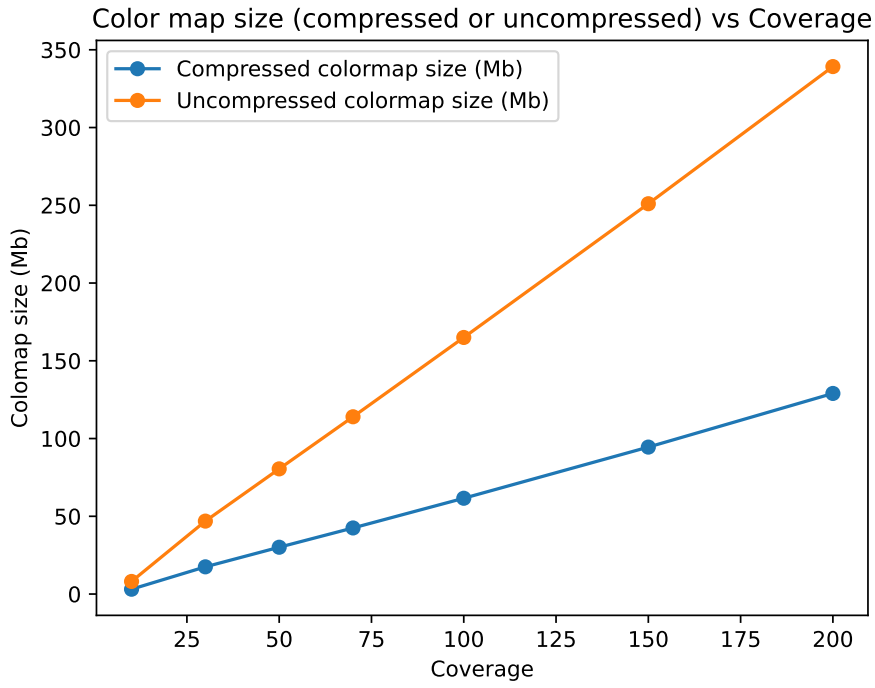


FIGURE 3.13 – *Comparaison de la taille de la table des couleurs, en les compressant ou non. Le jeu de données utilisé consiste en des reads simulés à partir du génome E. coli, avec un taux d'erreur de 0.01%. Le gain en mémoire augmente progressivement, jusqu'à atteindre un facteur 3 pour une profondeur de 200X.*

Nous avons donc à ce stade un index fonctionnel, que nous avons commencé à optimiser en espace. Une autre amélioration évidente concerne les k-mers, actuellement tous stockés sur 64

bits. Nous avons évoqué précédemment le principe des minimizers. C'est ce concept que nous avons choisi d'adopter pour poursuivre l'amélioration de K2R.

3.1.5 Minimizers

Un minimizer d'une séquence est le plus petit sous-mot de longueur m (selon un ordre total fixé) parmi les sous-mots couverts par une fenêtre glissante. Plus formellement, pour $m \leq k$ et un ordre total sur l'alphabet, le m-minimizer d'un k-mer x est le plus petit m-mer parmi les $k - m + 1$ sous-chaînes de x selon cet ordre (avec règle de départage en cas d'égalité).

Un ensemble de minimizers constitue une représentation légère mais approximative d'une séquence : des k-mers consécutifs partagent fréquemment le même minimizer, ce qui permet de ne le stocker qu'une fois au lieu de l'ensemble des k-mers qu'il couvre (Figure 3.14). Au-delà du schéma standard, des schémas plus élaborés réduisent encore la quantité de minimizer necessaire pour une couverture garantie : mod-minimizers, syncmers, universal hitting sets. Parmi ces approches, nous utilisons l'heuristique du decycling set (104), permettant de diminuer au maximum le nombre de minimizers necessaires.

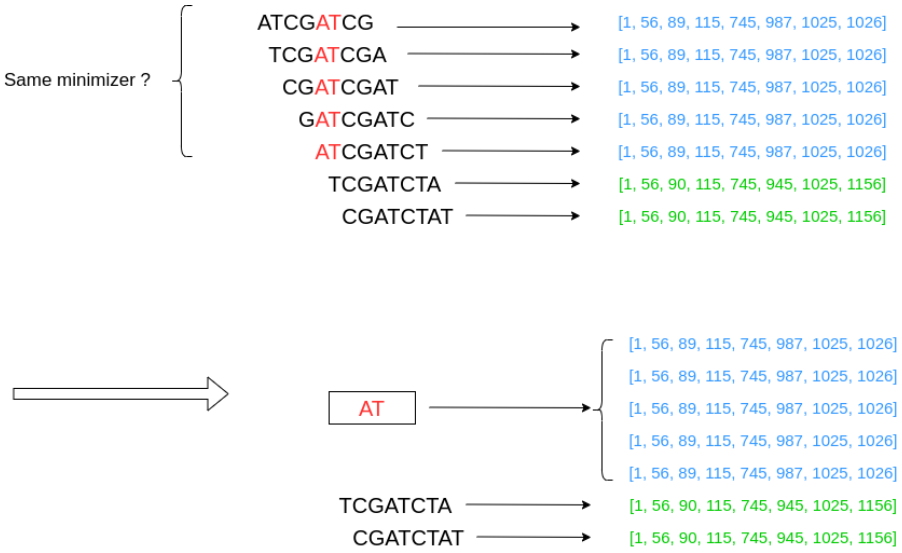


FIGURE 3.14 – *Réduction du nombre d'éléments stockés dans la table des k-mers en utilisant leur minimizers. Les 5 k-mers ayant pour minimizer AT sont compactés en ne gardant que ce dernier.*

La combinaison de l'utilisation d'une table des couleurs et des minimizers permet de réduire le nombre d'éléments stockés de plusieurs ordres de grandeur, comme illustré en Figure 3.15. Dans le cas du génome de *C. elegans*, 94 millions de k-mers sont indexables, mais seuls un peu plus de 6 millions de minimizers sont stockés et 4 millions de couleurs distinctes. Nous divisons donc par 15 le nombre de k-mers, et plus de 20 le nombre de couleurs.

	K-mers	Minimizers	Distinct Colors
E.Coli	4.553.982	443.123	88.345
C.Elegans	94.006.409	6.338.436	4.475.066

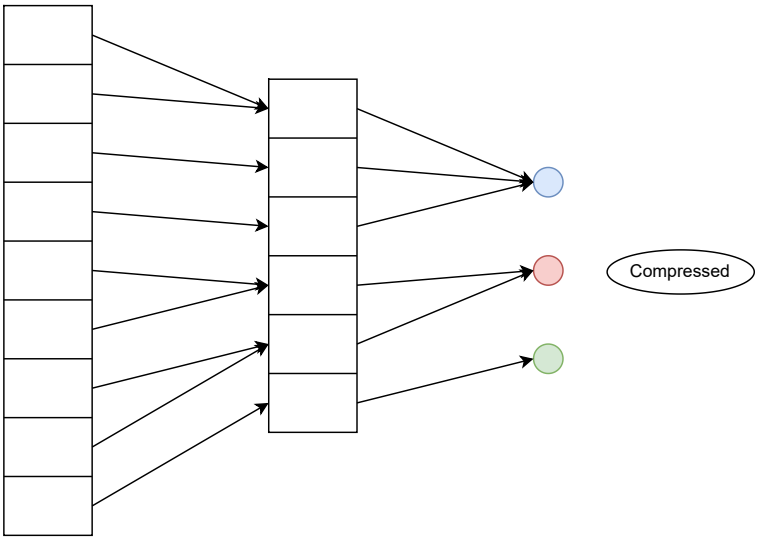


FIGURE 3.15 – *Impact en termes de nombre d'éléments stockés de l'utilisation des minimizers et de la table des couleurs. L'utilisation des minimizers nous permet ici de gagner un ordre de grandeur minimum, la table des couleurs de réduire d'au moins un tiers le nombre d'éléments stockés.*

En termes de taille finale de structure (Figure 3.16), en comparaison avec le fichier de reads indexés, nous sommes désormais capables de la réduire par un facteur 6 (ici à partir de reads générés à partir du génome d'*E. coli*, de longueur 10.000 et ayant un taux d'erreur de 1%). À titre de comparaison, cette même expérience avec la version initiale de K2R (Figure 3.4) multipliait par un facteur 10 la taille du fichier lorsqu'il était indexé.

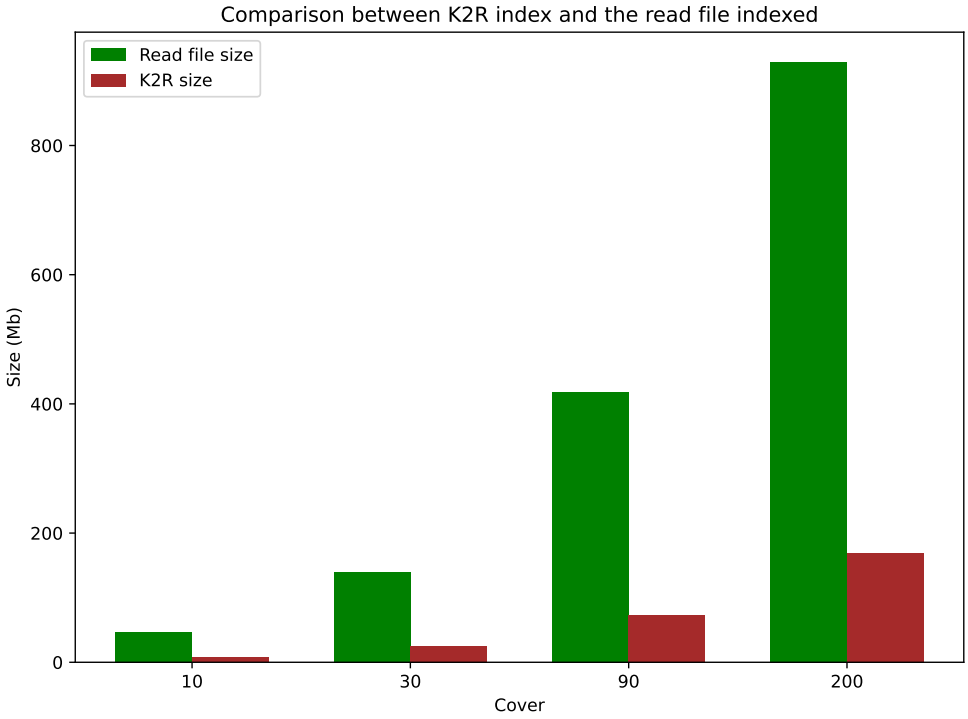


FIGURE 3.16 – *Comparaison entre la taille d'un fichier de reads générés à partir du génome d'E. coli, de longueur 10.000 et ayant un taux d'erreur de 1%, et la taille finale de l'index créé à partir de ce fichier, grâce à la version finale de K2R. Plusieurs couvertures sont présentées ici, allant de 10X à 200X. La taille de la structure d'indexation est environ 6 fois moins lourde que le jeu de reads.*

Un de nos objectifs initial était de pouvoir assurer des **requêtes exactes** à l'utilisateur, c'est-à-dire de n'avoir aucun faux positif en sortie. Or, avec les collisions induites par les filtres de Bloom et désormais les minimizers, cette condition ne peut pas être assurée. En effet, les minimizers peuvent induire des faux positifs. Plusieurs k-mers différents peuvent avoir le même minimizer. Dans le cas de K2R, et plus généralement dans le cas d'index approchés, un k-mer absent peut être considéré comme présent si son minimizer est égal à celui d'un k-mer appartenant réellement aux données. Les structures créées à partir de minimizers sont donc approximatives. Pour des questions de performances (temps et mémoire), nous souhaitons garder cet index approximatif. Nous allons donc gérer ce problème au niveau de la requête, en introduisant une étape de vérification exacte.

Chaque élément de l'indexation a pu désormais être optimisé : k-mers transformés en minimizers, couleurs uniques et compressées, utilisation de filtres. Le résultat de ces différentes étapes constitue l'état actuel de K2R, disponible sur GitHub : `https://github.com/LeaVandamme/K2R.git`.

Nous nous pencherons désormais sur la présentation de l'opération de requête.

3.1.6 Requêtes

L'étape de requête est entièrement indépendante de l'étape d'indexation. Chacune d'elles consiste à rechercher les reads contenant une séquence d'intérêt, à partir du calcul du nombre de minimizers puis de k-mers communs.

Rappelons que la structure d'indexation utilisée est approximative. L'emploi de minimizers pour représenter les k-mers, combiné à leur stockage dans des filtres de Bloom, introduit des faux positifs : certains éléments peuvent être indiqués comme présents alors qu'ils ne le sont pas réellement. Concrètement, lors d'une requête, un read peut présenter un nombre suffisant de minimizers communs avec la séquence recherchée, alors que le nombre réel de k-mers partagés est insuffisant. Ce taux de faux positifs dépend notamment de la taille du filtre de Bloom et de celle des minimizers choisis.

Néanmoins, l'objectif est d'obtenir une requête exacte, c'est-à-dire sans faux positifs dans les reads retournés. Pour cela, un mécanisme de vérification a été mis en place :

— Un premier calcul du nombre de minimizers communs entre le read et la requête est effectué. Si ce nombre est inférieur au seuil minimal, le read est écarté.
— Si le nombre de minimizers communs est supérieur ou égal à ce seuil, la séquence du reads ciblé est récuperée et une vérification exhaustive de tous les k-mers est réalisée afin de confirmer la présence de la séquence recherchée.

Un résumé du système de requête est présenté en Figure 3.17.

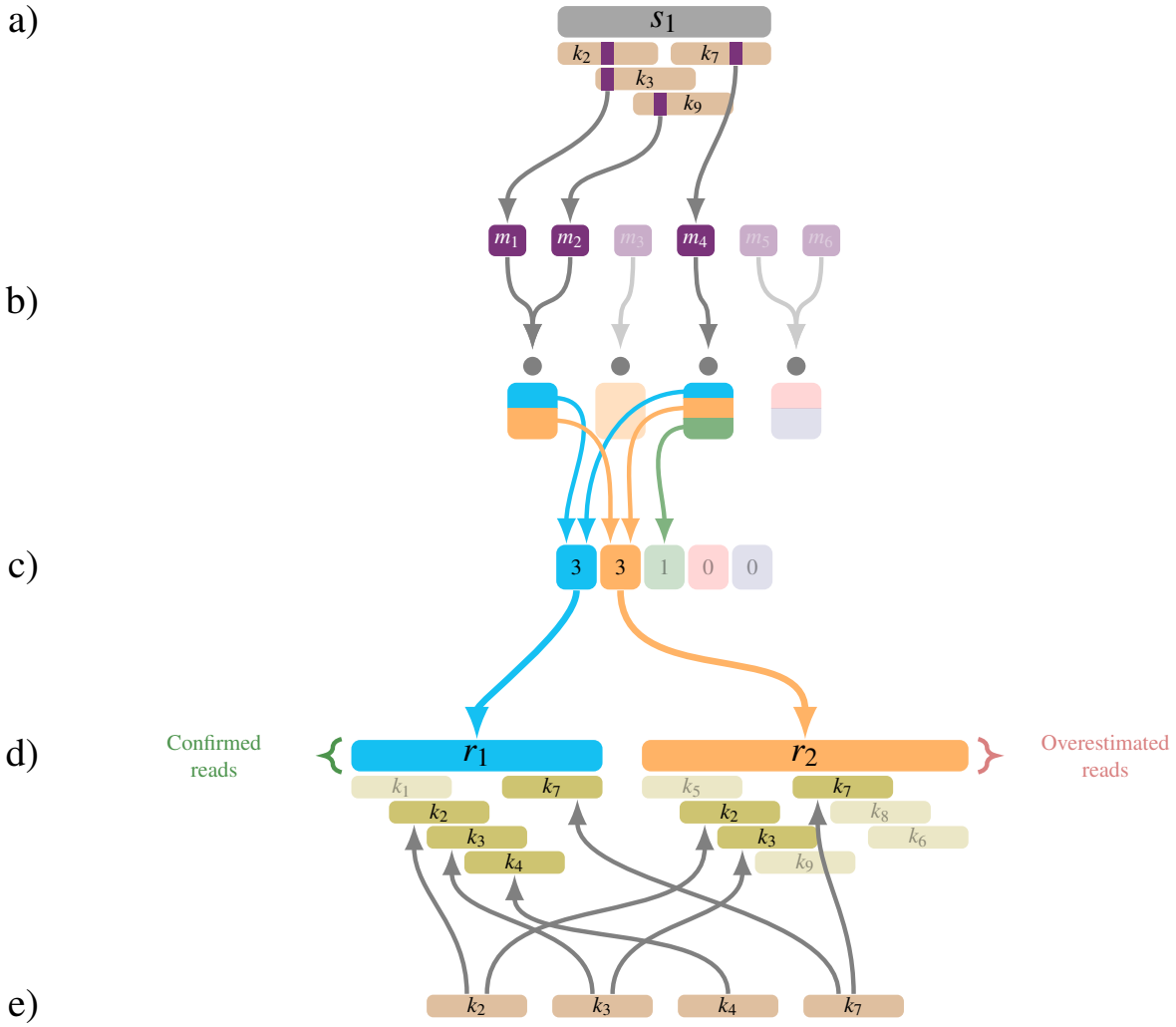


FIGURE 3.17 – *Résumé du fonctionnement des requêtes avec K2R. La première étape (a) est le découpage en k-mers de la séquence à requêter. Ses minimizers sont calculés, puis recherchés dans l'index (b). Seuls les reads contenant un pourcentage minimal de minimizers sont retenus (c), puis chaque k-mer de la requête est recherché dans ces reads afin de ne garder que les vrais positifs (d et e).*

3.2 Détails d'implémentation

3.2.1 Création de l'index

Dans notre structure d'index, illustrée en Figure 3.18 parties 5) et 6), nous employons un système de deux maps :

— minimizer map [minimizer : identifiant de couleur], où les minimizers et identifiants de couleurs sont des entiers

— color map [identifiant de couleur : couleur], où l'identifiant de couleur est un entier et la couleur un objet.

Nous avons choisi d'utiliser une hashmap unordered dense disponible sur `http://github.com/martinus/unordered_dense` en raison de sa rapidité. Elle a été exploitée dans plusieurs outils bioinformatiques, notamment kallisto (23) et mashmap (58).

Notre structure est dynamique, les reads sont ajoutés au fur et à mesure de la lecture du fichier d'entrée. Lors de l'insertion d'un read dans l'index, ses minimizers sont calculés et insérés dans la map des minimizers. Chaque identifiant correspondant est alors mis à jour pour incor-

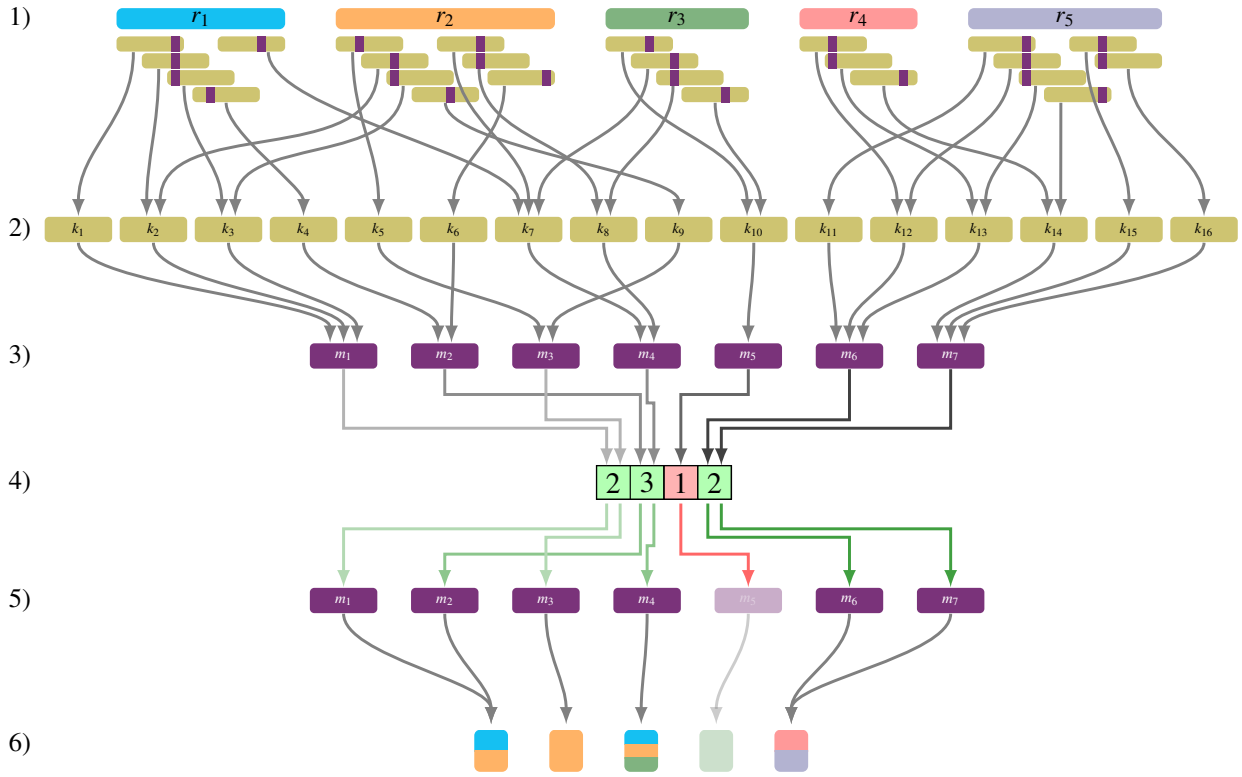


FIGURE 3.18 – *Schéma de la structure de K2R. Pour un ensemble de reads (1), nous pouvons extraire l'ensemble des k-mers (2). Chaque k-mer (2) est lié à un minimizer (3) avec des collisions possibles. En utilisant le counting Bloom filter (4), nous pouvons supprimer les minimizers trop peu fréquents (5). Chaque minimizer restant (5) est lié à une couleur (6).*

porer le nouvel identifiant de read dans sa liste de couleurs. Si la couleur associée ne préexiste pas dans la map, un nouvel identifiant est généré et ajouté, avec sa nouvelle couleur associée.

Pour que l'index reste léger et efficace, nous avons mis en place un compteur associé à chaque couleur indiquant le nombre de minimizers auxquels elle est associée. Lorsque l'association d'un minimizer passe à une couleur différente, deux scénarios se présentent. Premièrement, si un ou plusieurs minimizers sont encore associés à l'ancien identifiant de couleur, si la nouvelle couleur existe, le minimizer peut y être automatiquement associé, sinon une nouvelle couleur est créée à partir de l'ancienne (ajout du dernier identifiant à la suite des autres). Deuxièmement, si la couleur n'est plus associée à aucun minimizer, cette couleur est supprimée.

Notre choix de structure d'index lui confère des capacités dynamiques, permettant de pouvoir compléter un index déjà construit avec d'autres données. Alors que les structures statiques peuvent offrir des avantages en termes d'efficacité en mémoire, elles manquent de flexibilité et nécessitent une phase de construction plus lourde. Une perspective naturelle pour ce travail est de concevoir une structure statique plus efficace dans le cas ou la dynamicité n'est plus necessaire.

3.2.2 Requêtes

Input Un index créé au préalable avec K2R sous forme de deux fichiers binaires, un fichier FASTA contenant les séquences à rechercher.

Déroulement En suivant la Figure 3.17, 5 étapes sont nécessaires aux requêtes :

— a) La requête est découpée en *k*-mers, et les minimizers correspondants sont calculés.

— b) Les couleurs correspondantes aux minimizers sont accessibles directement depuis les deux maps créées lors de l'indexation.
— c) On recherche les reads qui contiennent un pourcentage minimum de ces minimizers.
— d) et e) Une fois cet ensemble constitué, on élimine les éventuels faux positifs. Les *k*-mers des séquences d'intérêt sont recherchés dans l'ensemble de reads constitué, en vérifiant chaque caractère. Chaque requête est donc exacte.

Output Plusieurs possibilités de retour sont possibles :
— Pour chaque séquence d'intérêt, un fichier FASTA est créé contenant les séquences de tous les reads qui ont un pourcentage minimum de *k*-mers communs avec elle. Cette solution nécessite une structure supplémentaire permettant de stocker les positions de chaque séquence dans le fichier, afin de les récupérer rapidement lors de la requête.
— Un fichier est créé avec le nombre de matchs pour chaque séquence d'intérêt, c'est-à-dire le nombre de reads comportant le pourcentage minimum de *k*-mers communs avec la séquence.

3.2.3 Filtres d'abondance

Nous avons vu précédemment que les erreurs de séquençage amènent de nouveaux *k*-mers. La figure 3.9 l'illustre en montrant l'obligation de créer une nouvelle couleur supplémentaire lorsqu'une erreur apparaît. Ces *k*-mers seront probablement très peu abondants, et peuvent ne pas être biologiquement intéressants à l'analyse. Leur suppression n'engendre donc dans ce cas aucune perte de signal, tout en allégeant l'index. Dans ce but, nous avons mis en place un mécanisme de filtrage optionnel des minimizers, et donc *k*-mers faibles (n'apparaissant qu'un nombre très limité de fois) en utilisant un Counting Bloom filter afin d'estimer approximativement l'abondance des minimizers dans le jeu de données. Un Counting Bloom filter ne peut que surestimer l'abondance, ce qui peut entraîner des faux positifs, certains minimizers faibles étant à tort considérés comme solides en raison de collisions de hachage avec de véritables minimizers solides. Il ne génère cependant aucun faux négatif.

Après ce comptage approximatif des minimizers, nous transformons le Counting Bloom filter en un Bloom filter classique, en convertissant chaque cellule (initialement stockée sous forme d'entier représentant l'occurrence) en un simple bit, 1 si l'abondance dépasse le seuil 0 sinon. Ce bit indique la solidité du minimizer associé, réduisant ainsi considérablement la consommation mémoire lors de la construction effective de l'index. Grâce à cette stratégie, nous limitons l'utilisation mémoire maximale, car les tables de hachage deviennent la principale source d'utilisation mémoire lorsque le Bloom filter est correctement dimensionné.

Nous observons également un phénomène inverse : certains minimizers surabondants sont liés à des régions hautement répétées dans un génome, que ce soit en raison de mécanismes biologiques ou du type de séquençage utilisé. Ces minimizers, pouvant être associés à un très grand nombre de reads, risquent de dégrader les performances de l'indexation et de "polluer" les résultats avec de nombreuses correspondances non pertinentes. C'est notamment l'étape de compression qui est impactée, TurboPFor devenant moins efficace sur de grosses listes d'entiers. Pour atténuer cet effet, nous implémentons un filtrage optionnel de l'abondance maximale des minimizers, une approche souvent adoptée par d'autres outils, à l'image de minimap2 (72, 59), qui filtre les 0.02% de *k*-mers les plus abondants par défaut.

Concrètement, pour l'utilisateur, il est courant d'exclure les minimizers apparaissant une

seule fois, ceux-ci étant souvent le résultat d'erreurs de séquençage. À l'inverse, une fréquence trop élevée peut également poser problème. Une borne supérieure est donc fixée afin de garantir la scalabilité de l'approche. En effet, chaque minimizer est associé à une liste d'identifiants de reads que nous cherchons à compresser, or, lorsque cette liste devient trop volumineuse, la compression devient un obstacle en termes de temps de traitement. Limiter l'abondance permet donc de préserver l'efficacité de la méthode.

Il est important de souligner que cette stratégie de filtrage n'est pas sans conséquence sur les performances en requête, en raison de la perte d'information qu'elle entraîne. Dans notre cas, en nous basant sur le jeu de données *C. elegans* à une profondeur de 100X, l'effet de ce filtrage est particulièrement notable lorsqu'on augmente le seuil minimal d'abondance (voir Figure 3.19). Par exemple, l'élimination des m-mers observés moins de quatre fois conduit à la suppression de 28 millions d'éléments, contre seulement 13 millions finalement conservés et indexés. Cette réduction drastique souligne l'impact potentiel du filtrage sur la sensibilité de l'index lors des requêtes.

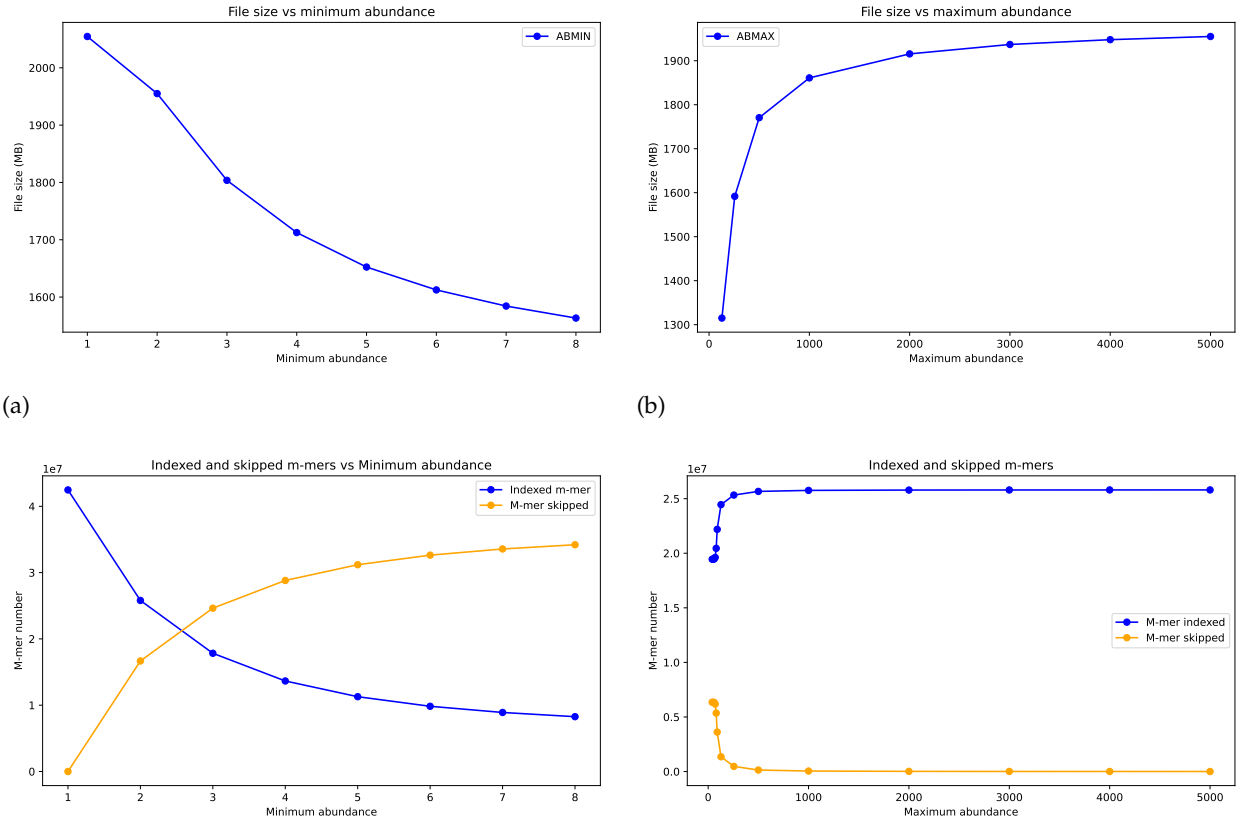
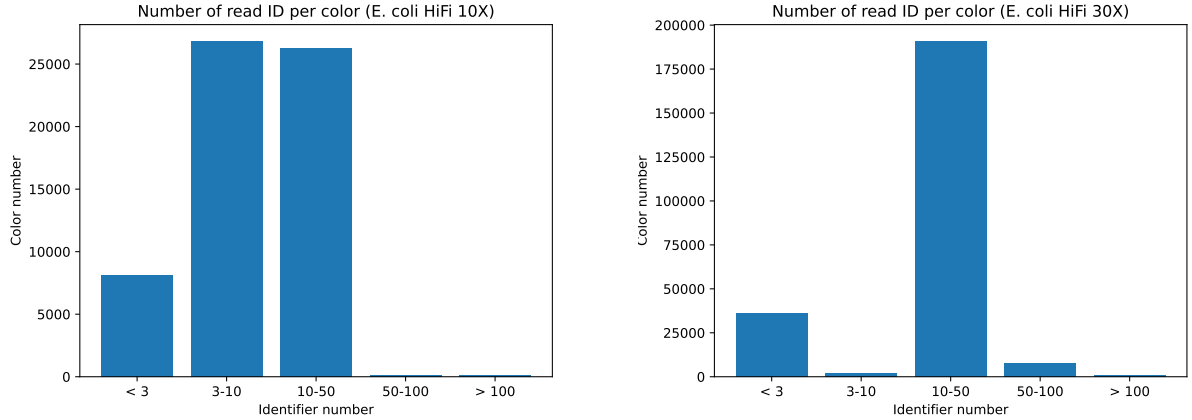


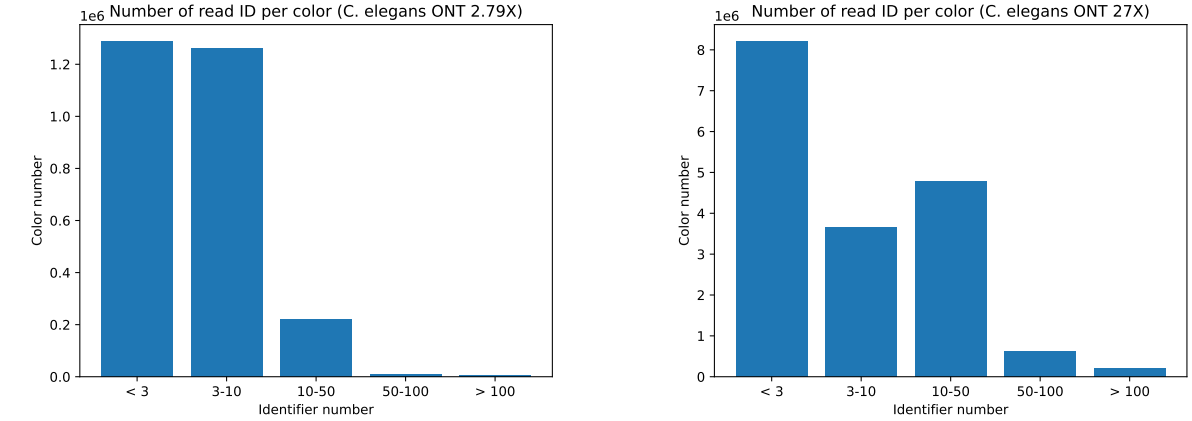
FIGURE 3.19 – *Impact de l'abondance donnée par l'utilisateur (minimum ou maximum) sur la taille finale de structure et le nombre de minimizers indexés. Le jeu de données utilisé consiste en un jeu de reads simulés à partir du génome C. elegans de longueur 10.000, ayant une profondeur de 100X et un taux d'erreurs variant de 0.1% à 10%. Dans le cas de minimum 3.19a d'abondance, on observe une diminution très rapide de la taille de structure (jusqu'à un minimum de 4), puis elle tend à se stabiliser progressivement. Pour le maximum 3.19b d'abondance, la taille explose assez rapidement lorsque le filtre est moins restrictif. Concernant les minimizers stockés (3.19c et 3.19d), la tendance est similaire.*

On peut aussi observer l'effet que ce filtrage peut avoir en en étudiant le nombre d'identifiants de reads pour chaque couleur (Figure 3.20) lors d'indexations. Un nombre de couleurs non négligeable contient très peu d'identifiants, et sont très probablement associées à des k-mers erronés.



(a)

(b)

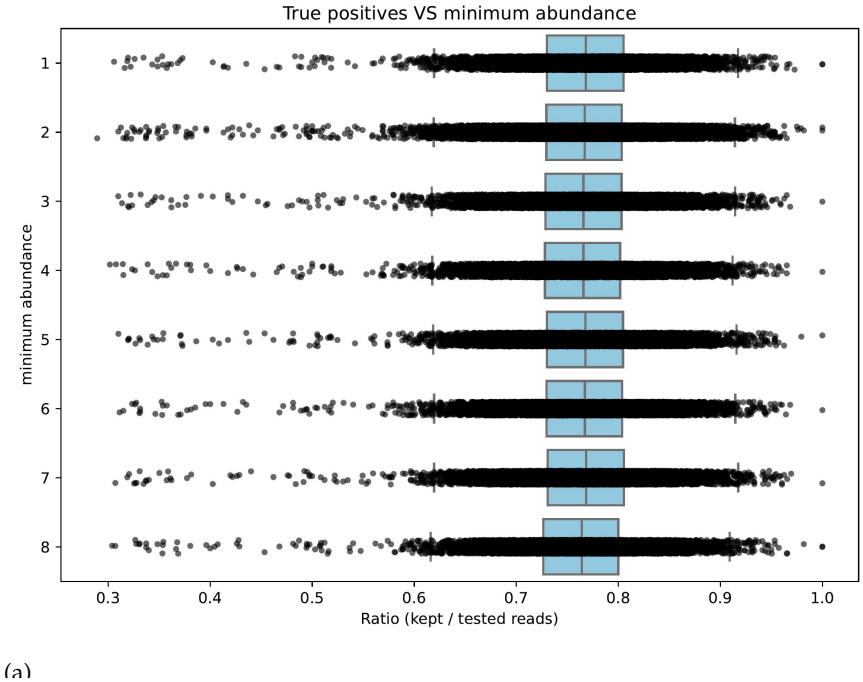


(c)

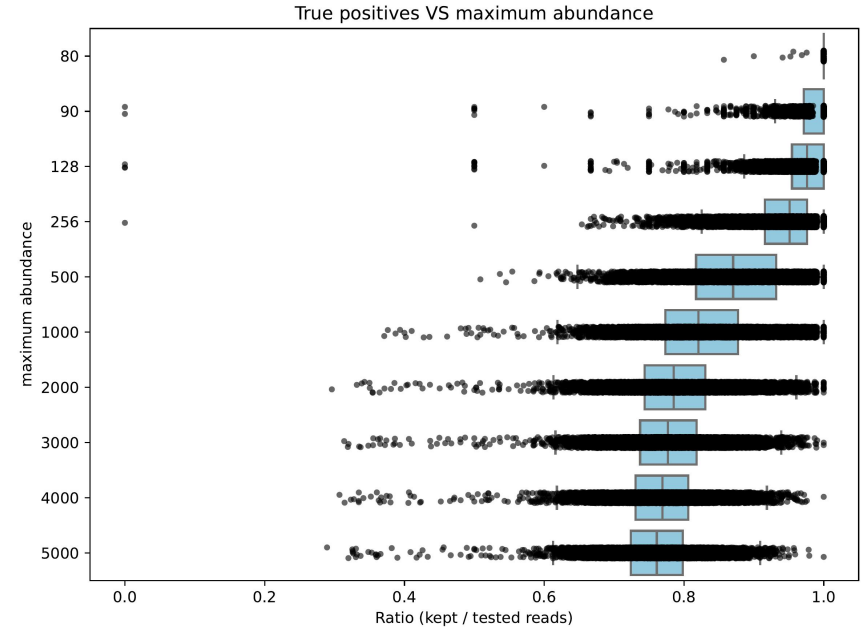
(d)

FIGURE 3.20 – *Statistiques sur le nombre d'identifiants de reads par couleur. Sur les jeux de données E. coli (Figures 3.20a et 3.20b) une grande majorité des couleurs ont un nombre d'identifiants correspondant globalement à la profondeur du dataset. Il y a cependant un nombre non négligeable de couleurs ayant moins de 3 identifiants, probablement issus d'erreurs. Dans le cas de datasets plus importants et plus erronés (6%) (Figures 3.20c et 3.20d), le nombre de couleurs ayant peu d'identifiants est décuplé, dépassant le nombre de couleurs correspondant à la profondeur.*

Afin de voir l'impact de ces suppressions sur le taux de faux positifs, nous avons réalisé les mêmes requêtes sur les différents index créés 3.21. En complément de ces faux positifs, des faux négatifs peuvent apparaître, c'est-à-dire des k-mers apparaissant dans le jeu de données, mais dont on donne l'absence. Ces cas peuvent arriver lorsque les filtres sont trop restrictifs, au point de ne pas stocker des k-mers d'intérêt. Ces cas-ci peuvent en pratique passer inaperçus, car on ne vérifie pas les reads n'ayant pas un pourcentage de minimizers communs suffisant.



(a)



(b)

FIGURE 3.21 – *Impact du filtre de l'abondance minimum et maximum sur le taux de faux positifs. Le jeu de données indexé consiste en un jeu de reads simulés de longueur 10.000, ayant une profondeur de 100X et un taux d'erreurs variant de 0.1% à 10%. Les requêtes sont des reads simulés sur le même jeu de données avec le même taux d'erreurs. Le minimum d'abondance n'a presque aucun impact sur les faux positifs. Le maximum semble par contre les impacter plus largement : au plus le maximum est élevé, au plus le nombre de faux positifs est augmenté.*

3.2.4 Parallélisation

Le développement de K2R s'est terminé par une longue phase de parallélisation, à l'indexation comme à la requête, afin d'optimiser au maximum les temps nécessaires pour réaliser les opérations voulues. Cette étape est complexe, notamment à cause du fait que les deux maps principales (m-mers et couleurs) sont modifiées simultanément, et au sein d'une même table les couleurs sont elles aussi modifiées en même temps (ajout d'identifiants de read). Il était donc nécessaire de veiller à ce que chaque opération n'entre en concurrence avec aucune autre, ce qui nécessitait une gestion des "lock" minutieuse.

La phase de filtrage est parallélisée à l'aide d'un tableau de mutex (MUTual EXclusion), qui est un mécanisme de synchronisation qui garantit qu'un seul thread à la fois peut accéder à une ressource partagée. Cela permet de sécuriser les différentes sections du Counting Bloom Filter, et de gérer les opérations concurrentes de manière efficace.

La parallélisation de la phase de construction de l'index est plus complexe en raison des dépendances avec des maps synchronisées. Pour atténuer cette difficulté, nous mettons en place une parallélisation inter-reads, où les minimizers sont calculés simultanément sur plusieurs reads en même temps.

Les tâches liées à la gestion des couleurs, notamment la décompression, la mise à jour, le tri et la recompression, sont également exécutées en parallèle. Elles sont protégées par des tableaux de mutex, garantissant l'intégrité des données. Grâce à cette architecture, il est possible de modifier simultanément les maps de couleurs et de m-mers sans risque de corruption des données.

Enfin, les requêtes peuvent elles aussi être exécutées en parallèle. Contrairement à la construction, délicate à gérer et nécessitant de nombreux mutex pour réaliser les opérations concurrentes, les requêtes sont gérées de manière indépendante de manière concurrente sans lock. L'augmentation du débit est donc encore bien plus élevée qu'à la construction.

Une subtilité dans notre implémentation bloque cependant l'utilisation d'un nombre de threads élevé, ou en tout cas minimise les performances. Comme expliqué ci-dessus, pour faciliter la parallélisation, nous avons décidé de paralléliser l'étape de traitement de chaque read. Donc pour chacun d'entre eux, on veut éviter que des threads concurrents, c'est-à-dire qui correspondent à des reads différents, ne modifient la structure de données pour les mêmes minimizers. Un système de lock a donc été mis en place au niveau des minimizers : au début du traitement, tous sont verrouillés et ne seront déverrouillés que lorsque qu'on a mis à jour la structure pour ce minimizer pour un read. Pour optimiser la mémoire, au lieu de verrouiller chaque minimizer et donc créer 2^{32} verrous, nous avons choisi de verrouiller des ensembles de minimizers (1024 ensembles en tout) où chaque ensemble regroupe les minimizers modulo 1024. Pour déverrouiller un ensemble le plus rapidement possible une fois que tous les minimizers de l'ensemble ont été traités, nous utilisons un compteur pour savoir quand déverrouiller.

Ce principe, bien que permettant d'assurer qu'aucune concurrence n'apparaît, va créer des verrouillages ralentissant le processus. En effet, si deux reads possèdent au minimum un minimizer en commun, alors ils ne pourront pas être traités en même temps. Lorsque le nombre de threads utilisés augmente, alors ceux-ci auront plus de chance de se bloquer entre eux. C'est pourquoi, comme illustré en Figure 3.22, le temps nécessaire à l'indexation ne décroit pas strictement en fonction du nombre de threads utilisés. Le pic d'optimisation se situe entre 3 et 5 threads. Une piste d'amélioration naturelle est donc de concevoir une structure ayant des propriétés similaires mais étant plus parallèle.

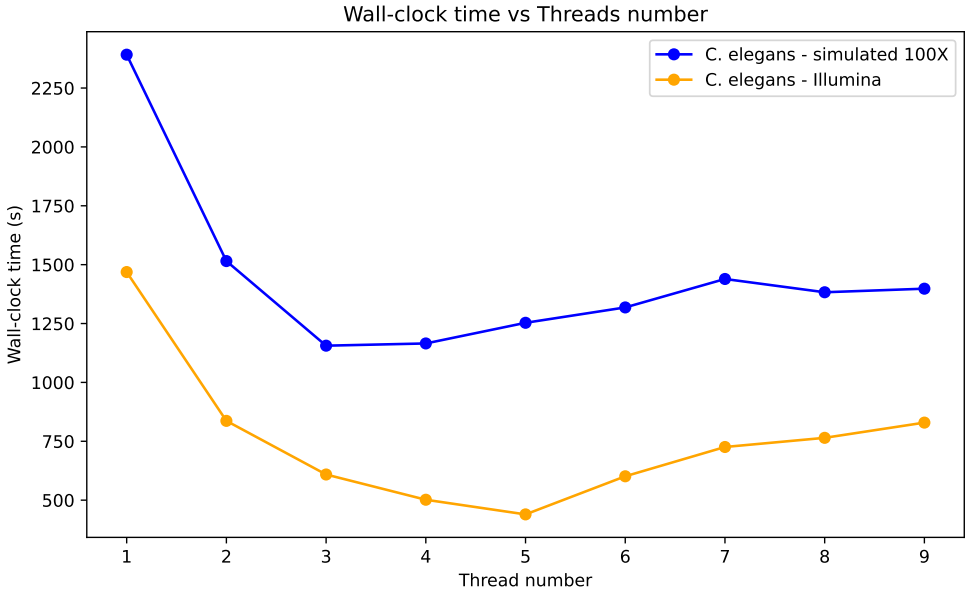
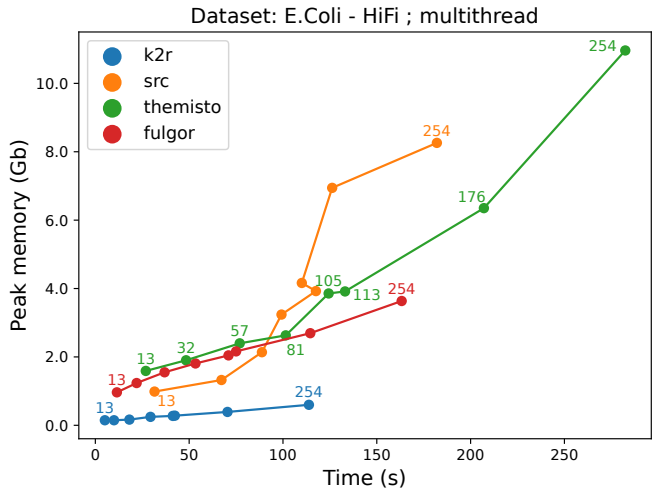
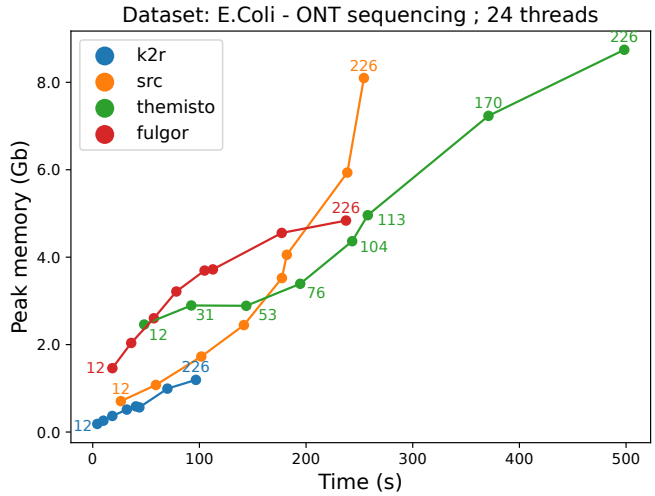


FIGURE 3.22 – *Impact du nombre de threads sur le temps nécessaire à l'indexation. Le pic d'optimisation se situe autour de 5 threads.*

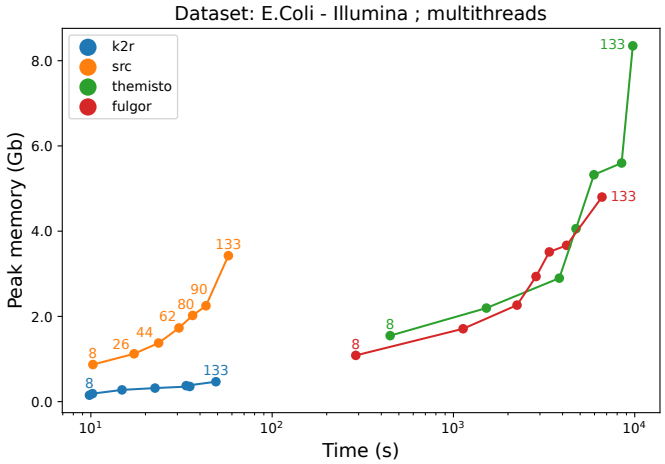
Nous avons voulu vérifier l'efficacité du multithreading mis en place. Nous avons donc relancé les mêmes expérimentations (sur jeux de données réels, sur plusieurs threads). K2R est lancé avec 5 threads pour les raisons évoquées ci-dessus, les autres outils sont lancés avec 32 threads. Les Figures 3.23 et 3.24 présentent ces résultats et confirment l'efficacité de la parallélisation, y compris sur de petits jeux de données tels que *E. coli*. Le temps nécessaire pour K2R est divisé par 3, et reste très largement inférieur au temps nécessaire pour l'état de l'art.



(a)

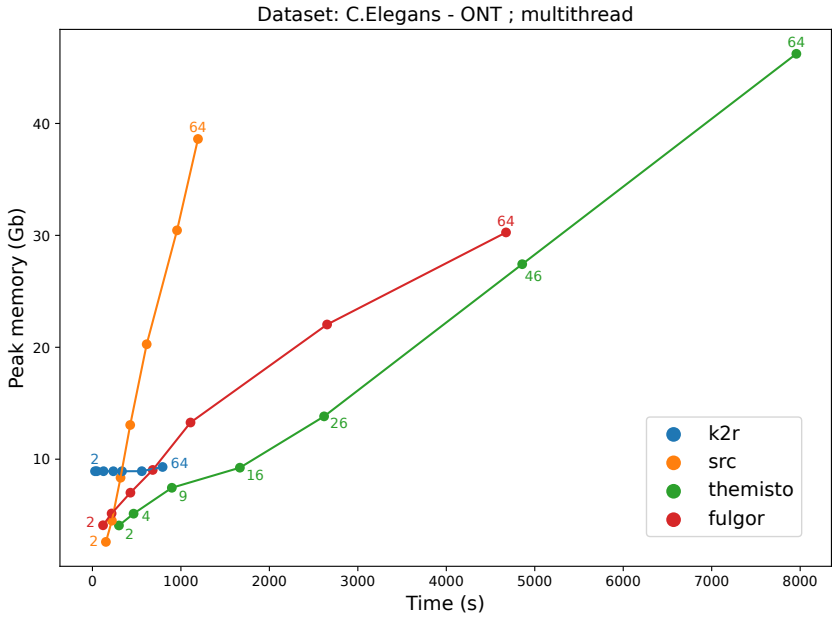


(b)

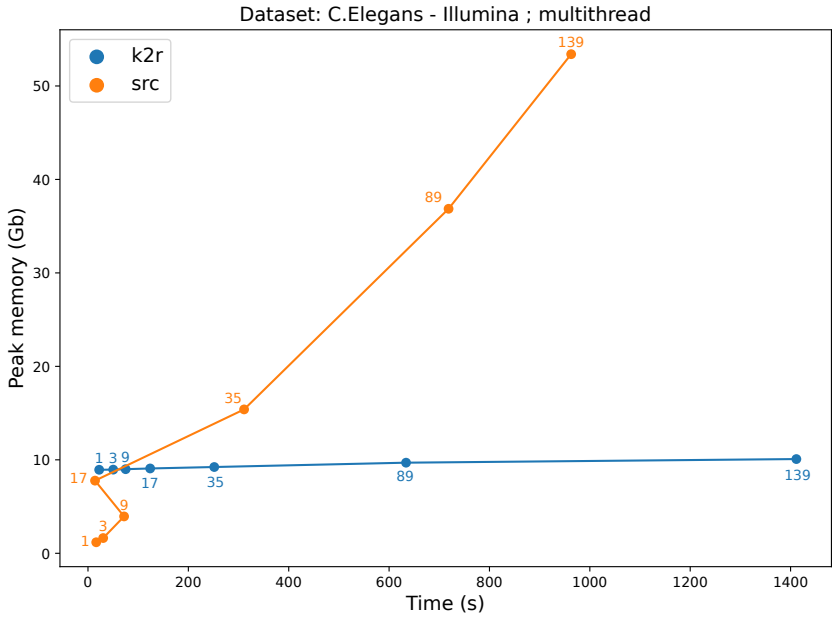


(c)

FIGURE 3.23 – *Comparaison de l'utilisation des ressources (temps d'exécution wall-clock et mémoire RAM) pendant la construction d'index avec K2R, SRC, Fulgor et Themisto en fonction des profondeurs, en utilisant plusieurs threads. Le jeu de données utilisé comprend des reads provenant de trois jeux de données différents du génome d'E. coli : (3.23a) HiFi, (3.23b) ONT, et (3.23c) Illumina, avec différentes profondeurs.*



(a)



(b)

FIGURE 3.24 – *Comparaison de l'utilisation des ressources (temps d'exécution wall-clock et mémoire RAM) pendant la construction d'index avec K2R et SRC, avec différentes profondeurs, en utilisant plusieurs threads. Les données utilisées consistent en des reads provenant de deux jeux de données différents du génome de C. elegans : (3.24a) ONT et (3.24b) Illumina, avec différentes profondeurs. On remarque que Themisto, Fulgor et Movi n'ont pas pu être testés ici, en raison respectivement de leur mode de fonctionnement et de leurs problèmes de passage à l'échelle.*

3.2.5 Homocompression

Pour corriger les erreurs de longueur de motifs homopolymériques, courantes dans les reads de séquençage HiFi et PacBio, nous proposons une fonctionnalité optionnelle d'homocompression. Cette technique représente les séquences des reads avec une perte en compressant les occurrences consécutives d'un même nucléotide X...X en une seule instance de X, par exemple : AAACCTCGGCG deviendrait ACTCGCG.

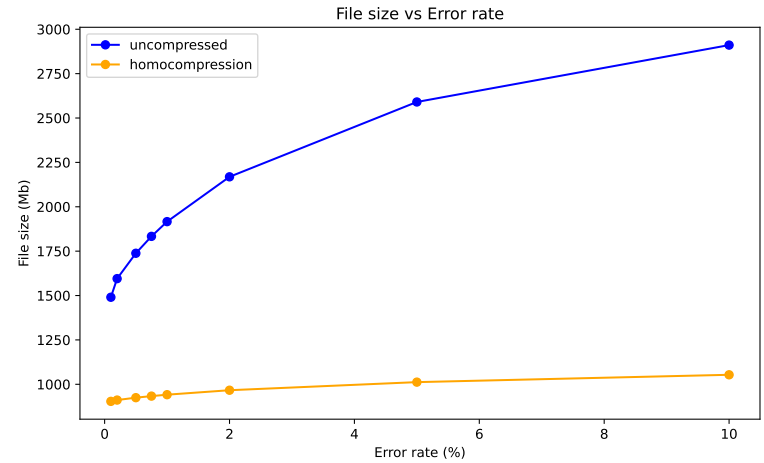
Comme évoqué dans l'article concernant La Jolla Assembler (14), assembleur de reads HiFi,

l'application de l'homocompression à ce type de reads peut entraîner une réduction par un facteur trois du nombre total d'erreurs, rendant ainsi une proportion significative des reads exemptes d'erreurs. Cette approche améliore non seulement la précision des données de séquençage, mais elle renforce également la fiabilité globale de l'analyse génomique en réduisant considérablement l'impact des erreurs liées aux homopolymères.

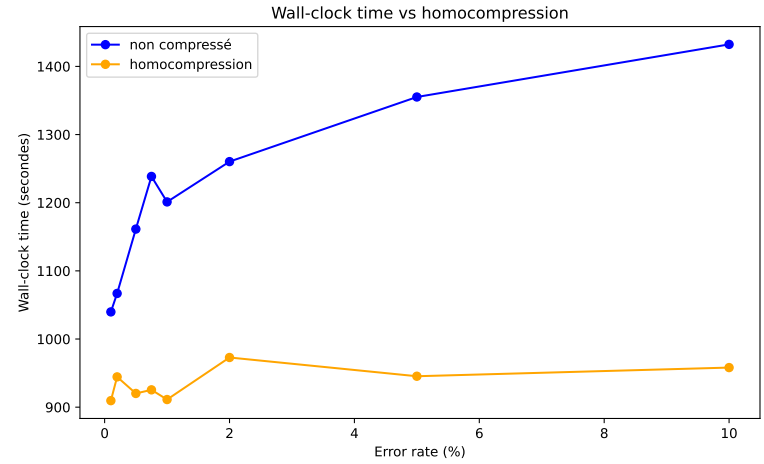
En termes d'impact sur l'index en lui-même, sa taille finale sera obligatoirement réduite, car les séquences sont raccourcies pour chaque répétition. Le temps d'indexation est donc aussi réduit. Cependant, chaque séquence à requêter doit être elle aussi homocompressée.

Nous étudions ici ces impacts, notamment en termes de taille finale de structure (Figure 3.25 (a)), et en temps de construction (Figure3.25 (b)).

Sur la Figure 3.25 (a), nous avons indexé un jeu de données simulé à partir d'un génome de *C. elegans*, ayant une profondeur de 100X, une longueur de 10.000 et un taux d'erreurs variant de 0.1% à 10%. Les deux courbes proposées montrent la taille finale stockée, avec et sans homocompression. On observe qu'au plus le taux d'erreurs est élevé, au plus le gain en mémoire est important (multiplié par deux ici entre les deux taux d'erreurs les plus éloignés). Même si le taux d'erreurs augmente, les temps et mémoire nécessaires restent presque constants. Comme attendu, le temps varie exactement de la même manière, en fonction de l'efficacité d'homocompression. Le gain est légèrement moins important, le facteur 2 n'est pas atteint.



(a)



(b)

FIGURE 3.25 – *Impact de l'homocompression sur la taille de l'index final (3.25a) et le temps wall-clock nécessaire (3.25b). Le jeu de données utilisé consiste en des reads simulés de longueur 10 000, avec une profondeur de 100× et un taux d'erreurs variant de 0,1 % à 10 %. En plus du gain en mémoire et en temps, on peut voir que le taux d'erreur n'a presque plus aucune incidence sur les performances.*

Dans la section suivante, nous aborderons les résultats de comparaison à l'état de l'art de K2R, en indexation comme en requête.

3.3 Comparaison à l'état de l'art

Nous avons décidé de comparer K2R à plusieurs outils qui constituent un état de l'art en termes d'indexation de *k*-mers :

— Short Read Connector (SRC) (87) : développé pour des reads courts, SRC est une structure associative (MPHF(79), associant les *k*-mers à la liste des identifiants de reads qui les contiennent).
— Fulgor (43) : un graphe de de Bruijn coloré, résultant de la combinaison d'un dictionnaire de *k*-mers avec un index inversé compressé.
— Themisto (3) : basé sur un graphe de de Bruijn en deux structures : une spectral BWT qui stocke les *k*-mers distincts, et les couleurs associées à chaque *k*-mer.
— Movi (133) : basé sur le Move index, il atteint à la fois un espace en $O_(r)$ et des requêtes en temps constant, où r est le nombre de runs dans la BWT.
— Full-br-index : amélioration du br-index (8), lui-même dérivé du r-index.

Pour chaque benchmark, nous comparerons le temps d'exécution, la mémoire RAM utilisée, ainsi que la taille finale de la structure une fois stockée sur le disque. Dans cette section, tous les outils ont été lancé sur un unique thread.

Toutes les expériences ont été réalisées sur un cluster équipé d'un processeur Intel(R) Xeon(R) Gold 6130 @ 2.10GHz, de 128 Go de RAM et fonctionnant sous Ubuntu 22.04.

Nous avons dans un premier temps testé K2R sur des jeux de données de reads simulés, ayant des profondeurs de 10X à 200X et des taux d'erreur variables (de 0.1% à 10%). Pour valider nos résultats sur des données réelles, nous avons appliqué la même analyse à des séquençages d'*E. coli*. Plus précisément, nous avons sélectionné un séquençage ONT récent avec un faible taux d'erreur d'environ 3% (Accession SRR26899125), un séquençage HiFi 20kb (Accession SRR11434954) et un séquençage en mode paired-end réalisé sur HiSeq X Ten (Accession DRR395239).

Afin d'étendre notre évaluation à un génome de taille intermédiaire, nous avons inclus deux jeux de données de *C. elegans* : un séquençage ONT (Accession SRR24201716) et un séquençage paired-end HiSeq X Ten (Accession ERR10914908).

Enfin, pour évaluer l'évolutivité de K2R, nous avons sélectionné deux jeux de données humains issus du projet T2T. Les séquençages HiFi (SRX7897685, SRX7897686, SRX7897687, SRX7897688 et SRX5633451) offrent une profondeur totale de 56.8X, combinant des bibliothèques de 20 kb et 10 kb. Quant aux séquençages ONT, détaillés à l'adresse suivante : `http://github.com/marbl/CHM13/blob/master/Sequencing_data.md`, ils totalisent une profondeur de 126X avec un taux d'erreur approximatif de 6%.

3.3.1 Création de l'index : utilisation temps/mémoire

Le premier aspect à comparer est l'efficacité en temps de calcul et en mémoire RAM utilisé lors du processus. Nous avons comparé K2R aux quatre outils présentés, en reportant à chaque fois le temps wall-clock ainsi que la RAM.

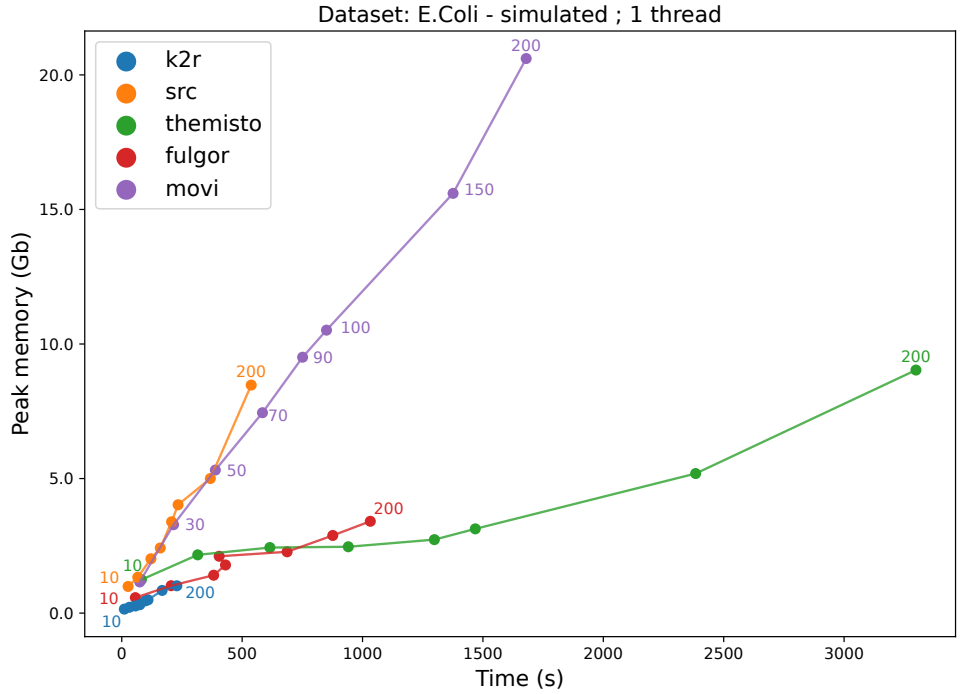
Données simulées Nous avons dans un premier temps testé sur des jeux de données simulés, afin d'avoir la main sur la taille des reads, ainsi que le taux d'erreurs et la profondeur de séquençage. Nous avons donc fait varier ces deux derniers paramètres afin de tester leur influence sur les résultats en Figures 3.26a, 3.26b, 3.27a et 3.27b.

Tout d'abord, nous avons exclu full-br-index du benchmark, car le plus petit jeu de données utilisé (une profondeur de 10X de reads simulés à partir du génome d'*E. coli*) a nécessité plus de 24 minutes de traitement, soit environ 20 fois plus longtemps que Movi et 150 fois plus longtemps que K2R. Cela suggère que full-br-index ne passe pas à l'échelle pour cette application.

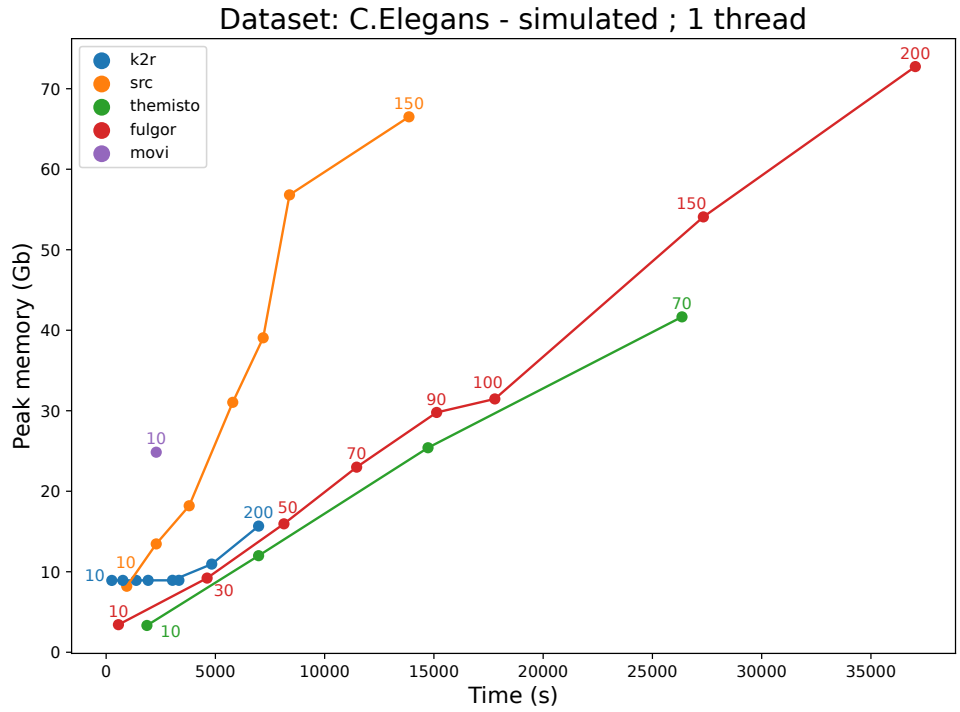
Ensuite, nous pouvons observer que K2R est nettement plus rapide et moins gourmand en mémoire que l'état de l'art, y compris sur de grandes couvertures. En revanche, Movi est bien plus lent, et ne passe pas à l'échelle sur une profondeur de 30X sur le génome *C. elegans*. Seuls K2R et fulgor sont capables d'indexer 200X de *C. elegans*.

K2R a une bonne gestion de la profondeur. Il faut dépasser une profondeur de 100X pour que le pic mémoire ne commence à augmenter légèrement. Nous pouvons nous y attendre, grâce notamment à l'utilisation des minimizers. Elle a par contre un impact important sur SRC, Themisto et Movi.

De manière similaire, le taux d'erreurs n'impacte que très peu K2R, qui reste plus efficace que l'état de l'art en faisant varier ce paramètre. Comme attendu, c'est Movi qui est le plus impacté par les erreurs, devenant notamment moins compressible.

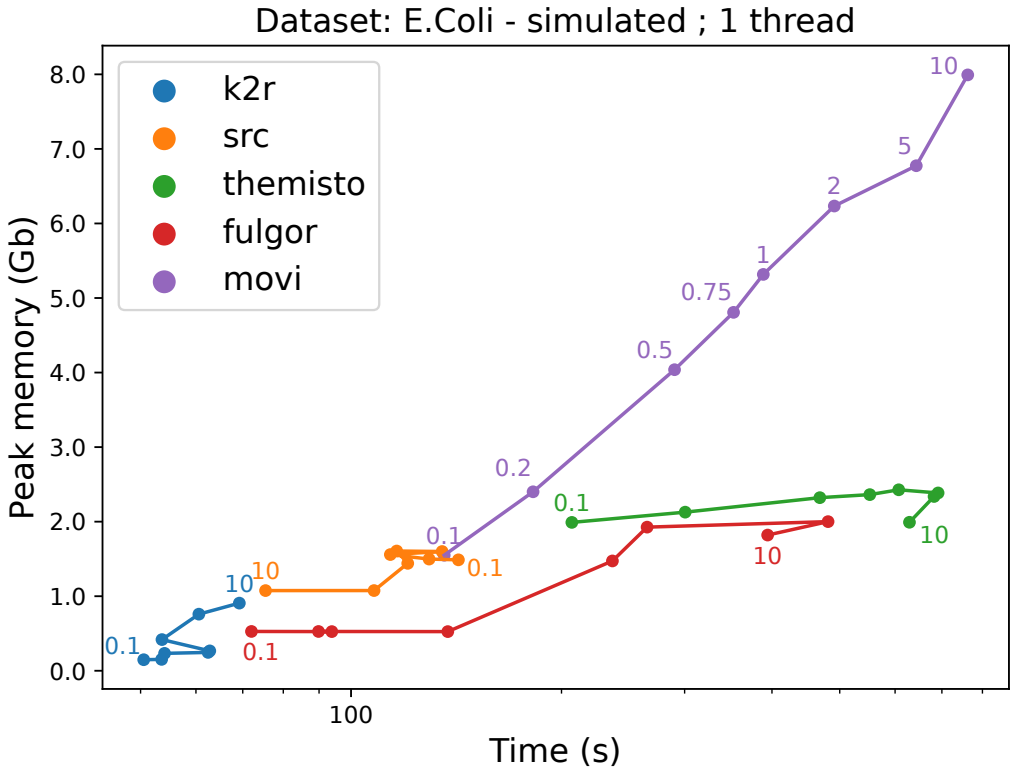


(a)

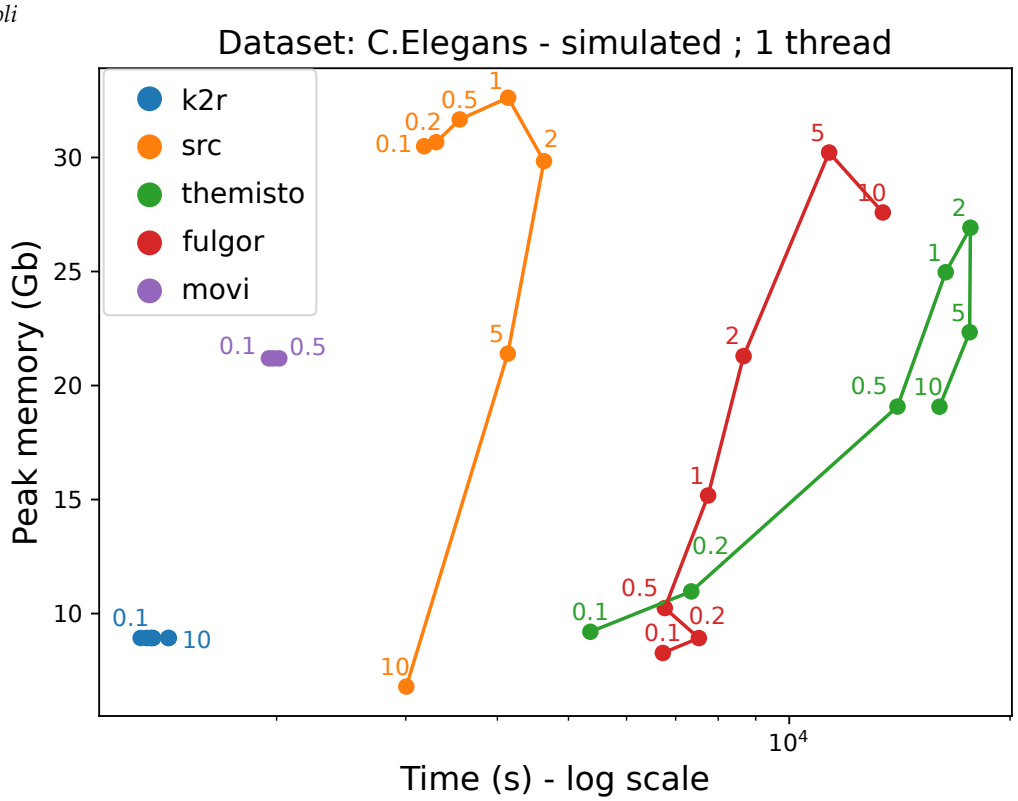


(b)

FIGURE 3.26 – *Comparaison de l'utilisation des ressources (temps *wall-clock* et pic mémoire RAM) pendant la construction de l'index avec K2R, SRC, Movi, Fulgor et Themisto, en fonction des profondeurs. Le jeu de données utilisé consiste en des reads simulés à partir des génomes (3.26a) E. coli et (3.26b) C. elegans, chacun d'une longueur de 10 000 et d'un taux d'erreur de 1%.*



(a) *E. coli*



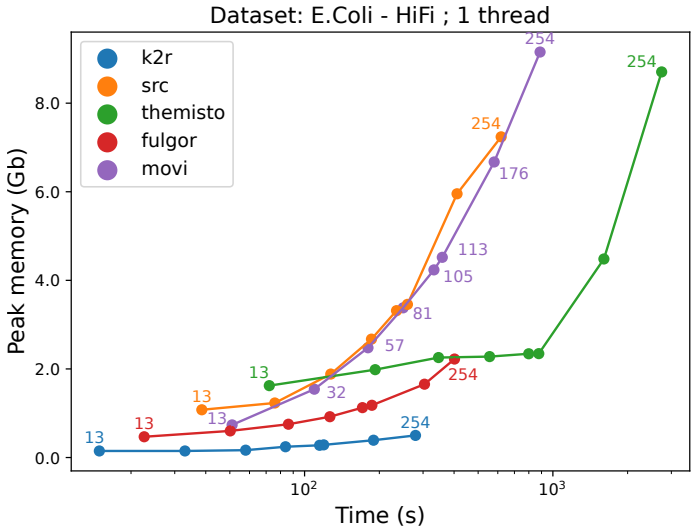
(b) *C. elegans*

FIGURE 3.27 – *Mémoire RAM et temps d'exécution wall-clock utilisés pendant la construction d'index en fonction du taux d'erreur (indiqué sur les étiquettes en pourcentage) pour des reads simulées (longueur 10 000, profondeur 50X) des génomes : 3.27a E. coli (haut) et 3.27b C. elegans (bas). Les graphiques sont en échelle logarithmique pour des raisons de lisibilité.*

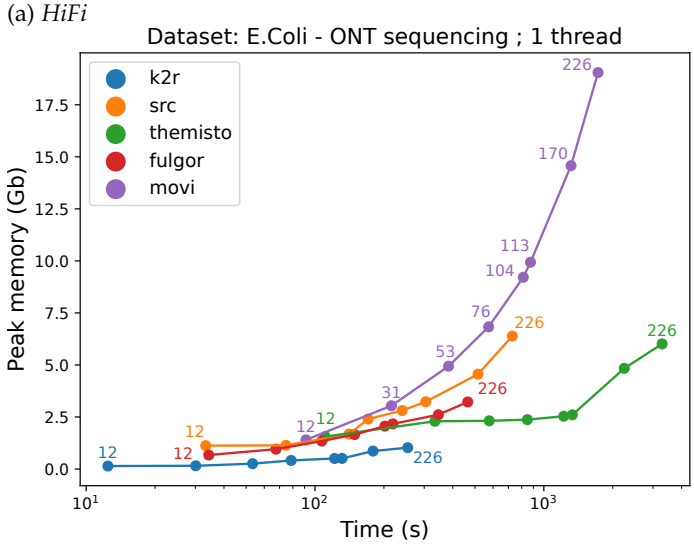
Données réelles Nous avons ensuite relancé les mêmes expérimentations sur des jeux de données réels (datasets HiFi, ONT et Illumina, pour *E. coli* et *C. elegans*. Les résultats sont présentés en Figures 3.28 et 3.29.

À l'image des données simulées, la profondeur n'a que peu d'impact sur K2R quel que soit le type de données. Le graphique concernant les données short read 3.28c est intéressant ici : on

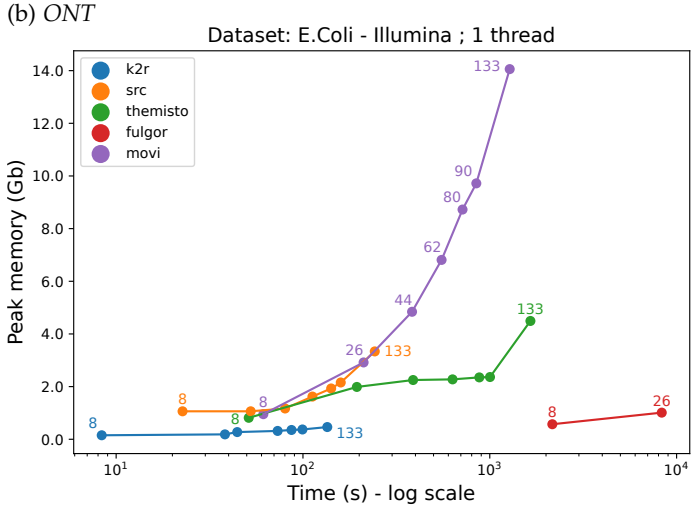
observe une meilleure efficacité de SRC, à la base créé pour ce type de données. K2R, étant pensé pour les longs reads, reste tout de même meilleur, surtout en mémoire. Fulgor, quant à lui, de par son mode d'utilisation nécessitant autant de fichiers qu'il y a de reads, explose en temps à cause du nombre de lectures de fichiers. Son utilisation est à privilégier sur des génomes entiers, moins nombreux.



(a) *HiFi*

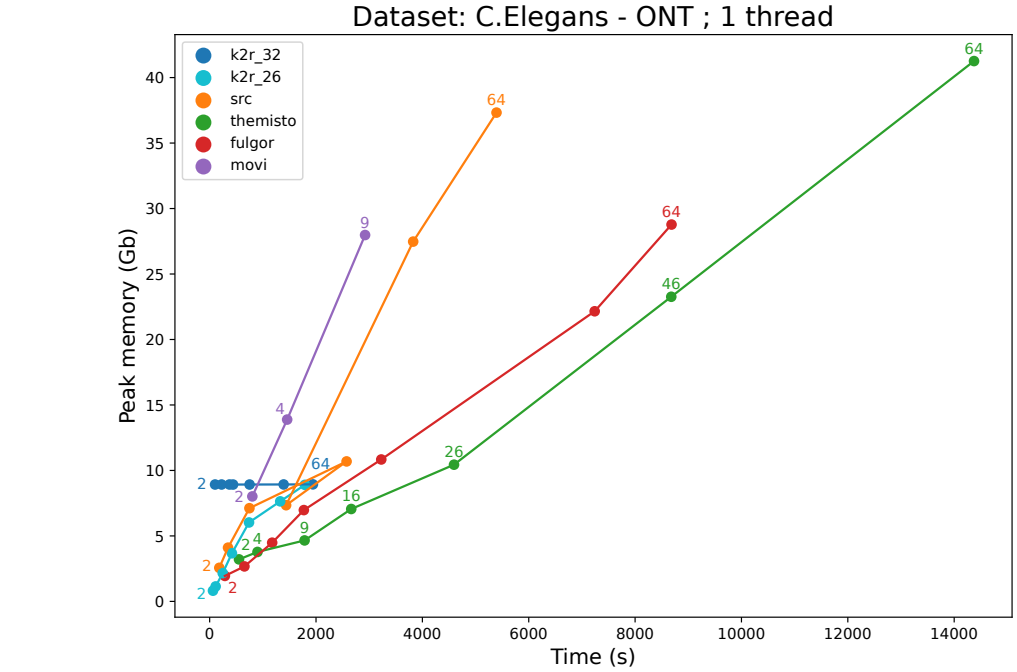


(b) *ONT*

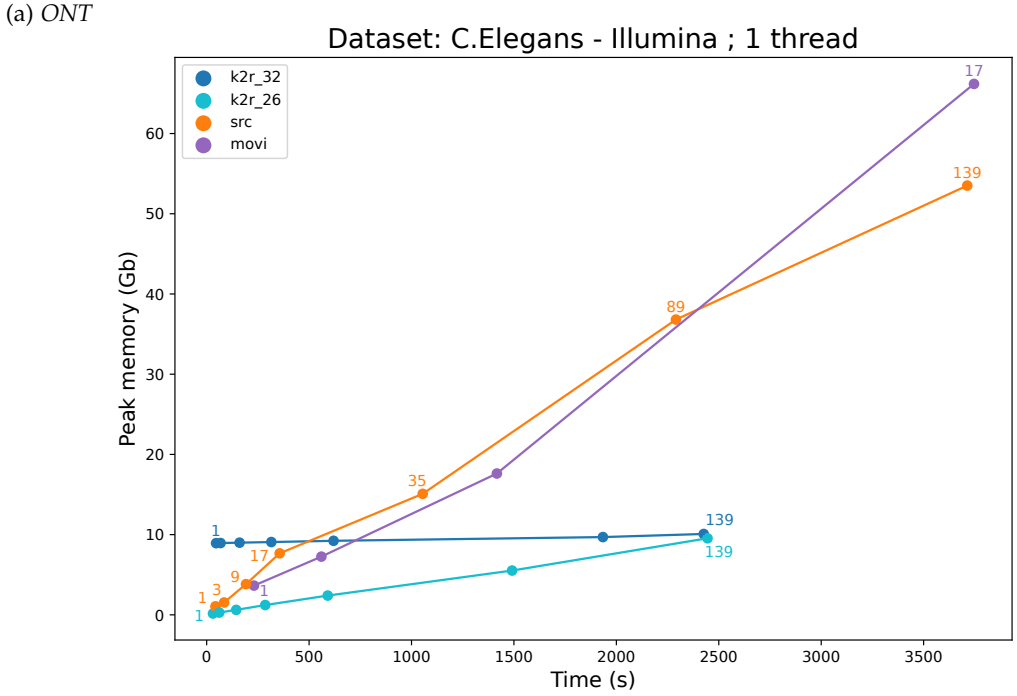


(c) *Illumina*

FIGURE 3.28 – *Mémoire RAM et temps d'exécution wall-clock utilisés pendant la construction d'index avec trois types de données présentant des profondeurs variables (indiquées sur les étiquettes) issues du génome de E. coli : 3.28a HiFi (haut), 3.28b ONT (centre) et 3.28c Illumina (bas), où l'on observe que Fulgor ne parvient pas à passer à l'échelle pour une profondeur de 44X. Les graphiques sont en échelle logarithmique pour des raisons de lisibilité.*



(a) *ONT*



(b) *Illumina*

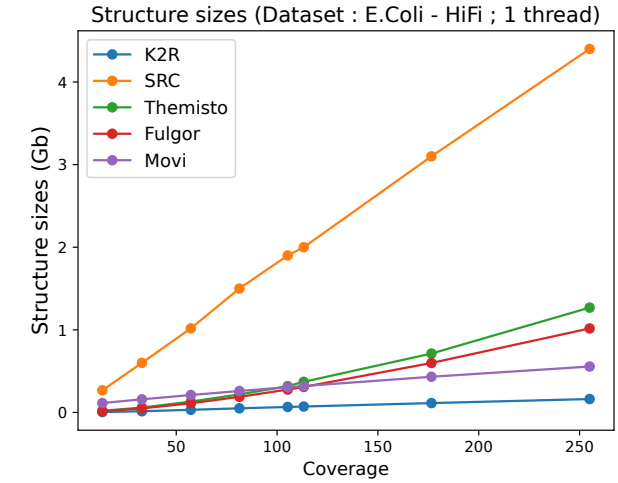
FIGURE 3.29 – *Mémoire RAM et temps d'exécution wall-clock utilisés pendant la construction d'index avec deux jeux de données issus du génome de C. elegans, présentant des profondeurs variables (indiquées sur les étiquettes) : 3.29a ONT (haut), où l'on observe que Movi ne parvient pas à passer à l'échelle pour une profondeur supérieure à 9X, et 3.29b Illumina (bas), où l'on constate que Movi ne peut pas passer à l'échelle au-delà de 17X. Fulgor ne peut pas être utilisé avec ce type de données en raison de son mode de fonctionnement. K2R 26 et K2R 32 font référence à la taille du counting Bloom filter (2^{26} ou 2^{32}) utilisées pour l'indexation.*

3.3.2 Création de l'index : taille de l'index

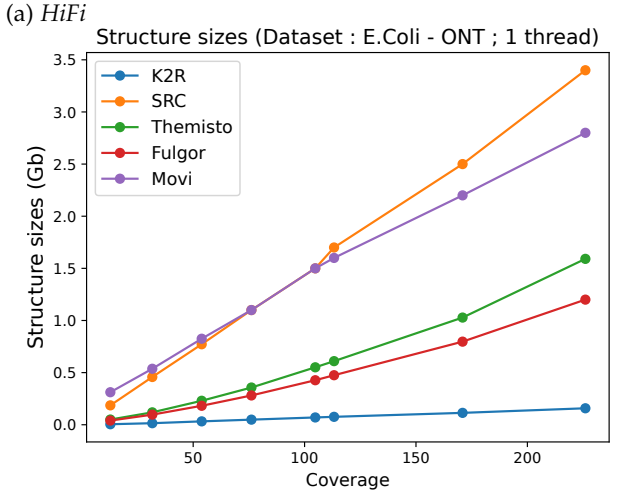
La taille de l'index stocké une fois créé est aussi un élément crucial à analyser. Un outil d'indexation utilisant peu d'espace permettra notamment de travailler sur des machines plus modestes.

Nous avons décidé pour ce cas de comparer les tailles d'index pour les jeux de données réels (HiFi, ONT et Illumina) en Figures 3.30 et 3.31, et simulés en Figures 3.32 et 3.33 pour *E. coli* et *C. elegans*, en fonction de la profondeur et du taux d'erreurs.

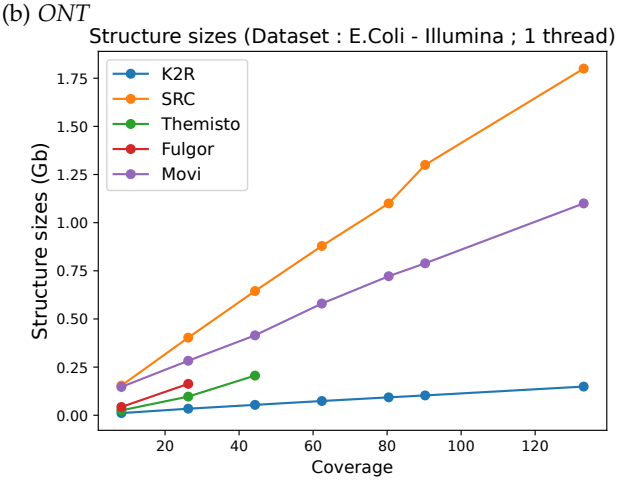
Nous voyons que dans chaque cas, le système de compression mis en place dans K2R porte ses fruits. La taille d'index est la caractéristique pour laquelle K2R domine le plus l'état de l'art. Dans le cas d'un génome un peu plus important comme *C. elegans*, nous sommes un ordre de grandeur plus légers que nos concurrents. Parmi eux, lorsque nous faisons varier la couverture, c'est SRC qui est le plus gourmand, suivi de Themisto et Fulgor. Movi quant à lui ne passe pas à l'échelle dans tous les cas, notamment pour *C. elegans*. Lorsque nous faisons varier le taux d'erreurs cependant, SRC obtient de bons résultats, jusqu'à rejoindre K2R pour le taux le plus élevé que nous ayons testé, 10%. C'est notamment dû au fait que SRC, à l'image de K2R, filtre les k-mers peu abondants.



(a) *HiFi*

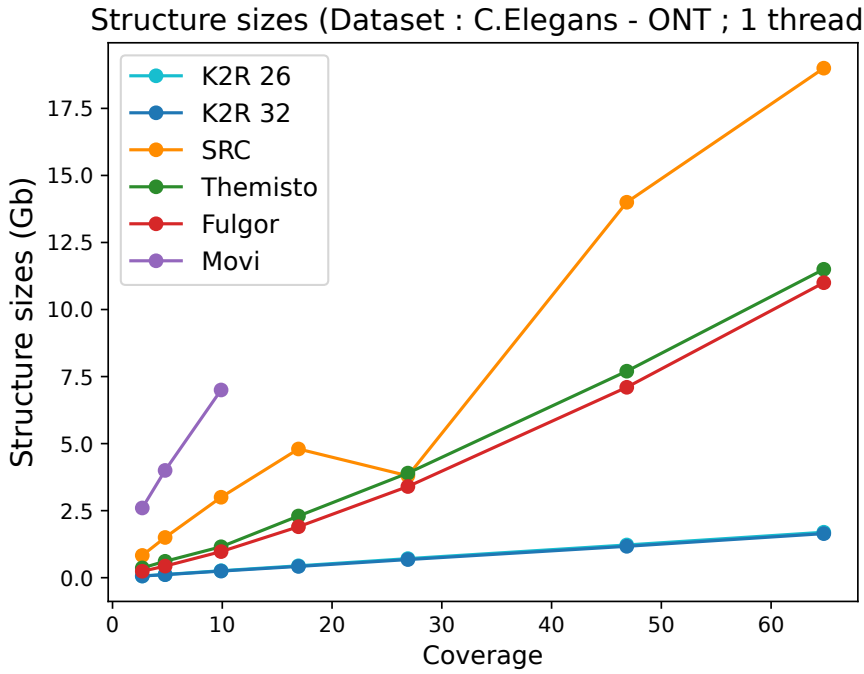


(b) *ONT*

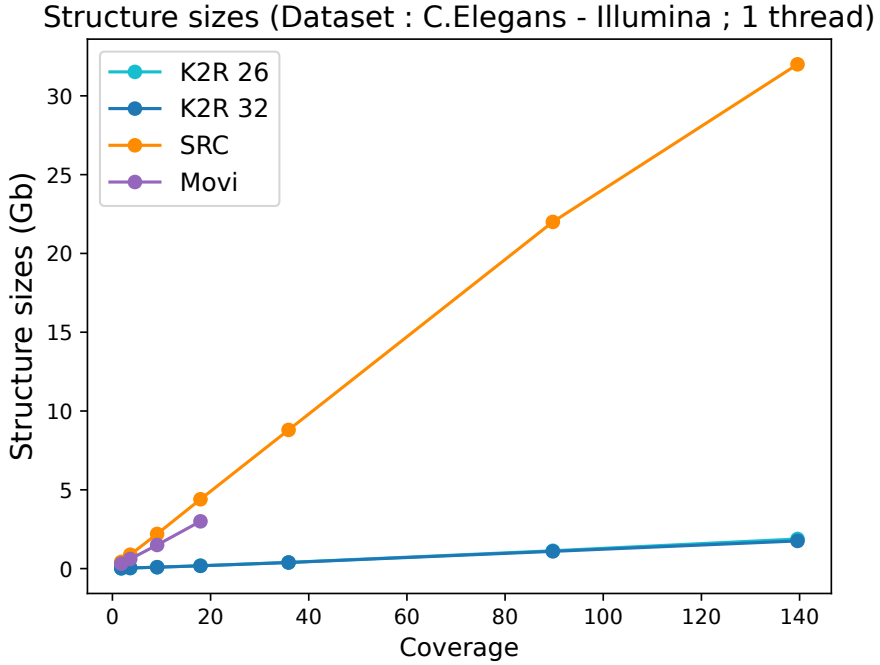


(c) *Illumina*

FIGURE 3.30 – *Comparaison de la taille de l'index pour trois jeux de données du génome d'E. coli, en fonction de la couverture : 3.30a HiFi (haut), 3.30b ONT (centre), 3.30c Illumina (bas).*

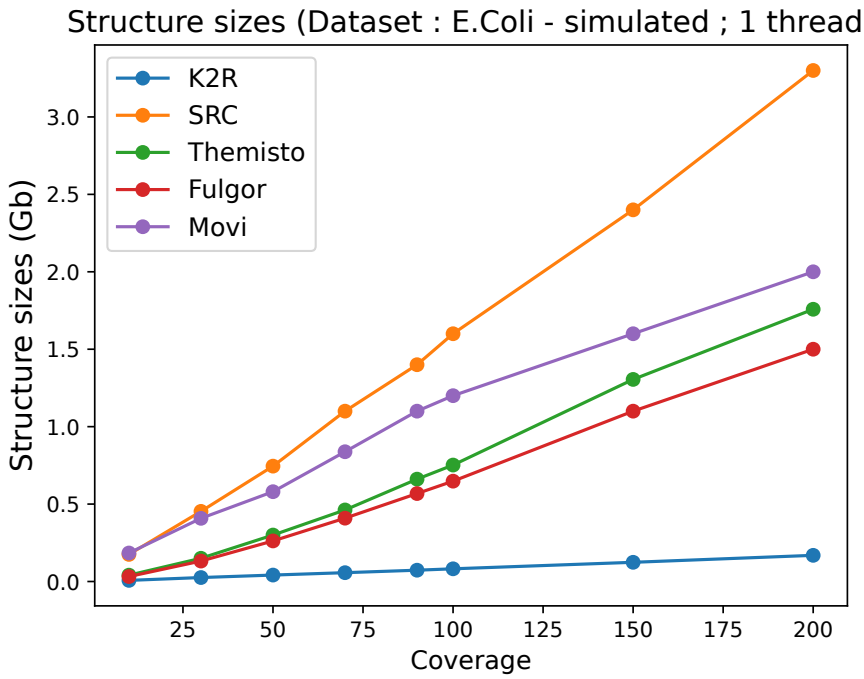


(a) *ONT*

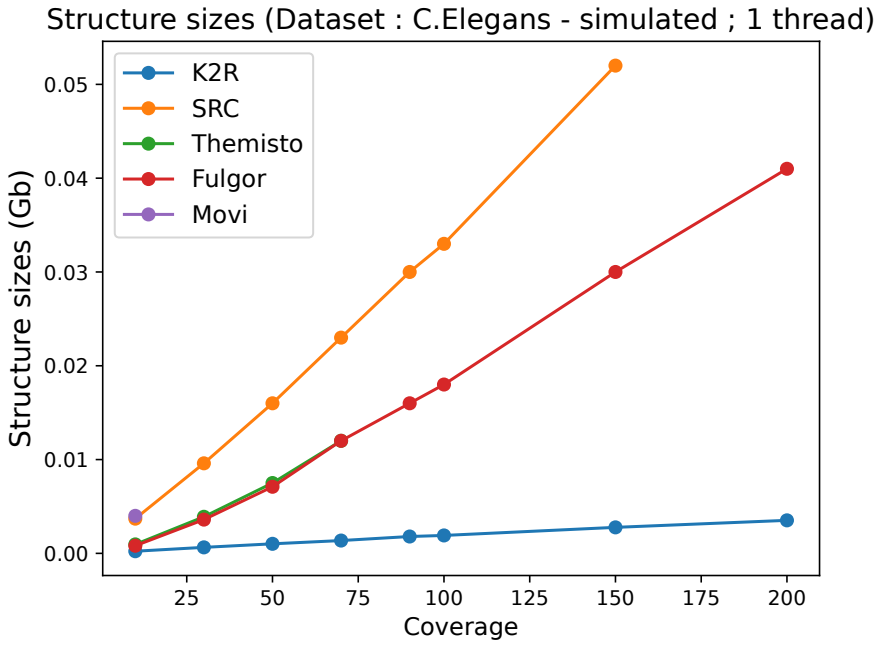


(b) *Illumina*

FIGURE 3.31 – *Comparaison de la taille de l'index pour deux types de jeux de données issus du génome de C. elegans, en fonction des couvertures : 3.31a (ONT) et 3.31b (Illumina). K2R 26 et K2R 32 font référence à la taille du counting Bloom filter (2^{26} ou 2^{32}) utilisée pour l'indexation.*

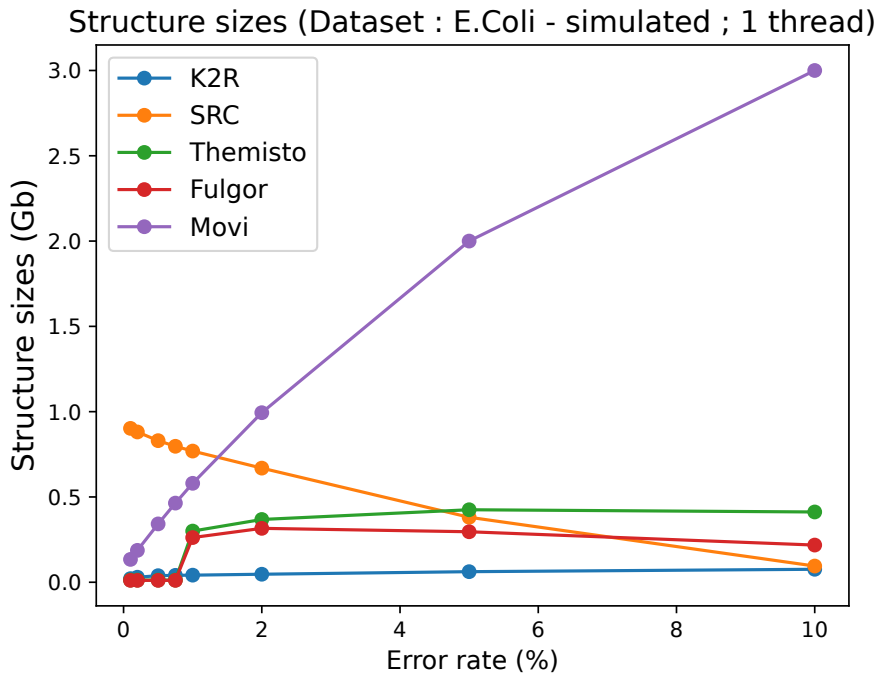


(a)

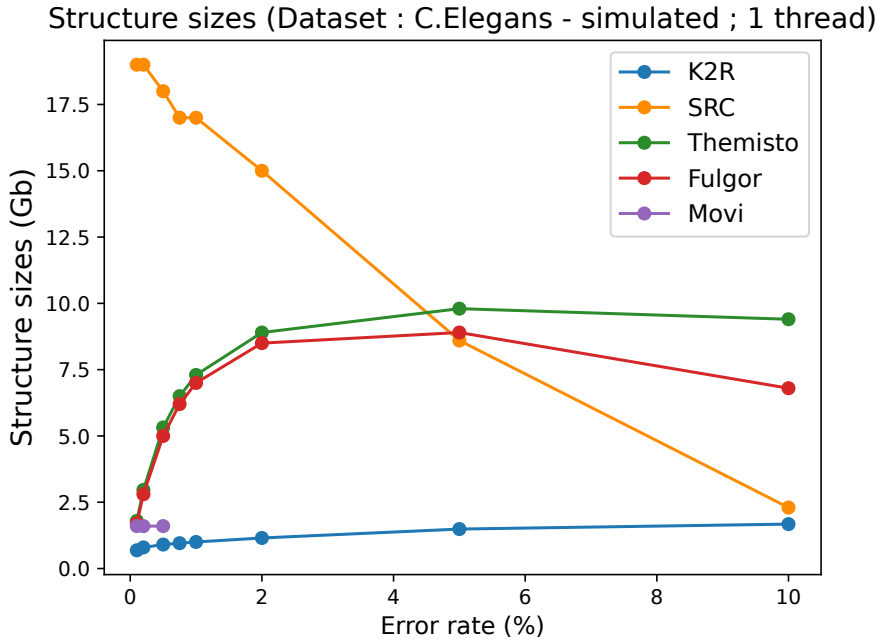


(b)

FIGURE 3.32 – *Comparaison de la taille de l'index pour différentes valeurs de couverture, à partir de reads simulés à partir des génomes de référence (3.32a) E. coli et (3.32b) C. elegans, avec un taux d'erreur de 1% et une longueur de reads de 10 000.*



(a)

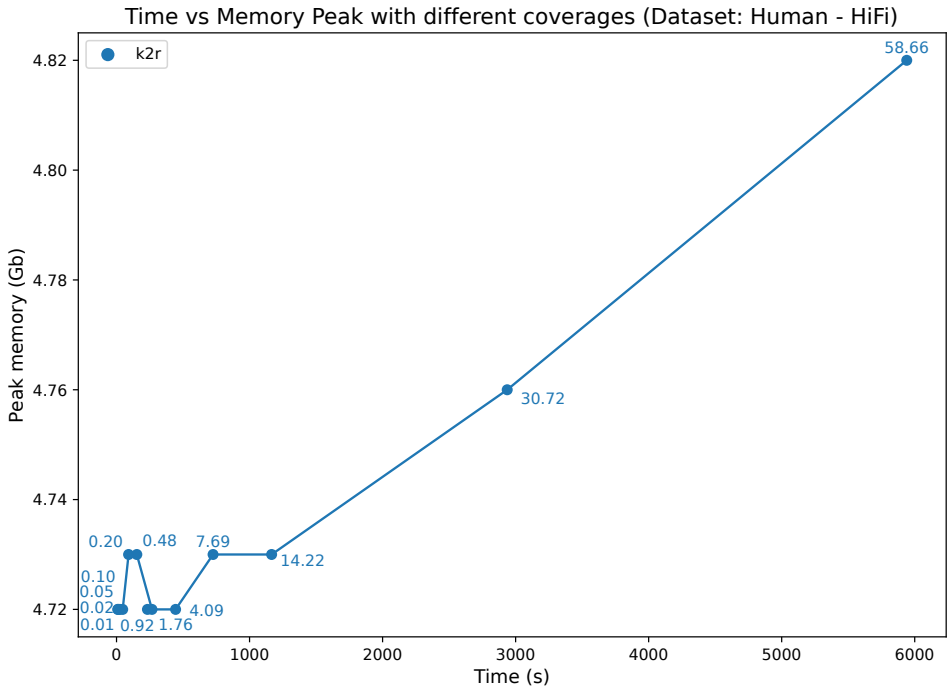


(b)

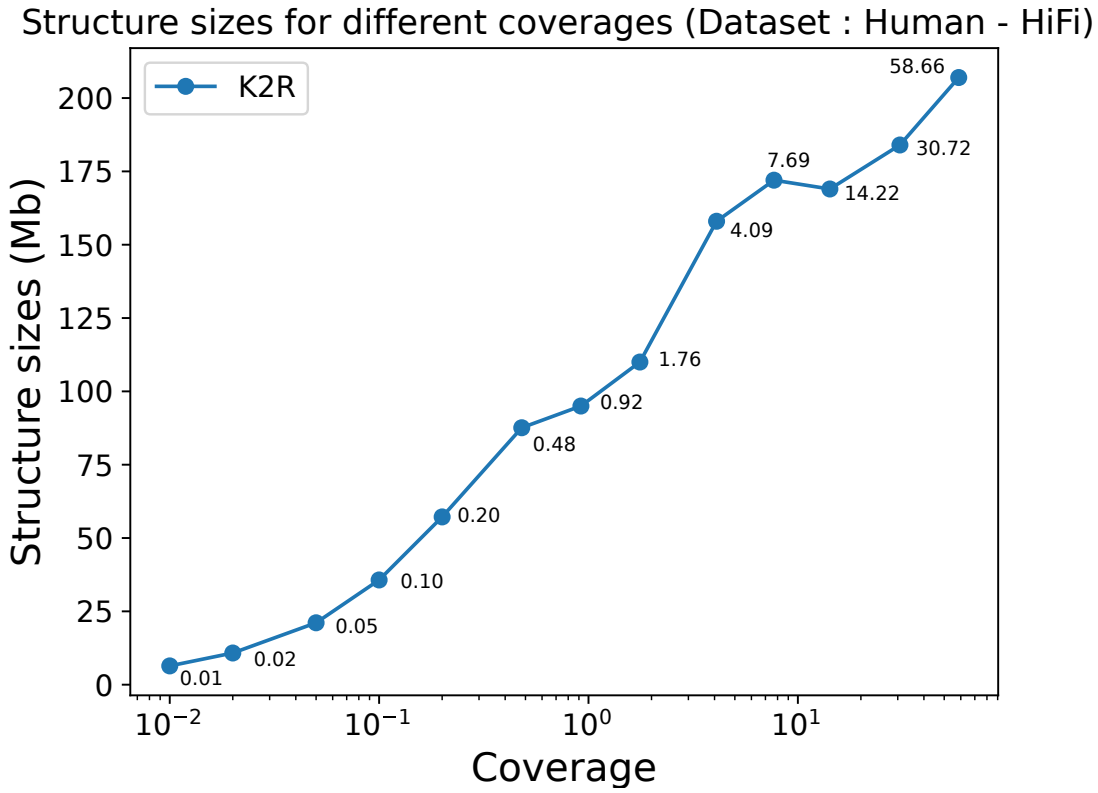
FIGURE 3.33 – *Comparaison de la taille d'index pour différentes valeurs de taux d'erreur, à partir de reads simulés à partir des génomes de référence (3.33a) E. coli et (3.33b) C. elegans, ayant une couverture de 50X et une longueur de 10 000.*

3.3.3 Génome humain

Pour conclure ce benchmark, nous avons décidé de tester nos performances sur un jeu de données humain (HiFi). Les résultats obtenus sont convaincants, avec pour une couverture de 58X un pic mémoire n'atteignant pas les 5Go, un temps raisonnable inférieur à 2 heures et une taille de structure d'environ 200Mo. À noter que pour ce jeu de données la version parallélisée de K2R a été utilisée, avec 5 threads.



(a)



(b)

FIGURE 3.34 – *Comparaison des tailles de structures, temps et mémoire utilisés pour différentes taille de couverture sur un jeu de données humain HiFi ayant une couverture totale de 56.8X.*

3.3.4 Requêtes

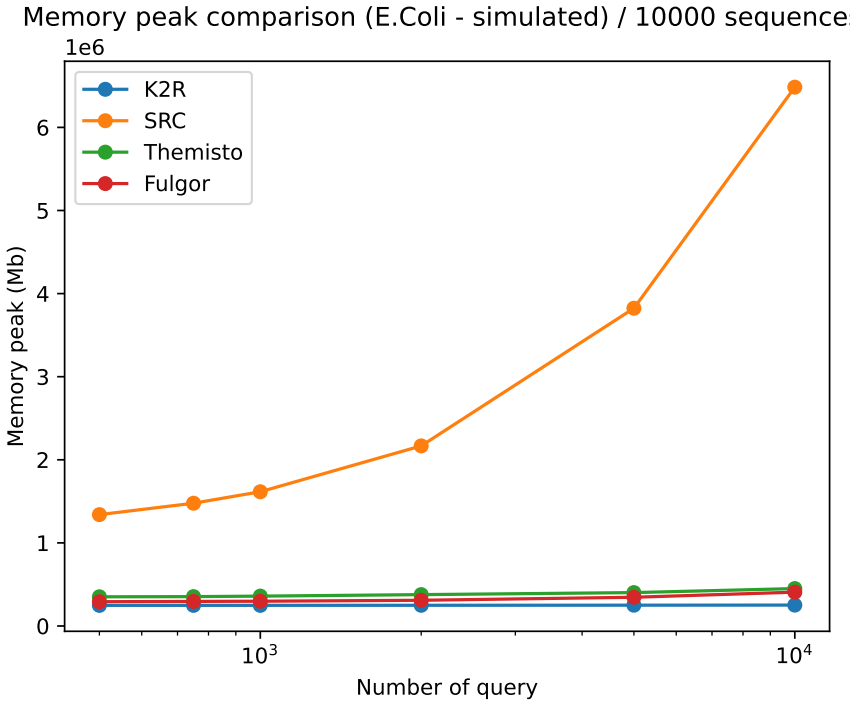
Pour finir, nous avons comparé l'efficacité (temps et mémoire) des requêtes avec l'état de l'art : Short read connector, Themisto et Fulgor. Movi a été exclu de ces résultats, car il effectue un type d'opération différent, basé sur la présence/absence et la similarité, plutôt que sur la

position des reads requêtés dans l'index. Plutôt que de mesurer le temps d'exécution pour une seule requête (ce qui serait impossible, notamment à cause des surcoûts comme le chargement des index), nous avons requêté des ensembles de séquences d'*E. coli* de taille variant entre 500 à 10.000 séquences, chacune d'entre elles d'une longueur de 10.000 bases. Deux types de requêtes ont été effectués : des requêtes positives, où les reads proviennent du génome de référence d'*E. coli*, et des requêtes négatives, où les reads ont été générés aléatoirement. Toutes les requêtes ont été exécutées avec 24 threads. Les résultats concernant les requêtes positives sont détaillés sur les Figures 3.35a pour la mémoire et 3.35b pour le temps wall-clock. Les résultats concernant les requêtes négatives sont détaillés quant à elles sur les Figures 3.36a pour la mémoire et 3.36b pour le temps wall-clock.

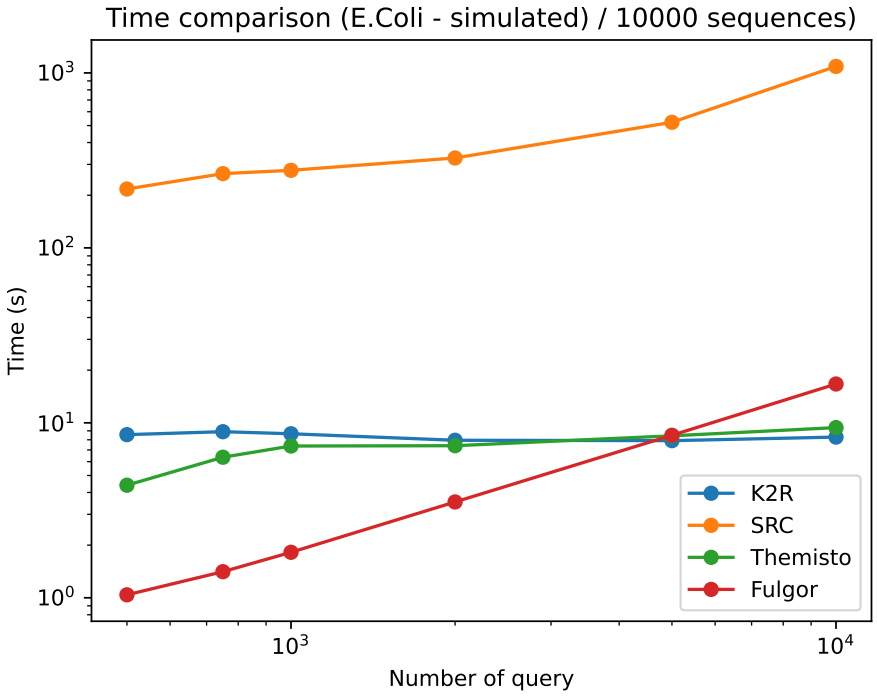
Tout d'abord, concernant l'utilisation de la mémoire RAM, K2R, Themisto et Fulgor restent constants, tandis que SRC voit son pic augmenter rapidement. On observe cette tendance sur les requêtes positives comme négatives.

En ce qui concerne le temps d'exécution, les observations diffèrent. SRC reste le plus lent d'un ordre de grandeur, voire deux pour les requêtes positives. Themisto suit, avec des performances comparables à K2R pour les requêtes positives, mais moins bon pour les requêtes négatives. K2R arrive ensuite, puis Fulgor qui se distingue comme le plus rapide dans ces tests, notamment en query négatives. Pour les query positives, la tendance s'inverse pour un nombre de requêtes plus important (à partir de 5.000 séquences).

En pratique, K2R, Themisto et Fulgor ont des différences de performances minimes (tous utilisent moins de 10 secondes dans tous les cas), ce qui les rend tous les trois utilisables et efficaces.

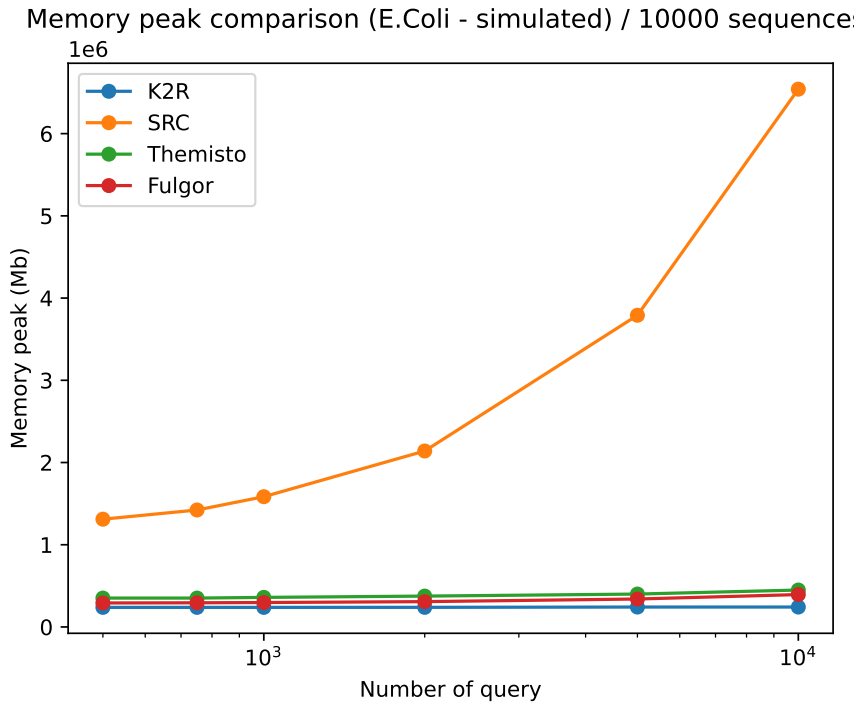


(a)

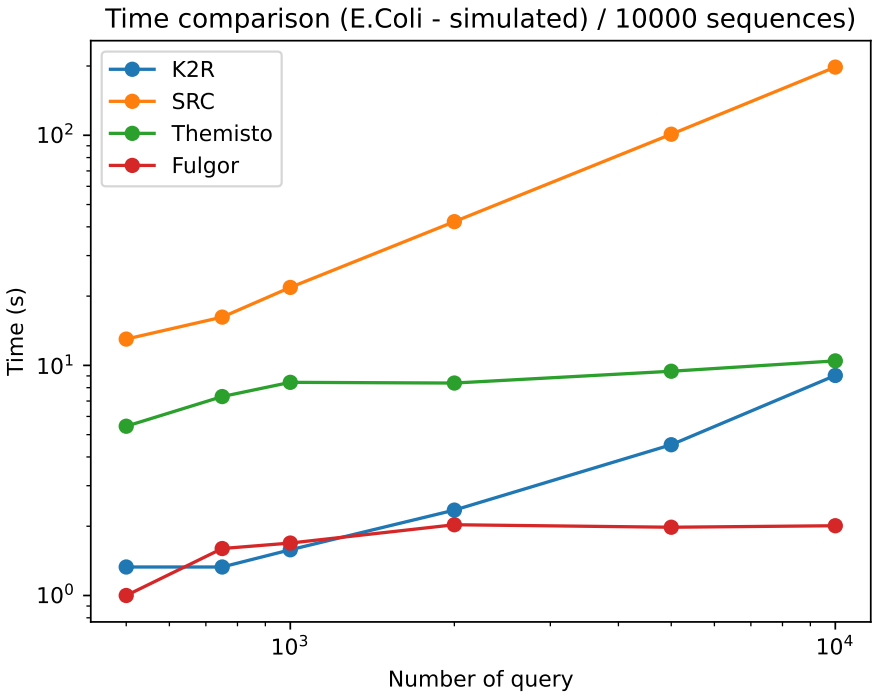


(b)

FIGURE 3.35 – *Comparaison du temps d'exécution (wall-clock) et du pic mémoire pour différents nombres de séquences requêtées (requêtes positives). L'index requêté est créé à partir de reads E. coli simulés de couverture 50X et ayant un taux d'erreur de 1%.*



(a)



(b)

FIGURE 3.36 – *Comparaison du temps d'exécution (wall-clock) et du pic mémoire pour différents nombres de séquences requêtées générées aléatoirement (requêtes négatives). L'index requêté est créé à partir de reads E. coli simulés de couverture 50X et ayant un taux d'erreur de 1%.*

En résumé, notre analyse aboutit à trois conclusions clés : Premièrement, K2R se distingue par son efficacité en construction, réduisant de manière significative l'utilisation de mémoire et de temps. Deuxièmement, et c'est peut-être le plus important, K2R crée des index moins lourd à stocker que ceux des outils de référence actuels, avec des tailles inférieures parfois de plusieurs ordres de grandeur, quelle que soit la couverture ou le taux d'erreur. Troisièmement, les requêtes effectuées avec K2R sont rapides et tout aussi économes en mémoire que celles exécutées avec Themisto et Fulgor.

Ces résultats prouvent l'efficacité concrète des graphes de de Bruijn teintés à grande échelle, et soulignent l'efficacité de l'implémentation que nous proposons avec K2R.

3.4 Influence des paramètres

Un nombre assez important de paramètres peuvent être choisis par l'utilisateur en utilisant K2R. Cependant, ces paramètres auront un impact important sur l'index créé, notamment en termes de nombre de minimizers indexés, et donc de potentiels faux positifs à gérer dans les requêtes. Ils auront aussi, bien entendu, un impact sur le temps et la mémoire utilisés.

Dans cette section nous allons donc évaluer ces différents impacts.

3.4.1 Taille des k-mers

Le choix de la taille des k-mers est un paramètre important pour la construction de l'index. Intuitivement, des k-mers plus courts sont plus redondants car le nombre de combinaisons possibles diminue avec k. Aussi, la quantité de k-mers par read augmentera très légèrement ($l - k + 1$ k-mer par read, l étant la longueur du read). Sur des reads longs on peut s'attendre à un impact limité, mais sur des reads courts ce détail doit être pris en compte.

On peut vérifier que dans la Figure 3.37, entre $k = 17$ et $k = 31$, le nombre de minimizers indexés baisse drastiquement. Cette baisse ralentit par la suite.

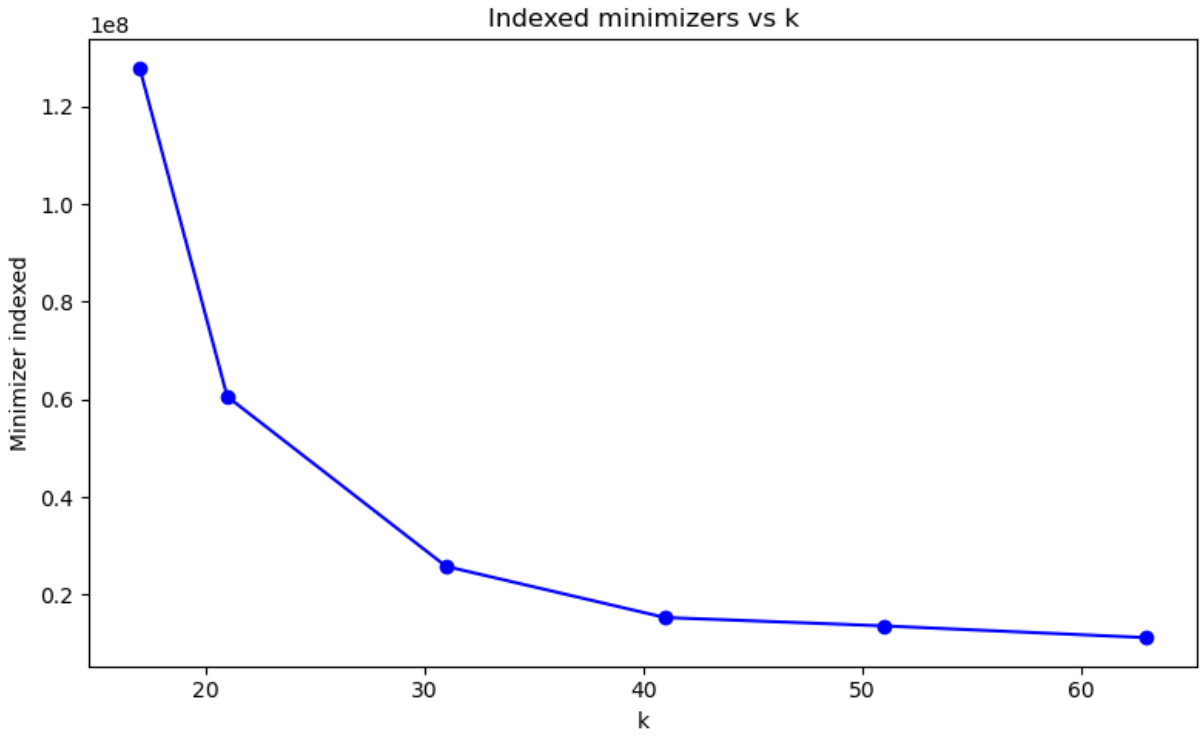


FIGURE 3.37 – *Impact de la taille de k sur le nombre de minimizers stockés. L'impact entre k = 17 et k = 31 est important (nombre de minimizers divisé par 6), puis l'impact se réduit.*

3.4.2 Taille des minimizers

L'utilisation de minimizers à la place de k-mers soulève naturellement une question : dans quelle mesure les faux positifs introduits par cette méthode influencent-ils les requêtes ? Pour y répondre, nous avons réalisé une évaluation illustrée dans la Figure 3.38. Nous y mesurons

la proportion de reads sélectionnés, par rapport au nombre total de reads atteignant le seuil requis de similarité en *k*-mers (fixé ici à 0.6, avec des tailles de minimizers allant de 11 à 21). Sans surprise, nous pouvons observer qu'une taille plus grande de minimizer améliore ce ratio et donc diminue le nombre de faux positifs, indiquant que la plupart des reads sélectionnés sont bel et bien pertinents par rapport à la requête. En effet, des minimizers plus grands sont moins fréquents, et réduisent les collisions. À noter que les collisions peuvent également être dues au choix de la taille du filtre de Bloom ainsi qu'aux fonctions de hachage associées.

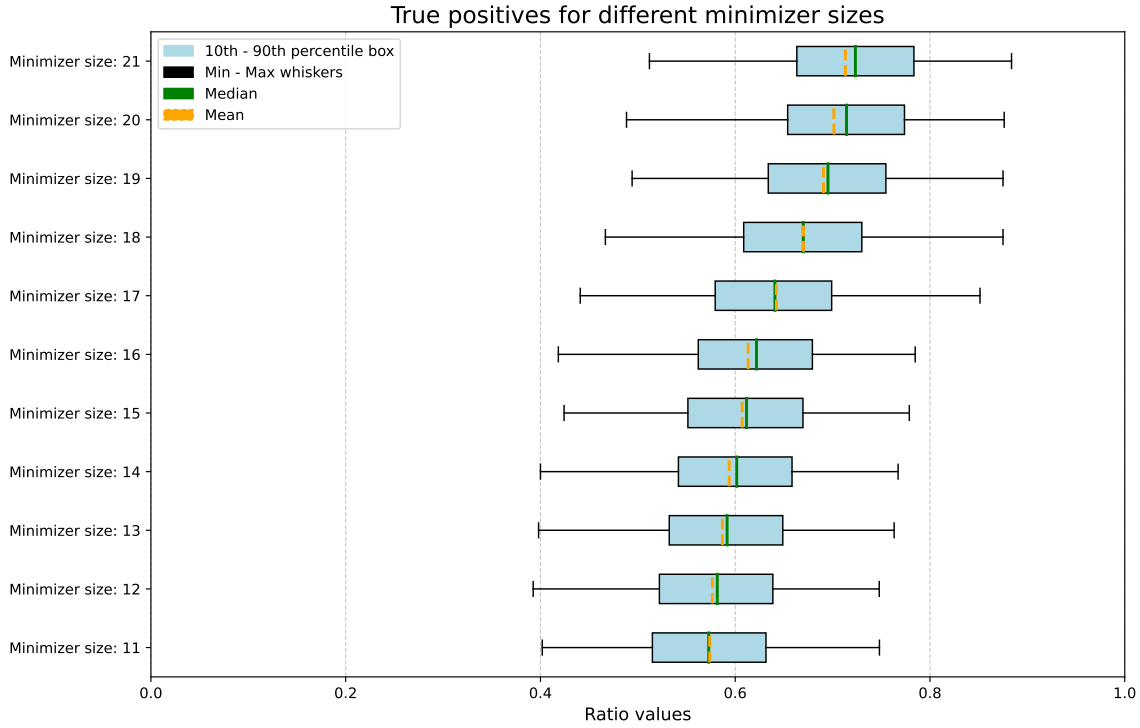


FIGURE 3.38 – *Impact de la taille des minimizers sur le nombre de faux positifs dans les requêtes. Une taille plus grande de minimizer améliore ce ratio et donc diminue le nombre de faux positifs.*

3.4.3 Taille de counting bloom filter

Le counting Bloom filter dans K2R permet non seulement de stocker les valeurs de hash des minimizers, mais également de compter leur occurrence dans le but de filtrer les éléments trop fréquents ou au contraire trop peu fréquents. Étant une structure à taille fixe, et plusieurs *k*-mers pouvant avoir le même minimizer, des faux positifs peuvent apparaître à cette étape. Intuitivement, on peut deviner qu'un filtre plus petit provoquera des collisions, tandis qu'un filtre plus large les évitera.

On confirme cette intuition dans la Figure 3.39 montrant en fonction de la taille du filtre (allant de 2^{18} à 2^{32}) le nombre de minimizers uniques et le nombre de minimizers total indexés. Ces courbes sont logiquement liées. Dans le cas de notre dataset *C. elegans* 100X de profondeur, les valeurs clés de changement sont les tailles de 2^{24} et 2^{26}. Pour des tailles plus faibles, les collisions sont trop importantes et très peu, voire aucun minimizer unique n'apparaît. Au fur et à mesure que la taille est augmentée, les valeurs se stabilisent en fonction de la taille du jeu de données et du nombre réel de minimizers. Par exemple ici, pour notre taille maximale de 2^{32} qui est donc la plus "réaliste" (moins de collisions), autour de 25 millions de *m*-mers sont stockés et environ de 16 millions sont uniques et donc ignorés.

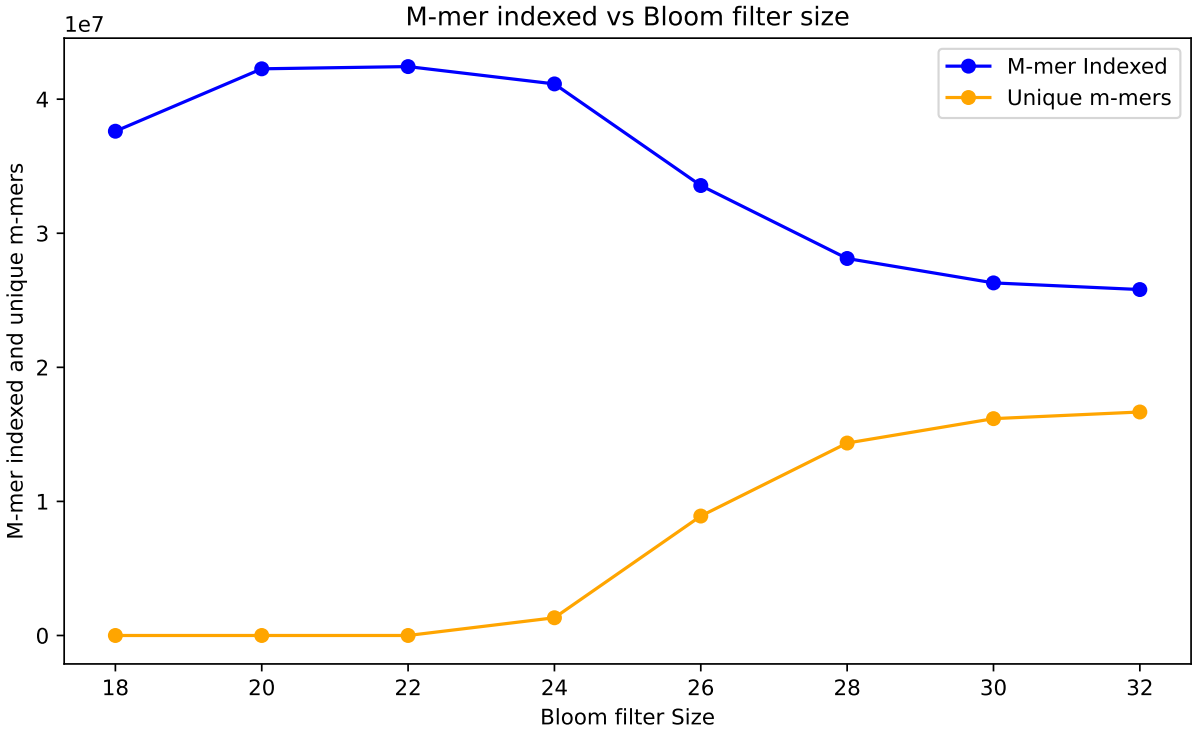


FIGURE 3.39 – *Impact de la taille du filtre de Bloom sur le nombre de minimizers indexés. Le jeu de données indexé consiste en un jeu de reads de longueur 10.000 simulé à partir du génome de C. elegans, ayant une profondeur de 100X et un taux d'erreurs de 1%. Dans le cas de ce dataset, les valeurs clés de changement sont les tailles de* 2^{24} *et* 2^{26}.

3.5 Discussion et perspectives sur K2R

Nous avons ainsi conçu un index capable de passer efficacement à l'échelle, tout en garantissant des performances élevées en termes de vitesse et de justesse lors des requêtes. Comparé aux approches actuelles de l'état de l'art, notre méthode démontre des gains significatifs, autorisant une utilisation sur une large variété de jeux de données, tant en termes de nature que de volume. Ces résultats laissent entrevoir un fort potentiel d'applicabilité dans des domaines tels que la quantification en transcriptomique ou métagénomique, l'assemblage ou encore le génotypage.

L'outil propose en outre de nombreux paramètres configurables, permettant une adaptation fine aux besoins spécifiques de l'utilisateur. En particulier, le filtrage fondé sur l'abondance des *k*-mers se révèle crucial dans le cas de reads courts, où il contribue au bon fonctionnement de la structure. Toutefois, ces paramètres influent directement sur la structure de l'index : leur manipulation nécessite donc une compréhension précise de leurs effets afin d'en optimiser les performances selon les contextes d'usage.

Malgré les résultats obtenus, plusieurs axes d'amélioration restent envisageables pour K2R. Un premier point concerne, comme énoncé précedemment, l'indexation des données issues de reads courts. Bien que la structure soit compétitive dans ce contexte, elle nécessite un filtrage des minimizers selon leur abondance. En effet, des listes trop volumineuses d'identifiants de reads ne permettent pas de passer à l'échelle, notamment en raison des limites imposées par le schéma de compression utilisé (TurboPFor).

Actuellement, K2R se limite à l'indexation d'un seul jeu de données à la fois. Une extension naturelle consisterait à permettre la fusion de plusieurs index, afin de faciliter les requêtes sur des jeux de données multiples et d'enrichir les scénarios d'utilisation.

Par ailleurs, les index produits ne sont pas réversibles : il n'est pas possible, à partir d'un index, de reconstruire les séquences initiales. Cela s'explique notamment par la redondance des *k*-mers, aussi bien à l'intérieur d'un même read qu'entre plusieurs reads, ainsi que par le fait qu'un même minimizer peut être partagé par plusieurs *k*-mers, rendant leur attribution ambiguë. Une adaptation structurelle serait donc nécessaire pour permettre une réversibilité partielle ou complète, en conservant notamment des informations sur la position exacte de chaque élément dans les séquences d'origine.

Il serait également intéressant d'inclure un prétraitement des données, comme la correction des reads. Cette étape permettrait de réduire le nombre de *k*-mers et, par conséquent, de minimizers à stocker. Le nombre de couleurs serait également diminué, car des reads corrigés successifs partageraient davantage de minimizers, évitant ainsi la création de couleurs contenant un nombre limité d'identifiants. Cela aurait plusieurs impacts : réduction de l'usage de la mémoire RAM grâce à un traitement moins lourd, diminution de l'espace disque nécessaire pour la structure finale, et gain de temps lors de l'indexation et des requêtes.

Ces limitations et idées constituent autant de pistes de travail qui pourraient prolonger et enrichir le développement de K2R, en renforçant sa capacité à répondre à des besoins variés dans le domaine de l'indexation génomique.

Un article a été publié concernant K2R (122), reprenant l'ensemble des méthodes et résultats présentés ci-dessus, publié dans le journal *Bioinformatics Advances*.

Chapitre 4

Stratégies complémentaires

K2R est né d'un besoin atomique, commun à de nombreuses applications : associer efficacement les *k*-mers aux reads qui les contiennent dans un contexte de longs reads, afin de localiser des séquences caractéristiques, de détecter des variabilités d'expression ou encore de faciliter certaines étapes d'assemblage. Nos résultats montrent qu'il est possible de proposer des requêtes exactes avec un très fort débit (plus de 1.200 séquences de longueur 10.000 requêtées par seconde), tout en conservant des temps de construction et une consommation de mémoire vive compatibles avec de grands jeux de données (15 minutes, 10Go de mémoire disque et 40Go de RAM pour l'indexation d'un jeu de reads HiFi issus du génome humain), ce qui rend K2R directement utilisable dans plusieurs scénarios pratiques. Bien que ces performances restent encore optimisables, elles ne constituent pas, en l'état, un goulot d'étranglement pour un certain nombre d'applications envisagées.

Cependant, pour des applications multi-échantillon à large échelle, un point faible persiste : la taille de l'index sur disque. Dans le cadre d'un fichier unique, cette taille demeure raisonnable. En revanche, lorsqu'il s'agit d'indexer de vastes collections de jeux de données, le coût de stockage devient rapidement prohibitif, comme on peut l'anticiper en extrapolant aux volumes de séquençage humain ou à des projets de type LOGAN. Si on souhaitait par exemple indexer l'ensemble des séquençages humains, cela représenterait au minimum 60.000 To (10 Go par index, 6 millions de séquençages en tout), soit approximativement la taille de SRA (Sequence Read Archive).

Dans un premier temps, nous avons cherché des moyens de pallier cette limitation, ce qui a motivé la conception d'outils complémentaires visant soit à mieux exploiter la structure de données, soit à modifier le paradigme d'indexation pour s'adapter à d'autres cas d'usage.

Dans un premier temps, nous nous sommes intéressés à la partie la plus coûteuse de l'index : le stockage des couleurs, c'est-à-dire la représentation compacte des ensembles d'identifiants de reads associés à chaque *k*-mer. Le schéma de compression utilisé par K2R étant basé sur le *delta encoding*, il est particulièrement efficace lorsque les identifiants de reads formant une couleur sont proches les uns des autres. De manière générale, même en utilisant le delta encoding, cette composante domine largement la taille totale de l'index (jusqu'à 96% lors de l'indexation d'un jeu de données ONT issu du génome humain, 97% pour des données HiFi). Cette observation nous a conduits à chercher un moyen de réordonner les reads afin de rapprocher ceux qui se ressemblent, de façon à maximiser l'efficacité de la compression différentielle. Un tel réordonnancement est possible car l'ordre des reads n'a pas de sémantique biologique et n'est pas censé impacter les analyses en aval. C'est dans cette perspective qu'a été conçu **OReO** (Optimizing Read Order) (48), un outil dédié au réordonnancement des reads dont l'objectif est de réduire la

taille des couleurs et, par conséquent, l'empreinte mémoire et disque de K2R. Cet outil ainsi que ses applications est présenté en Section 4.1.

Même en optimisant ainsi la taille de l'index final, indexer de très grandes collections reste néanmoins difficilement envisageable. Pour certains scénarios, il est plus pertinent d'inverser la logique d'indexation : plutôt que d'indexer l'ensemble des reads ou des collections, on choisit d'indexer uniquement les séquences requêtes, puis de parcourir les reads pour identifier les correspondances, comme cela se faisait classiquement avant la généralisation des grands index en mémoire. Une approche naïve fondée sur un dictionnaire de k-mers est déjà utilisée dans l'outil Back to Sequences (BTS) (11), qui associe les k-mers aux identifiants de séquences. En reprenant les idées de K2R, par exemple via un filtrage des séquences fondé sur leurs minimizers, il est toutefois possible d'accélérer ce type de démarche. Ces réflexions ont conduit au développement de **K2Rmini** (88), présenté en Section 4.3, une variante plus rapide que BTS proposant des capacités similaire à K2R en indexant les requêtes au lieu des jeux de données eux-mêmes. L'interêt de cette méthode réside dans le fait que l'on évite de construire et de maintenir des index coûteux pour un grand nombre de jeux de données, ce qui le rend mieux adapté aux contextes où le nombre de jeux de données d'intérêt est élevé ou lorsque les requêtes sont trop rares pour justifier un index auxiliaire dédié.

Enfin, indépendamment de ces questions de passage à l'échelle, de nombreuses applications exigent des comparaisons entre reads. Les approches existantes, à l'image de K2R, ont un coût quadratique, chaque read étant potentiellement comparé à tous les autres. En nous appuyant sur des travaux antérieurs (NIQKI (1)) visant à optimiser ce type de requêtes, nous nous sommes tournés vers des méthodes de *sketching*, qui consistent à sélectionner un ensemble de k-mers de taille fixe pour construire une représentation compacte stockant le contenu d'une séquence de manière approximative mais contrôlée. Un tel résumé permet de réaliser des comparaisons plus rapides et moins gourmandes en mémoire, au prix d'un compromis explicite entre précision et efficacité. C'est dans ce cadre que nous avons conçu un index inversé plus flexible, appelé **ONIKA** (57), adapté à la comparaison à grande échelle de reads ou d'autres séquences. Cet index est présenté en détails dans la Section 4.2

Dans la suite de ce chapitre, nous présentons ces trois outils développés en prolongement de K2R : OReO, K2Rmini et ONIKA, en détaillant leurs principes de conception, leurs liens avec la structure initiale, ainsi que les résultats expérimentaux qui illustrent leur efficacité et leur complémentarité vis-à-vis de K2R.

Ces trois outils ont fait l'objet d'une publication ou d'un preprint : OReO a été publié dans le journal *Bioinformatics Advances* en juin 2025, K2Rmini et ONIKA ont chacun fait l'objet d'un preprint respectivement en juin et novembre 2025.

4.1 Traitement des données en amont : OReO

En complément de l'optimisation de K2R en lui-même, une autre piste à explorer consiste en un traitement des données en amont, afin de favoriser les performances lors de l'indexation. En particulier, dans notre cas, nous savons qu'en termes de mémoire la compression des couleurs réalisée par TurboPFor joue un rôle crucial avec le delta encoding. Plus les identifiants de reads sont proches, plus la compression sera efficace. Dans cet exemple, elle est particulièrement efficace pour les entiers 558 et 560, séparés d'une différence de seulement 2.

$$[1, 5, 6, 558, 560] \rightarrow [1, 4, 1, 552, 2]$$

Pour tirer un maximum de profit de cette méthode, nous aimerions donc rapprocher au maximum les identifiants de reads contenus dans les listes à compresser. Afin d'amener la preuve de concept que nous avons mise en place pour répondre à cette problématique, une mise en contexte sur les outils de compression utilisés actuellement est nécessaire et sera présentée ici.

Trois principales catégories de méthodes existent pour la compression de données génomiques :

— Les compresseurs génériques optimisés : se basent sur des techniques de compression connues mais s'adaptent aux caractéristiques des séquences génomiques, à l'exemple de Fastqz (19) qui utilise la compression PAC ou encore BIND (20), DELIMINATE (91) et DSRC (39) qui utilisent le LZ77.

— Les méthodes basées sur référence : l'alignement des reads à réordonner est une solution si une référence proche des reads est disponible. Par exemple, Gencompress (35), CRAM (55) Fastqz ou encore FQZip (134) mappent chaque read sur une référence et ne gardent que les variations.

— Les méthodes de réordonnancement : tirent profit du fait que si des reads similaires ou se chevauchant se retrouvent proches dans les données, alors la compression peut être réalisée de manière plus "locale", sans obligatoirement garder un contexte global. De nombreux outils implémentent ce concept, comme Coil (127) et recoil (130) qui cherchent des overlaps entre suffixes et préfixes, ou encore BEETL (32) qui optimise l'ordre des reads pour des compressions basées sur la BWT.

Par la suite, des outils ont été développés pour répondre à la problématique des longs reads, beaucoup plus bruités à l'origine mais dont le taux d'erreur tend à baisser au cours du temps. NanoSpring (90) crée un index MinHash des données qui permet de regrouper les données proches pour ensuite créer un graphe consensus. CoLoRd (65) quant à lui recherche les similarités entre les k-mers des reads, dans le même but de les regrouper.

Des dizaines d'outils sont en réalité disponibles pour réaliser des compressions de données de séquencage. Cependant, leur impact reste limité car les utilisateurs se tournent presque automatiquement vers des compresseurs génériques comme gzip ou zstd pour leur facilité d'utilisation et leur compatibilité. Dans le cas d'une éventuelle décompression plus tard, il est necessaire que les outils soient maintenus, ce qui est le cas des compresseurs génériques. Au lieu de développer un nouveau compresseur, nous nous sommes tournés plutôt sur une manière de faciliter leur compression en traitant les données en amont et notamment en réordonnant les reads.

C'est pourquoi notre équipe a développé l'outil OReO (48), qui est un outil dont le principe global est de regrouper les reads similaires ou se chevauchant dans le fichier qui les contient, de manière à ce qu'ils soient les plus proches possibles entre eux. Leurs identifiants dans K2R deviennent logiquement proches également. Rappelons que dans notre cas, l'identifiant associé à un read correspond à sa position dans un fichier FASTA. C'est pourquoi s'ils sont consécutifs, alors leurs identifiants le seront aussi.

OReO réordonne donc des fichiers FASTA de longs reads, de manière à ce que les reads qui se chevauchent soient proches. Aucune référence extérieure n'est nécessaire au processus, qui est le suivant :

— Assemblage produisant un graphe de contigs

— Ordonnancement des contigs, en parcourant le graphe

— Placement des reads dans le fichier en utilisant l'ordre des contigs et le placement des reads dans chaque contig

Un schema du principe général d'OReO est donné en Figure 4.1.

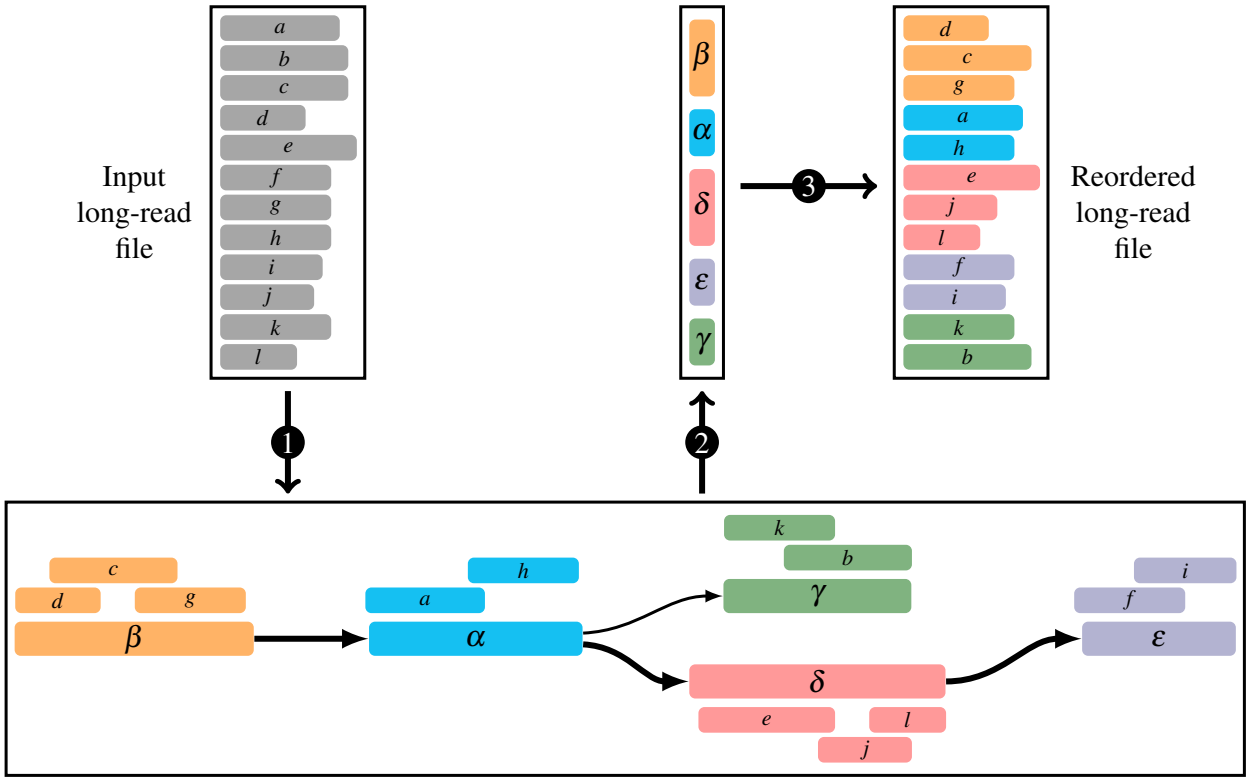


FIGURE 4.1 – *Principe de fonctionnement d'OReO. (1) Un graphe de contigs est construit, (2) puis l'ordre des contigs est calculé en traversant ce graphe, (3) et enfin les reads sont réordonnés en fonction de leurs positions sur les contigs.*

Dans un premier temps, le développement de OReO s'est inscrit uniquement dans le projet de K2R, afin de favoriser la compression de ses couleurs et optimiser l'espace utilisé lors du stockage de sa structure finale. Cependant, nous nous sommes rendu compte lors des tests réalisés du potentiel important qu'un pré-traitement tel que celui-ci pouvait avoir sur les outils de compression classiques largement utilisés, tels que ceux présentés ci-dessus.

Un benchmark a été réalisé sur divers jeux de données (*E. coli*, *A. Thaliana* et Humain, pour chacun HiFi et ONT) afin de prouver l'efficacité de ce concept pour l'étape de compression (Figure 4.2). Chaque test compare deux cas : l'utilisation du compresseur avec et sans réordonnancement avec OReO. Nous avons inclus les compresseurs génériques gz, bzip2, xz et zst. Deux compresseurs longs reads formant un état de l'art sont aussi inclus : NanoSpring et CoLoRd. À noter que ces deux outils utilisent une méthode de réordonnancement interne qui leur est propre, OReO n'est donc pas utilisé en amont de leur compression.

On observe de manière logique un gain en compression plus important sur les données HiFi, moins bruitées. Pour les plus forts taux de compression, un ordre de grandeur minimum est gagné au niveau de la taille du fichier final, pour tous les outils. Seul xz sur le jeu de données *E. coli* obtenait de très bon résultats, le gain est donc moins prononcé. Concernant les données ONT, les gains sont beaucoup plus minimes. xz et zst sont tout de même améliorés pour les plus gros taux de compression, atteignant des niveaux compression comparables à NanoSpring et CoLoRd.

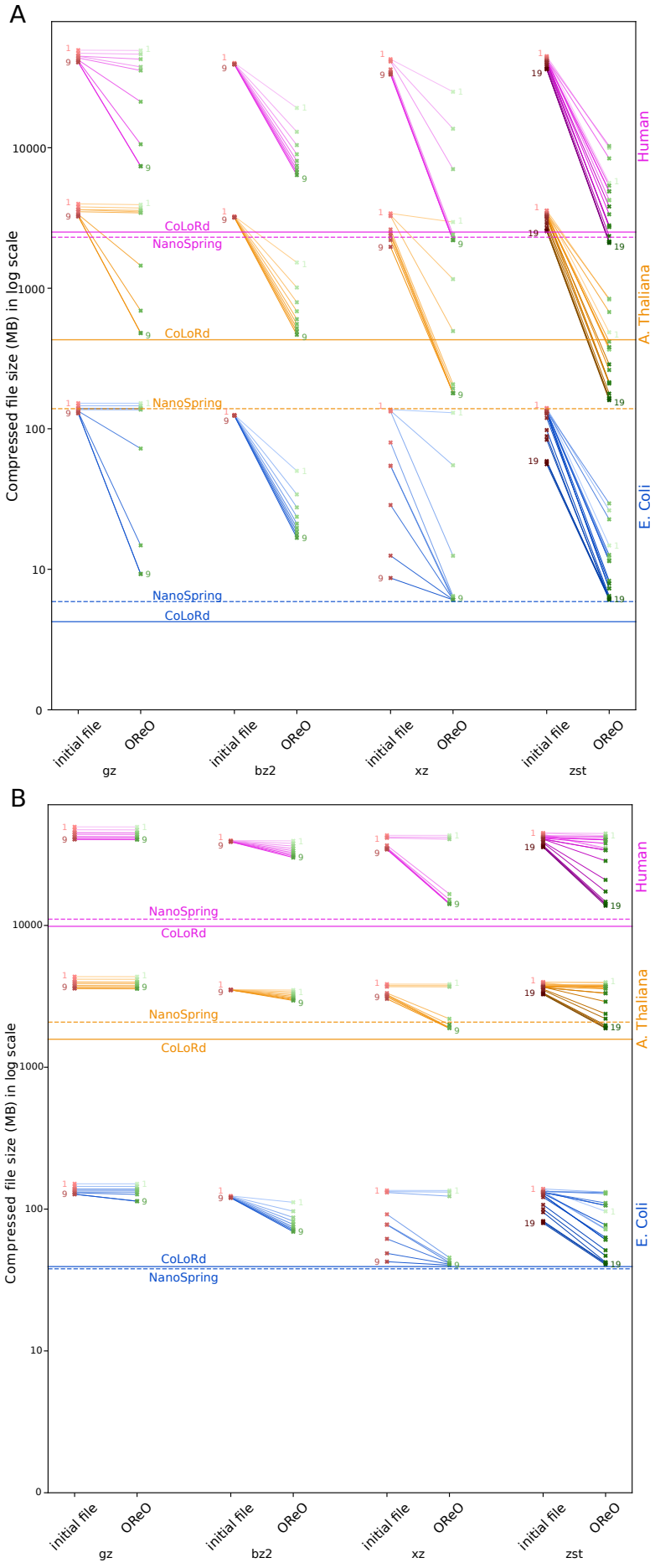


FIGURE 4.2 – *Résultats d'utilisation d'OReO sur plusieurs outils de compression. Les données utilisées sont de type HiFi (A) ou ONT (B). Chacun est présenté en utilisant OReO ou non au préalable, afin de constater la différence. chaque courbe pour un même outil sur le même jeu de données correspond à un niveau de compression différent. Les résultats de CoLoRd et NanoSpring sont également indiqués à titre indicatif. On observe de manière générale une meilleure aide à la compression pour les données HiFi, moins bruitées.*

Toujours pour la compression, une évaluation montrant le coût en temps contre le ratio de compression est disponible en Figure 4.3. Cette fois-ci ce sont des reads HiFi et ONT issus du génome humain qui sont utilisés. Le réordonnancement ne ralentit pas l'opération, tout en offrant des gains en mémoire significatifs. La combinaison d'OReO avec les compresseurs génériques ne présente aucun inconvénient dans ce cas.

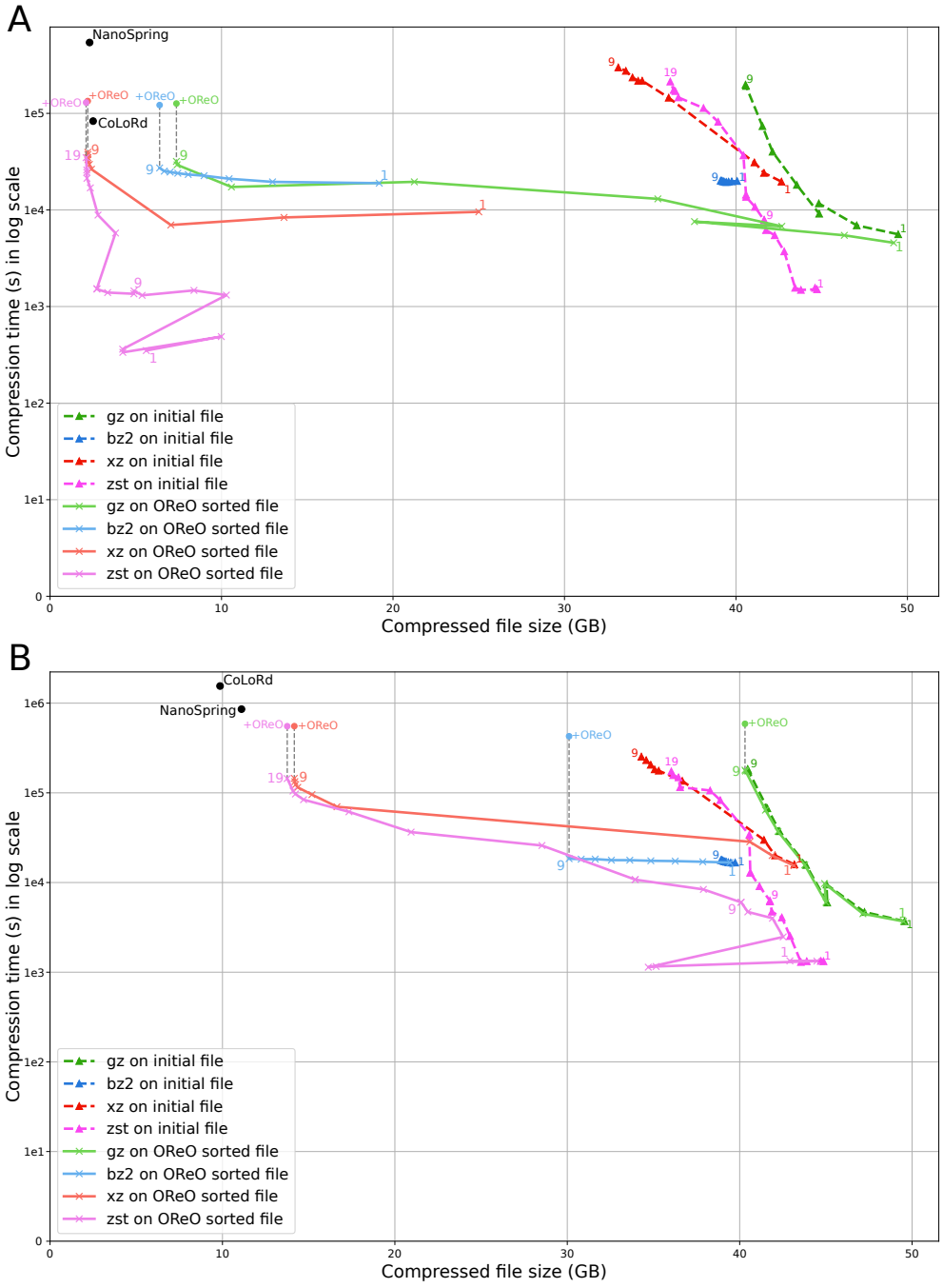


FIGURE 4.3 – *Résultats d'utilisation d'OReO en temps et en mémoire sur des données issues du génome humain, de type HiFi (A) ou ONT (B). OReO n'augmente jamais le temps total d'exécution, tout en offrant des gains en mémoire significatifs.*

En complément de l'optimisation en compression pour chaque outil, le temps de décompression a également été testé. Les mêmes jeux de données et outils ont été utilisés et les résultats sont visibles en Figure 4.4. Cette étape est aussi importante à tester que la compression, dans le sens où un utilisateur peut vouloir décompresser plusieurs fois le même fichier. Le temps nécessaire pour cette étape est donc indispensable à évaluer.

Une fois de plus, on peut observer l'utilité du réordonnancement qui ici, notamment sur les

données HiFi , permet de gagner jusqu'à un ordre de grandeur en temps. Des exceptions sont faites cependant pour bzip2 et zstd.

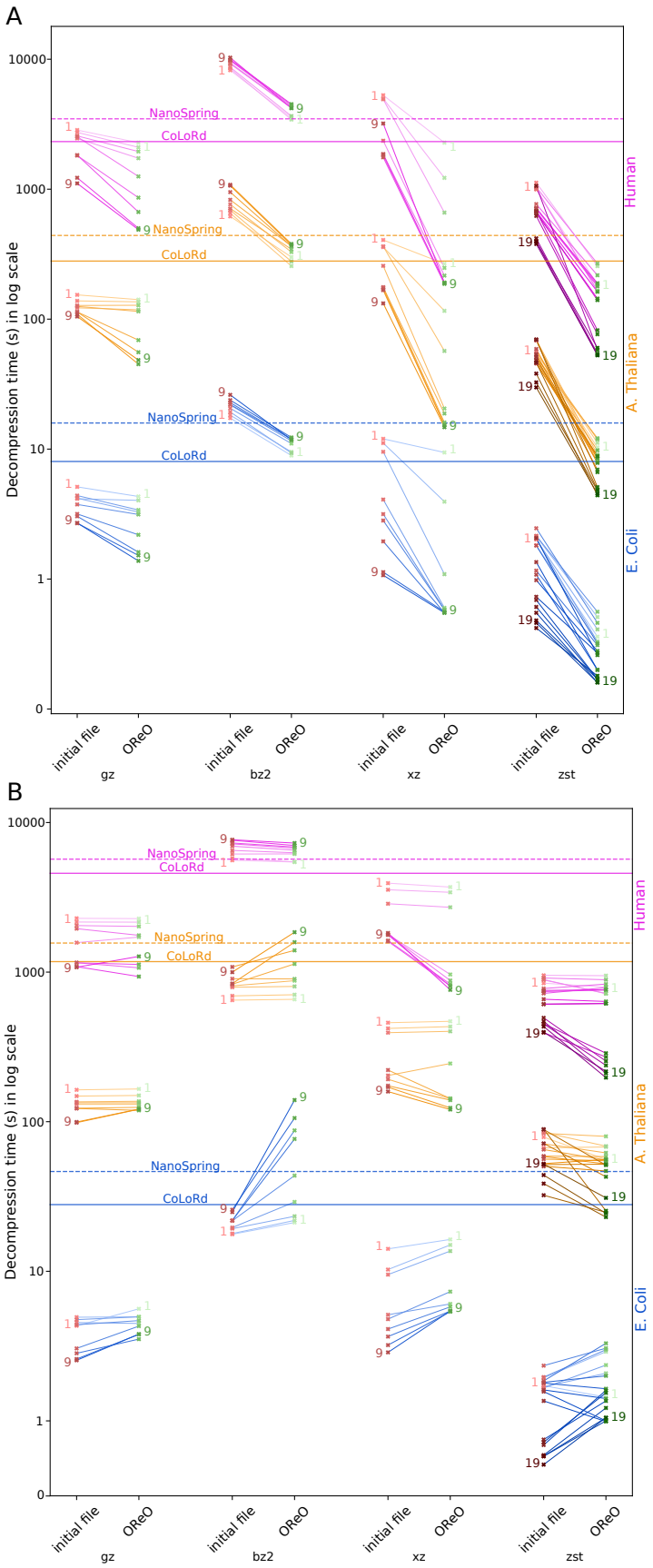


FIGURE 4.4 – *Résultats d'utilisation d'OReO sur plusieurs outils de compression. Les données utilisées sont de type HiFi (A) ou ONT (B). Chacun est présenté en utilisant OReO ou non au préalable, afin de constater la différence. chaque courbe pour un même outil sur le même jeu de données correspond à un niveau de compression différent. Les résultats de CoLoRd et NanoSpring sont également indiqués à titre indicatif. On observe de manière générale une meilleure aide à la compression pour les données HiFi, moins bruitées.*

L'influence de l'utilisation d'OReO a évidemment été testée sur K2R, le but initial étant d'optimiser celui-ci. Différents jeux de données issus de plusieurs génomes ont été utilisés : *E. coli*, *A. Thaliana* et Humain (Table 4.1). Chacun a été testé pour des données HiFi et ONT. On observe que la taille des index créés peut être jusqu'à 5 fois plus petite, notamment sur les données HiFi. Dans tous les cas, la taille d'index est plus petite. Le temps de création de l'index reste sensiblement le même avec ou sans utilisation d'OReO, il n'y a donc pas d'inconvénient à ces gains en mémoire.

Dataset		Time (s)	Index (MB)
E. coli HiFi	Initial/OReO	114/**100**	60/**40**
E. coli ONT	Initial/OReO	112/**103**	60/**13**
A. Thaliana HiFi	Initial/OReO	3197/**2805**	1781/**367**
A. Thaliana ONT	Initial/OReO	3924/**3777**	2685/**2185**
Human HiFi	Initial/OReO	**21951**/22625	6959/**1731**
Human ONT	Initial/OReO	**36718**/38302	6997/**5095**

TABLE 4.1 – *K2R time usage and index size on various input file.*

Ces différents résultats prouvent que le réordonnancement des reads permet d'améliorer significativement les résultats de compression des méthodes génériques, mais également d'autres outils tels que K2R, utilisant la compression via le delta encoding.

4.2 Un index complémentaire : ONIKA

La mise en place de méthodes d'indexation fondées sur les k-mers a permis de dépasser les limites des approches dépendantes d'une référence, notamment celles basées sur l'alignement. L'étape d'assemblage, coûteuse et parfois destructrice, constituait un frein majeur à de nombreuses analyses. Les index orientés k-mers ont ensuite ouvert la voie à de nombreuses applications, calcul de similarité entre échantillons, détection de SNPs, recherche de gènes, localisation de séquences,tout en supprimant la contrainte d'assemblage. Des structures comme K2R ont démontré leur capacité à passer à l'échelle pour de larges volumes de données. Cependant, comparer directement de grands ensembles de documents reste prohibitif en temps et en mémoire. Pour pallier cela, une approche plus compacte a émergé : la réduction des ensembles de k-mers en **sketchs**.

Ces sketchs sont des représentations condensées et de taille fixe qui ne conservent qu'un sous-ensemble représentatif des k-mers, appelés *fingerprints*, généralement obtenus par hachage. La comparaison entre deux sketchs permet d'estimer la similarité entre les séquences correspondantes sans comparer explicitement tous leurs k-mers. Le principe repose sur l'indice de Jaccard, qui mesure la proportion d'éléments partagés entre deux ensembles et donc un proxy de leur similarité :

$$J(A, B) = \frac{|A \cap B|}{|A \cup B|}$$

où A et B désignent deux ensembles de k-mers.

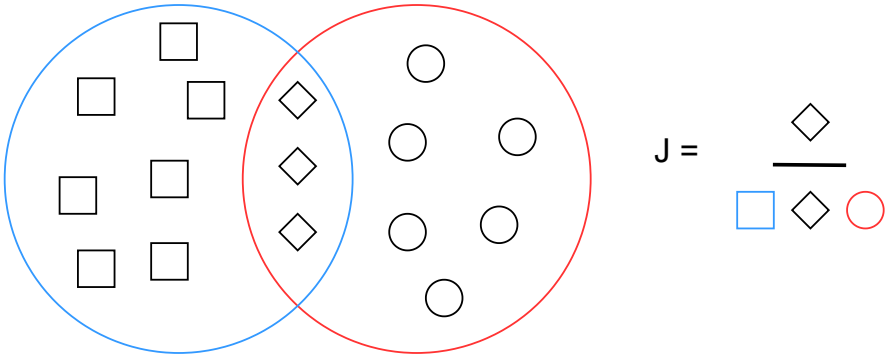


FIGURE 4.5 – *Illustration du calcul de l'indice de Jaccard entre deux ensembles de k-mers.*

Le sketching repose historiquement sur la méthode du **MinHash** (25). Chaque k-mer est haché plusieurs fois selon un ensemble de fonctions de hachage, et seule la plus petite valeur obtenue pour chaque fonction est conservée. Le sketch résultant contient donc les minima pour n fonctions de hachage. Cette méthode fournit une estimation fiable de l'indice de jaccard lorsque les jeux de données sont de taille comparable et relativement proches. Chaque fingerprint résultant d'un hachage aléatoire, la précision de l'estimation croît avec le nombre de fingerprints inclus dans le sketch. Ainsi, plus le sketch est grand, plus l'approximation de $J(A, B)$ est stable, au prix d'un coût mémoire plus élevé.

Des optimisations ont permis de réduire le nombre de fonctions de hachage à deux : l'une pour déterminer la position dans le sketch, l'autre pour calculer la valeur de la fingerprint. Cette approche, adoptée par des outils comme **Dashing** (12), conserve la même structure tout en diminuant fortement le coût de construction et la consommation mémoire.

Le sketch final se compare ensuite élément par élément. L'approximation de la similarité est donc linéaire en la taille des sketchs, ce qui rend la méthode adaptée aux comparaisons massives. Les outils modernes comme **Bindash** (136) et **Dashing** (12) introduisent des optimisations supplémentaires sur la compression des fingerprints : Bindash tronque les bits non significatifs, tandis que Dashing applique une variante de **HyperLogLog**, qui encode la longueur des préfixes de zéros plutôt que les valeurs elles-mêmes.

Le choix de la taille des fingerprints et du sketch (W et S) résulte d'un compromis entre précision et mémoire. Une taille de fingerprint trop faible augmente les collisions, tandis qu'une taille trop grande gaspille de l'espace sans gain significatif. Typiquement, $W = 16$ bits offre un bon équilibre, les collisions restant rares ($\approx 1/65{,}000$). La Figure 4.7 illustre la relation entre taille du sketch et erreur d'approximation sur l'indice de Jaccard.

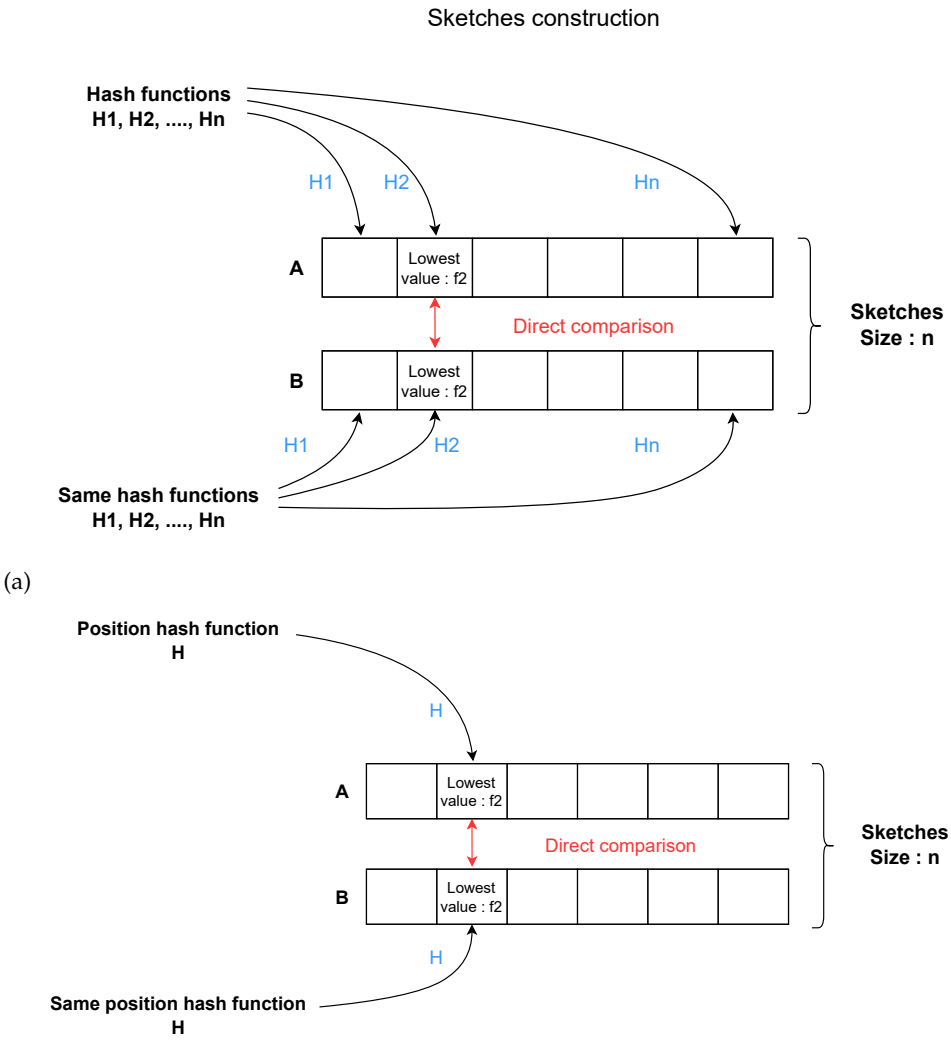


FIGURE 4.6 – *Méthodes de construction de sketchs. (a) MinHash naïf utilisant plusieurs fonctions de hachage ; (b) variante optimisée utilisant deux fonctions seulement, l'une pour la position et l'autre pour la valeur.*

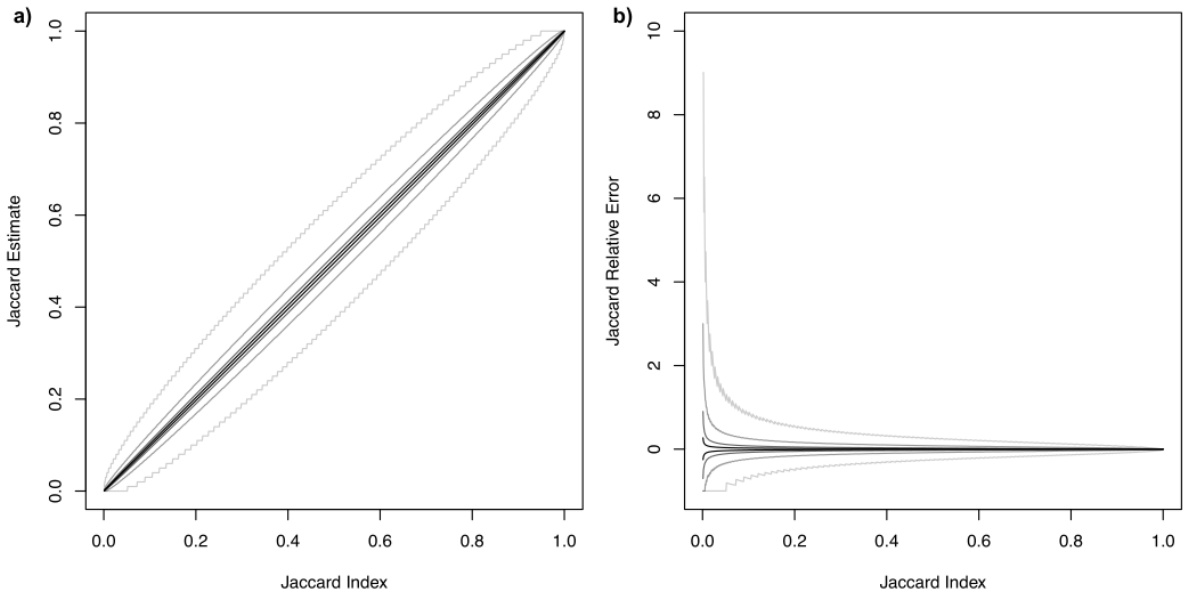


FIGURE 4.7 – *Impact de la taille des sketchs sur l'estimation de l'indice de Jaccard (a) et l'erreur relative (b), d'après (99). Les tailles de sketch utilisées sont 100 (gris clair), 1.000, 10.000 et 100.000 (noir). Ces graphiques montrent principalement que dans le cas d'un indice attendu faible, le taux d'erreur peut rapidement augmenter si la taille du sketch est trop petite.*

En synthèse, le sketching fournit une représentation condensée et probabiliste de grands ensembles de k-mers, permettant la comparaison rapide d'échantillons génomiques à grande échelle. Cette approche constitue la base méthodologique sur laquelle s'appuie notre contribution, **ONIKA**, qui étend ces principes à un modèle d'index inversé optimisé pour la comparaison de très grands jeux de données tout en maîtrisant l'empreinte mémoire et le coût des requêtes.

4.2.1 Index inversés : NIQKI & ONIKA

La manière la plus simple de construire un index de sketchs est de concaténer les sketchs de tous les documents (Figure 4.8). Pour interroger un ou plusieurs documents, on calcule leur sketch et on le compare séquentiellement à chaque sketch stocké dans l'index. On obtient alors une liste d'identifiants de documents associés à leurs scores de similarité, ou une matrice complète de similarité entre l'ensemble des requêtes et des documents.

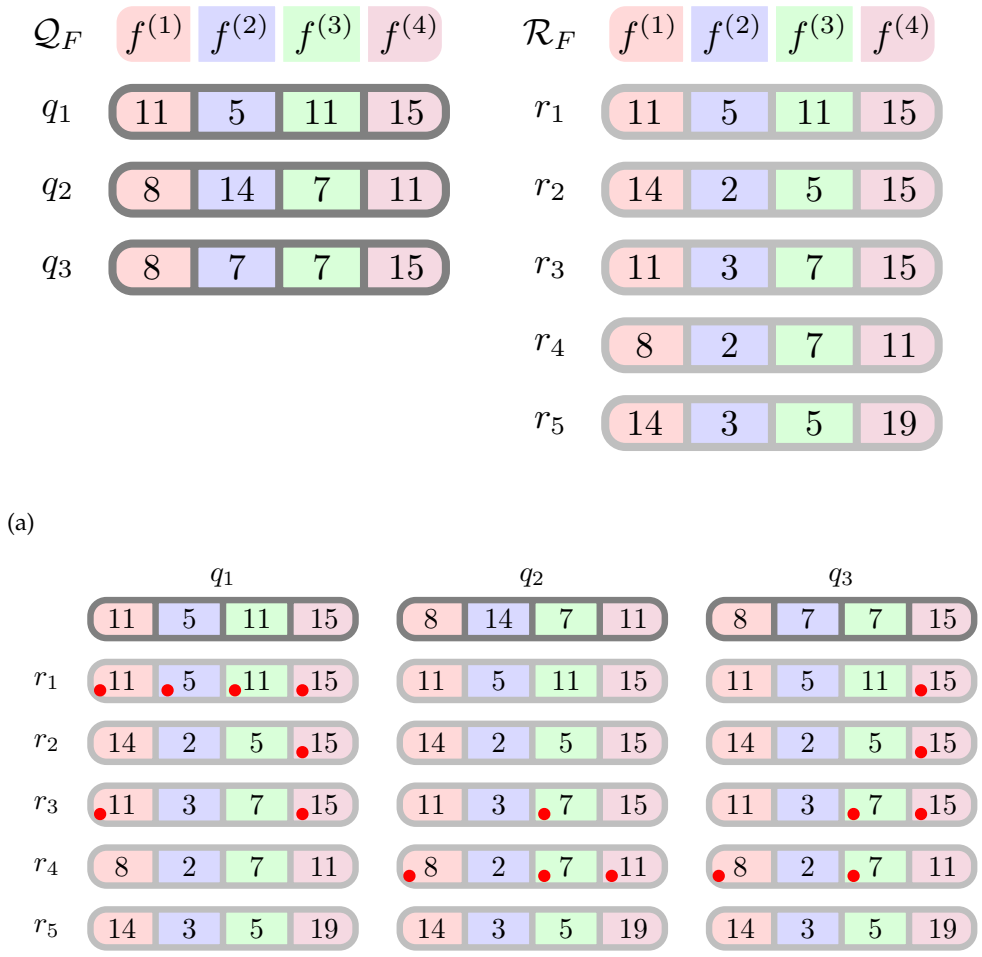


FIGURE 4.8 – *Index et requête pour un index direct. La sous-figure (a) présente un jeu de reads R_F composé de cinq séquences (r_1 à r_5) et trois séquences à requêter Q_F (q_1 à q_3). Chaque séquence est représentée par un sketch de quatre fingerprints ($f(1)$ à $f(4)$). La sous-figure (b) illustre la comparaison de chaque sketch de requête avec ceux de la base, les correspondances étant marquées par des points rouges.*

Cette approche, dite **directe**, présente une limite majeure : lors d'une requête, il faut parcourir l'intégralité de l'index, même si la majorité des fingerprints n'ont aucun intérêt pour la requête. Comme les requêtes sont souvent beaucoup plus petites que l'index complet, ce parcours exhaustif induit un coût élevé.

Une alternative consiste à inverser la relation d'indexation : au lieu d'associer un document à

ses fingerprints, on associe chaque fingerprint à la liste des documents qui la contiennent. Cette approche correspond à un index inversé, illustré en Figure 4.9. Elle permet d'accéder directement aux fingerprints présentes dans la requête, sans parcourir le reste de l'index.

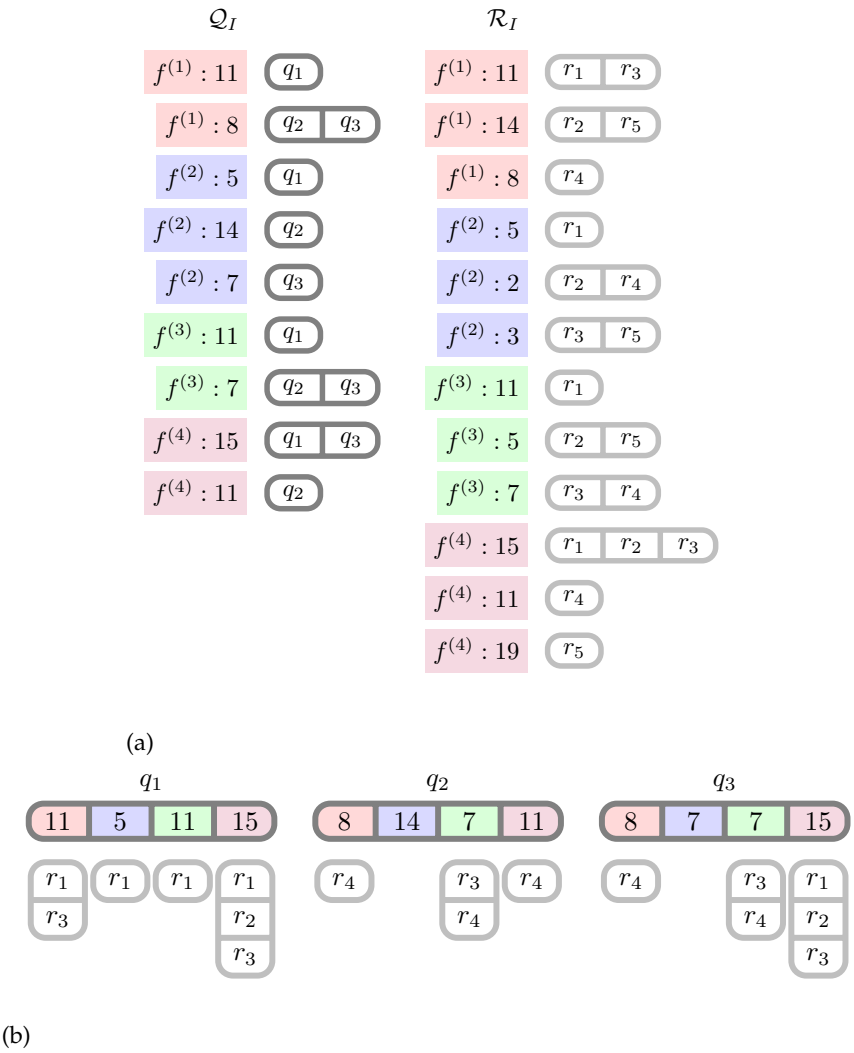


FIGURE 4.9 – *Indexation inversée pour la comparaison de sketchs. Les mêmes jeux de données que dans la Figure 4.8 sont utilisés. (a) : chaque fingerprint est associée à la liste des reads dans lesquels elle apparaît. (b) : la comparaison est directe, seules les fingerprints communes aux requêtes et à l'index sont explorées.*

Cette logique d'index inversé est utilisée dans **NIQKI** (1), qui associe pour chaque fingerprint la liste des jeux de données où elle apparaît. Néanmoins, NIQKI doit encore accéder plusieurs fois à une même fingerprint si plusieurs requêtes la partagent, ce qui limite son efficacité.

Pour dépasser cette contrainte, nous avons conçu **ONIKA**, une extension pleinement inversée du paradigme précédent. Ici, chaque fingerprint est associée directement à l'ensemble des documents la contenant, aussi bien pour les jeux de données indexés que pour les requêtes. Les deux structures étant organisées par partitions, puis par fingerprints, leur comparaison se fait en un seul parcours séquentiel (Figure 4.10).

Pour chaque fingerprint commune, si les deux listes d'identifiants (requêtes et index) sont non vides, les scores des paires correspondantes sont incrémentés. Cette approche évite les accès aléatoires et les répétitions, tout en garantissant un temps de traitement proportionnel uniquement au nombre de correspondances effectives.

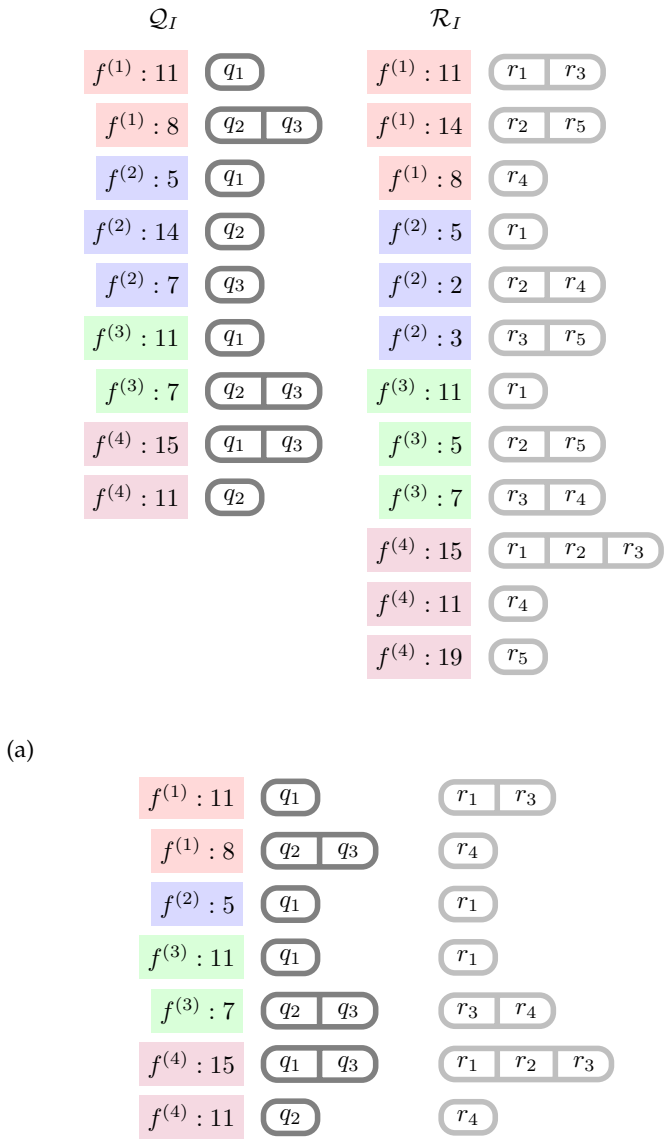


FIGURE 4.10 – *Requête entre deux index inversés. Les éléments des sketchs sont triés par partition (couleur) puis par fingerprint. Si les deux fingerprints d'une partition possèdent des listes de reads non vides, les paires correspondantes sont comptabilisées.*

En pratique, ONIKA atteint une complexité optimale : chaque correspondance entre deux sketchs est traitée une seule fois, sans coût d'accès aléatoire. Cette conception assure un passage à l'échelle linéaire en la taille effective des correspondances, ce qui permet la comparaison efficace de millions de sketchs de grands jeux de données génomiques.

Dans le but d'optimiser une nouvelle fois le coût final en mémoire, nous avons ajouté un réordonnancement optionnel des reads en amont de la création de la structure. À l'image d'OReO, ce réordonnancement permet de regrouper les reads similaires, qui contiendront des fingerprints identiques. Une même fingerprint sera donc plus facilement associée à des identifiants de reads successifs ou proches.

4.2.2 Résultats

Afin de valider les performances d'ONIKA, nous avons décidé de le comparer à deux outils : BinDash2 (135) et Dashing2 (13), outils les plus proches auxquels nous pouvons nous comparer. Dashing2 suit le même principe que Dashing mais utilise SetSketch (41). BinDash2 est une version améliorée de BinDash, qui a de meilleures performances notamment grâce à l'utilisation du

SIMD. Nous avons dans un premier temps indexé des génomes aléatoires de taille 1Mb (Figures 4.11 et 4.12).

En termes de mémoire, ONIKA est presque un ordre de grandeur moins gourmand que Bindash2. Cependant, une forte différence subsiste avec Dashing2, également un ordre de grandeur moins lourd.

En termes de temps, par contre, ONIKA est bien plus rapide que les deux outils, de deux voire trois ordres de grandeur.

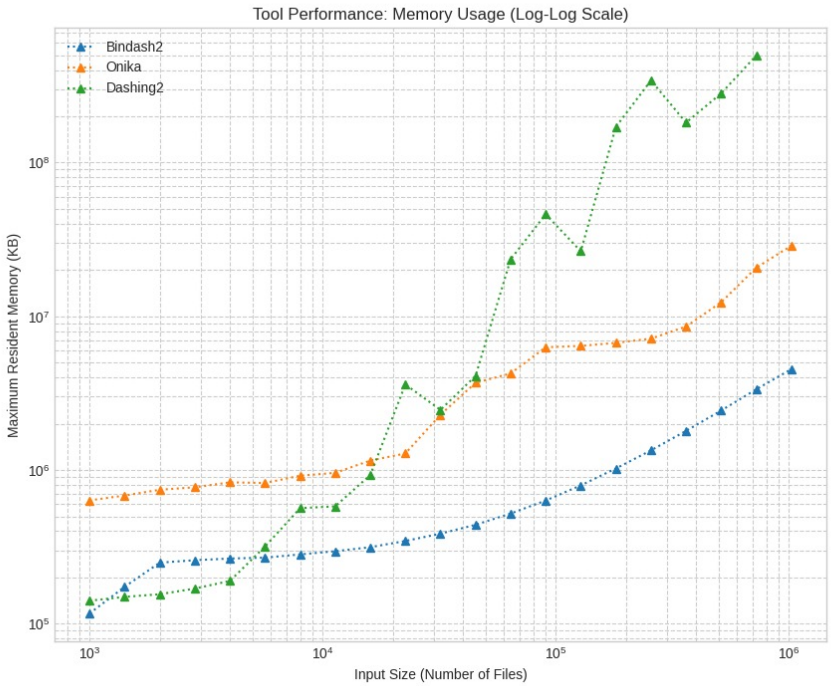
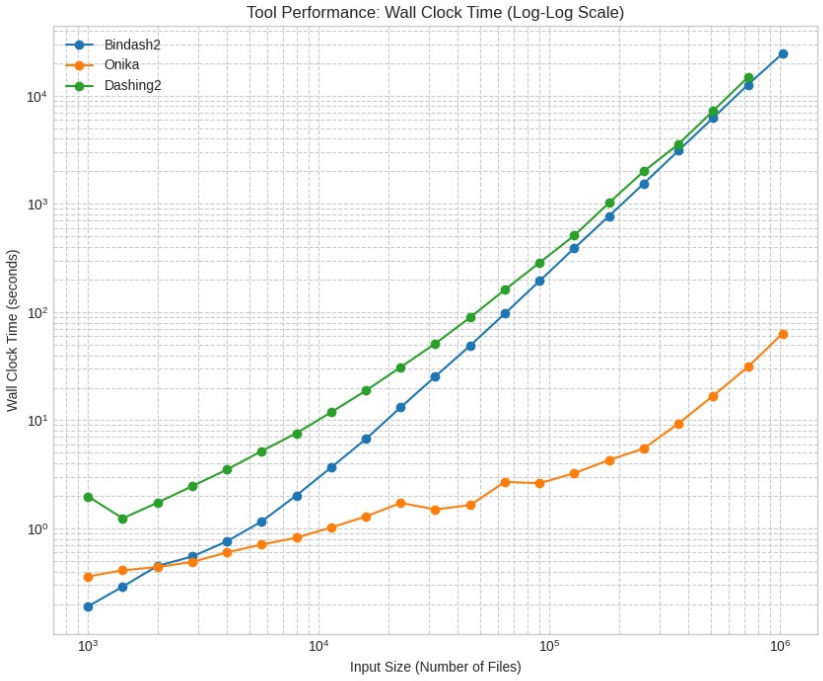
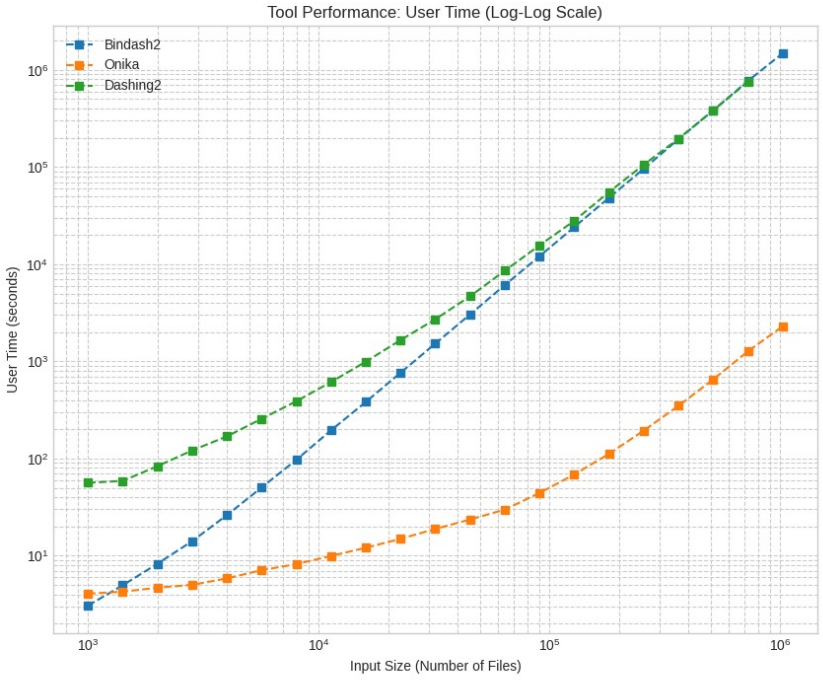


FIGURE 4.11 – *Performances d'ONIKA en mémoire. Le jeu de données utilisé consiste en des génomes simulés de manière aléatoire de taille 1Mb.*



(a)



(b)

FIGURE 4.12 – *Performances en temps wall-clock (a) et système (b) d'ONIKA. Le jeu de données utilisé consiste en des génomes simulés de manière aléatoire de taille 1Mb.*

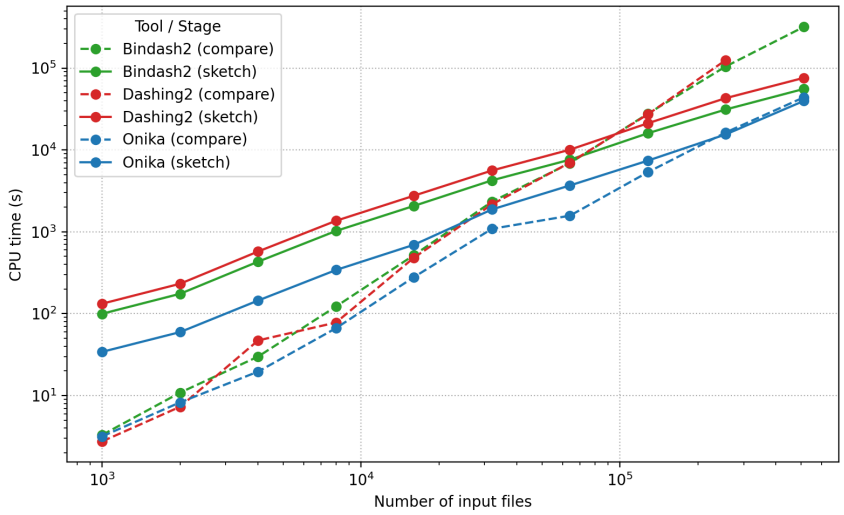
Nous avons par la suite testé les performances d'ONIKA dans le cadre de la comparaison de génomes bactériens à grande échelle, qui est un cas d'usage classique des méthodes basées sur le sketching. ONIKA est toujours comparé à Dashing2 et Bindash2.

Dans la Figure 4.13a, nous présentons les temps de création des sketchs et de comparaison sur des collections de génomes bactériens RefSeq de taille croissante. La construction des sketchs

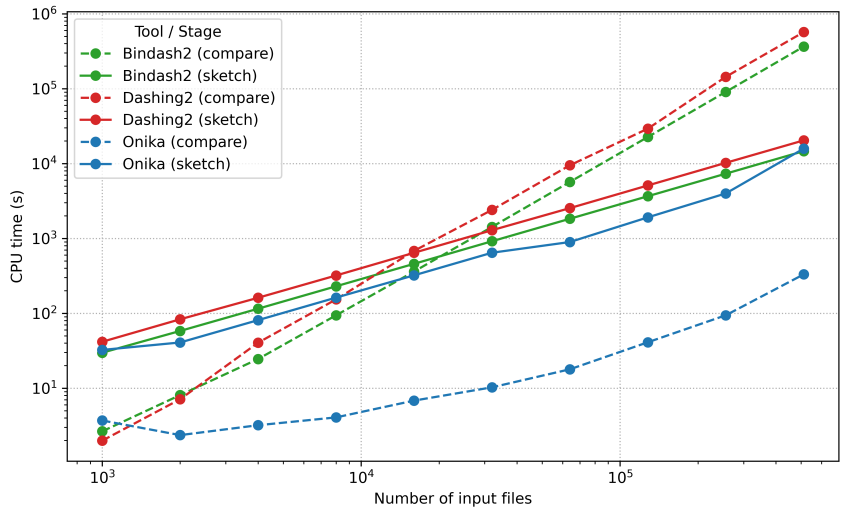
avec ONIKA est plus rapide que celle des deux concurrents. Plus important encore, sa phase de comparaison est presque identique sur les petites collections mais devient un ordre de grandeur plus rapide lorsque la taille de la collection augmente.

Cependant, ce test n'avantage pas ONIKA. Notre approche se base sur le score total de la matrice de similarité et est pénalisée lorsque de nombreux jeux de données sont très similaires, ce qui augmente le nombre de paires à comparer. Les collections bactériennes sont connues pour contenir une forte redondance, ce qui rend ce contexte proche d'un pire cas pour ONIKA.

Pour illustrer un test sur un cas plus avantageux, la Figure 4.13b montre la même expérience sur des collections de séquences aléatoires indépendantes de longueur 1Mb. Ces jeux de données ne partagent aucune similarité significative, ce qui minimise le score total de la matrice et révèle le comportement optimal de notre outil. Dans ce contexte, ONIKA est plus de trois ordres de grandeur plus rapide que l'état de l'art. Bien qu'indexer des séquences aléatoires soit artificiel, cette expérience donne une indication de la performance que l'on peut attendre de futures grandes collections de génomes, non redondantes.



(a)



(b)

FIGURE 4.13 – *Performances en temps CPU d'ONIKA sur plusieurs tailles de collection de génomes bactériens RefSeq (a), ainsi que sur des séquences aléatoires de taille 1Mb (b). Dû aux caractéristiques des collections bactériennes (redondances), ONIKA a des performances moins marquées sur ce cas, mais reste compétitif. Par contre, pour des séquences générées aléatoirement, ONIKA obtient un temps de comparaison d'un ordre de grand grandeur plus faible.*

Une autre application importante du sketching est la détection de reads similaires au sein d'un jeu de données, par exemple dans le cadre d'alignement sans référence ou encore d'assemblage. Pour mesurer nos performances dans ce contexte, nous avons indexé plusieurs couvertures de données HiFi du génome *A. Thaliana*. À noter que nous ne nous sommes pas comparés à Bindash2 ici car l'outil n'est pas développé pour ce cas d'utilisation dans le sens où chaque read devrait être représenté par un sketch, ce qui n'est pas envisageable. Nous présentons les résultats obtenus en Figure 4.14.

L'utilisation classique (sans pré-traitement des données, ici dénoté baseline en légende) révèle qu'ONIKA peut être jusqu'à un ordre de grandeur plus rapide que Dashing2 lorsque la couverture augmente. La tendance est la même en termes de taille de sketchs, avec une taille presque trois fois inférieure pour une couverture de 100X. Nous avons également souhaité observer l'impact que pourrait avoir un réordonnancement de reads en utilisant OReO en amont. Il permet, à l'image de K2R, de réduire la taille de l'index inversés en rapprochant les identifiants de reads les uns des autres. On observe un impact très faible de ce réordonnancement sur le temps CPU utilisé, mais celui-ci augmente, comme attendu, au niveau de la taille finale. L'impact est impressionnant sur Dashing2, mais ONIKA reste plus léger au final. Le réordonnancement propre à ONIKA permet également une réduction de la taille des sketchs, à un niveau plus faible.

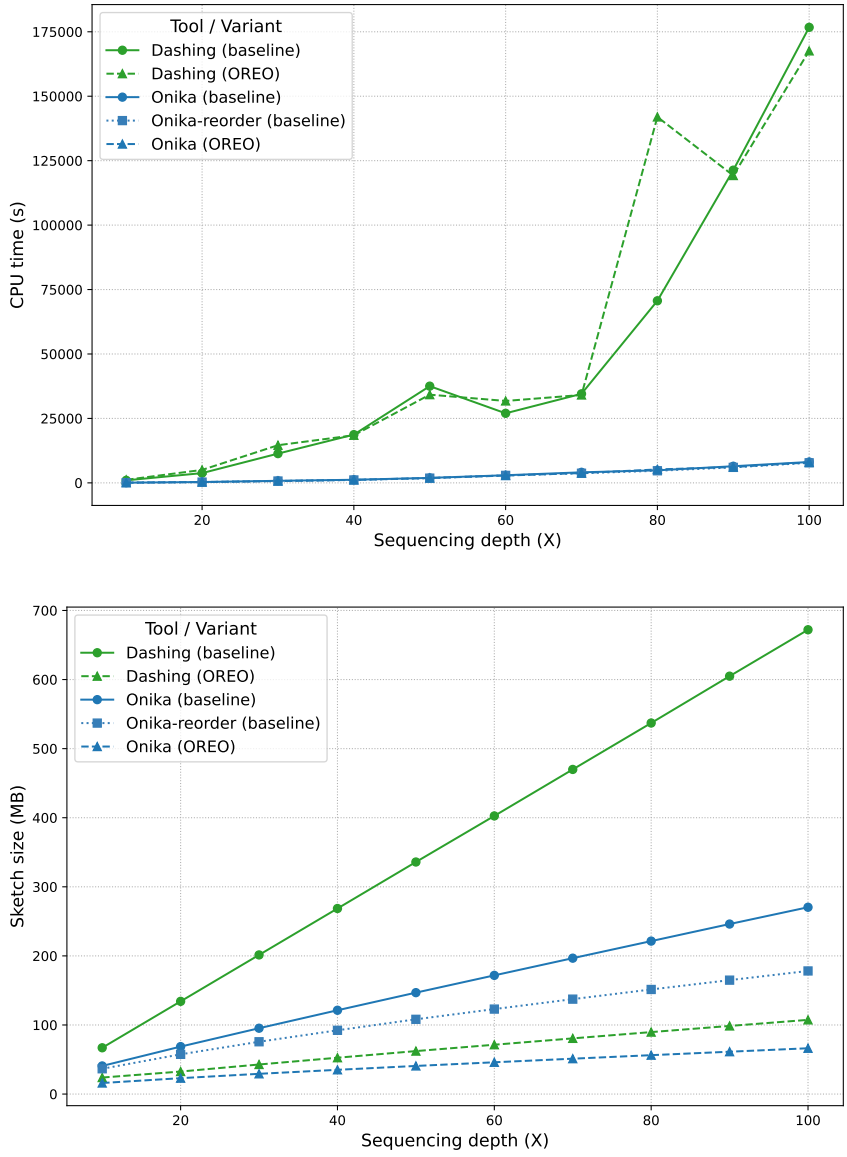


FIGURE 4.14 – *Performances en taille de skecths (a) et en temps CPU (b) d'ONIKA dans le cadre de calcul de similarité entre reads. Les reads à comparer sont issus de plusieurs couvertures de données HiFi du génome A.Thaliana.*

Pour terminer, nous avons précédemment soutenu que la représentation en index inversé utilisée par ONIKA ne devrait pas dépasser la taille d'un index direct stockant explicitement les sketchs. Nous avons souhaité montrer expérimentalement ce point. La Figure 4.15 montre les tailles de sketch pour les expériences RefSeq présentées plus haut (Figure 4.13). Pour assurer une comparaison équitable, tous les sketchs sont compressés avec zstd -1, qui est la valeur par défaut d'ONIKA. Nous observons que les tailles de sketch produites par ONIKA sont comparables à celles de Bindash2, index direct, ce qui confirme en pratique les attentes théoriques, et que l'étape optionnelle de réordonnancement dans ONIKA peut réduire la taille des sketchs de plus de 35%. En pratique, nous nous attendons à ce que les gains liés au réordonnancement dépendent fortement du niveau de redondance présent.

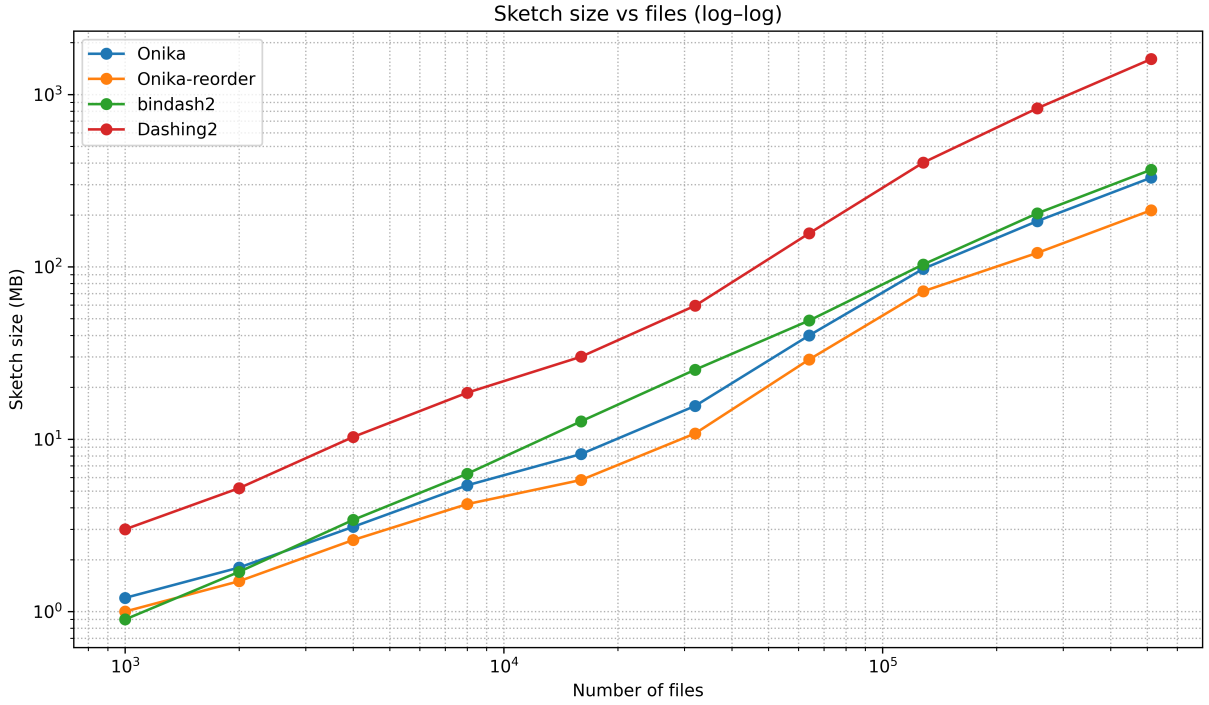


FIGURE 4.15 – *Comparaison des tailles de sketchs entre index directs (Dashing2 et Bindash2) et ONIKA. Les jeux de données utilisés sont des collections de génomes bactériens de différentes tailles. On observe qu'ONIKA a des tailles de sketchs comparables à celles d'index directs.*

4.2.3 Discussion et perspectives

Nous avons développé ONIKA, permettant de comparer efficacement des jeux de données pour mesurer leur ressemblance. Bien que nous soyons déjà capables d'être plus efficaces que les outils de référence BinDash2 et Dashing2, des pistes sont encore à étudier afin de continuer à améliorer ONIKA. En particulier, le pire cas en termes de performances auquel nous pouvons nous confronter est celui de jeux de données identiques, qui nécessiterait en l'état actuel de rechercher les reads dans lesquels apparaissent toutes les fingerprints, en ayant une correspondance à chaque fois. Il serait peut-être possible de proposer plusieurs algorithmes différents en fonction de la similarité attendue. Dans le cas d'une similarité importante, une option serait de calculer sur le même principe les différences, afin d'inverser le problème et garder des requêtes efficaces.

Une perspective future consiste également à comparer ONIKA avec K2R, afin d'évaluer si les résultats obtenus en termes de reads associés aux minimizers sont comparables. Avec K2R, les

requêtes sont toujours exactes, il serait donc intéressant de mesurer à quel point ONIKA peut s'en approcher tout en ne stockant que des sketches.

4.3 K2Rmini

La plupart des approches d'indexation transforment le document pour accélérer des requêtes multiples (recherche de motifs, séquences d'intérêt). Cette logique, suivie par K2R, amortit le coût lorsqu'on effectue beaucoup de requêtes. En revanche, pour un nombre limité de requêtes, ce coût devient disproportionné : l'indexation de grands jeux de données exige du temps et de la mémoire, et impose souvent des structures probabilistes avec faux positifs pour passer à l'échelle.

À l'échelle LOGAN (30) (centaines de To de contigs), un index exhaustif "à la base près" est impraticable. Deux stratégies complémentaires s'imposent : (i) un index global de présence, léger et probabiliste, pour présélectionner les documents pertinents, (ii) une vérification *streaming* exacte sans index lourd sur les cibles retenues. On inverse alors le paradigme classique : on indexe la requête, puis on parcourt directement les cibles pour retrouver les séquences correspondantes. C'est le principe de BTS (11). Avantage : pas de faux positifs inhérents à la structure d'index, un espace réduit côté requête, et un coût proportionnel au volume réellement scanné.

BTS repose sur une table de hachage des k-mers de la requête. Sa limite vient du lookup par k-mer sur le flux d'entrée : avec de grandes requêtes ou des lots de requêtes, la table grossit, sort du cache, et le débit chute. Nous avons donc développé K2Rmini (88), qui index les minimizers en plus des k-mers pour filtrer au plus tôt les séquences non pertinentes. Seuls les reads/contigs partageant un nombre suffisant de minimizers avec la requête passent au comptage exact des k-mers. Si la fenêtre de sélection $(k-m)$ fait w, un nécessaire pour N k-mers partagés est d'avoir au moins $\lceil N/w \rceil$ minimizers partagés. Ce préfiltrage réduit d'un facteur $\simeq 2/(w+1)$ le nombre de tests, garde une table compacte et cache-friendly, et garanti l'éxactitude des résultats en n'étant qu'un filtre nécessaire avant vérification exacte. K2Rmini exploite en outre une extraction SIMD des minimizers (63), ce qui rapproche le temps de traitement de la limite I/O.

Opérationnellement, la chaîne devient : sélection rapide des documents par index global de présence, indexation des requêtes en minimizers, parcours séquentiel des cibles, filtrage par minimizers, puis comptage exact des k-mers sur les seules séquences candidates. Par rapport à K2R, la différence est donc le « côté indexé » (corpus pour K2R, requêtes pour K2Rmini), comme illustré en Figure 4.16. Cette dualité permet de choisir la voie la plus économique selon le ratio {taille documents, nombre de requêtes, volumétrie par requête} tout en conservant une vérification exacte finale.

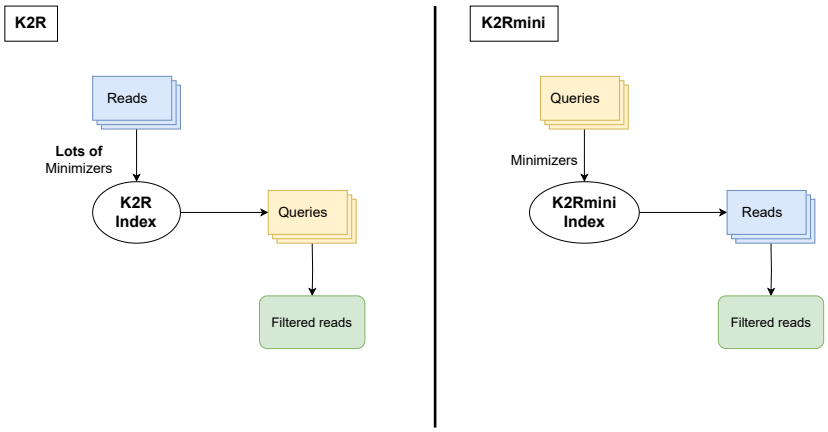


FIGURE 4.16 – *Principes de K2R et K2Rmini. La principale différence de fonctionnement entre les deux outils réside dans ce qui est indexé : le jeu de données pour K2R, les éléments à requêter pour K2Rmini.*

En complément, K2Rmini exploite l'exécution parallèle via SIMD (Single Instruction, Multiple Data, voir Figure 4.17). Cette technique permet d'accélérer les traitements en appliquant une même instruction à plusieurs données simultanément, ce qui est particulièrement efficace pour le traitement massif de séquences.

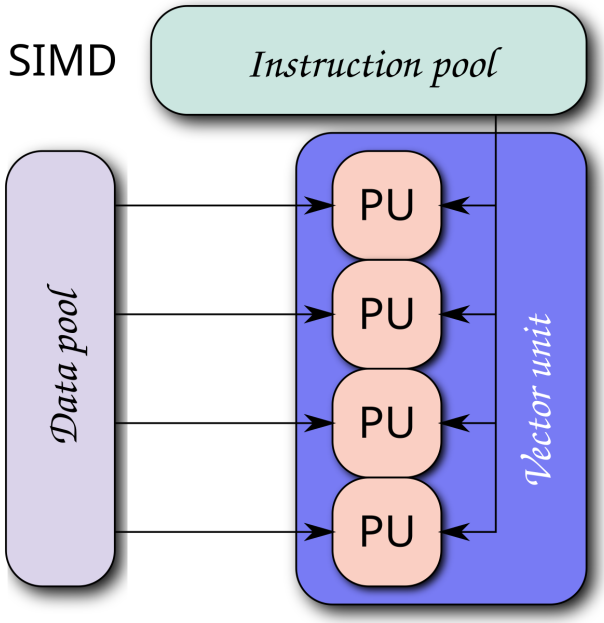


FIGURE 4.17 – *Schéma de fonctionnement du SIMD. Figure reprise de :* `https://en.wikipedia.org/wiki/Single_instruction,_multiple_data`*. De manière générale, le SIMD est un mode où une seule instruction est exécutée en parallèle sur plusieurs données grâce à un ensemble d'unités de traitement (Processing Unit, PU) qui se regroupent en un vecteur (Vector Unit).*

Prenons un exemple simple pour illustrer le principe du SIMD (Single Instruction, Multiple Data) : l'addition de deux vecteurs. Si l'on considère deux vecteurs de taille 3, A=[1,2,3] et B=[4,5,6], leur addition élément par élément donne [1+4,2+5,3+6]. Sans SIMD, le processeur exécuterait trois instructions successives, une pour chaque paire d'éléments. En revanche, avec SIMD, une seule instruction permet de traiter simultanément plusieurs paires de données, réduisant ainsi significativement le temps de calcul. Dans notre cas d'utilisation, cette stratégie est particulièrement avantageuse pour le calcul des minimizers.

Nous avons réalisé un benchmark montrant les performances de K2Rmini, en le comparant

aux outils qui constituent l'état de l'art. Les résultats sont présentés en Figure 4.18. Nous avons en l'occurrence testé les outils suivants :

— BTS (Back-to-Sequences) (11), basé sur une table de hachage
— SeqKit2 (115), optimisé pour les fichiers Fasta
— Les outils dérivés de Grep (grep, ripgrep (46), grepq (34), fqgrep (45))
— Hyperscan (125), recherche des motifs par expressions régulières multiples

Les expérimentations ont été menées sur un jeu de données *E. coli* HiFi (Accession : SRR11434954), présentant une couverture d'environ 100X. Les tests consistent à effectuer des requêtes (positives ou négatives) sur un nombre variable de *k*-mers successifs présents ou non dans ce jeu de données.

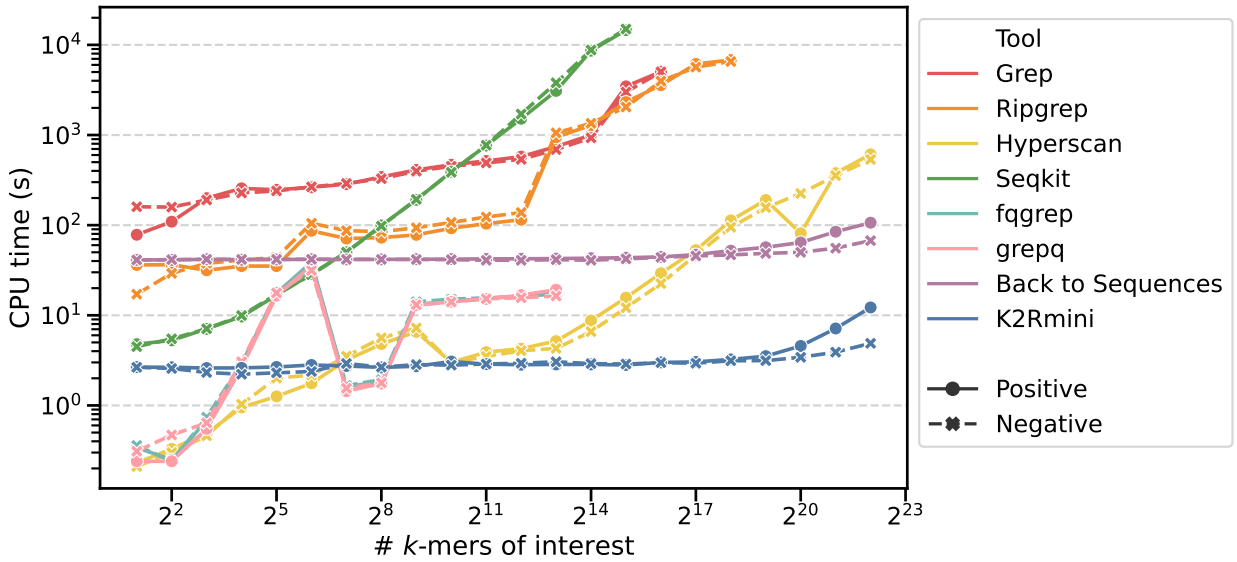


FIGURE 4.18 – *Comparaison des performances de K2Rmini avec l'état de l'art, en termes de temps CPU (durée pendant laquelle le processeur est effectivement occupé à exécuter un programme) pour des requêtes positives et négatives. Nous faisons ici varier le nombre de k-mers à requêter (de 2 à 2^{22}), en les recherchant dans un jeu de données E. coli HiFi de couverture 100X.*

Une première observation concerne la performance des méthodes basées sur le hachage (BTS et K2Rmini). Ces deux approches s'avèrent particulièrement efficaces dès lors que le nombre de *k*-mers à rechercher devient important. Dans le cas d'un petit volume de requêtes (jusqu'à environ 2^{13} séquences), seul Hyperscan parvient à rivaliser, voire à surpasser ces méthodes en termes de rapidité.

En comparaison directe, K2Rmini se distingue par une exécution environ un ordre de grandeur plus rapide que BTS. Cependant, cet écart tend à se réduire à mesure que le nombre de *k*-mers à requêter augmente. En effet, plus le volume de séquences recherchées croît, plus il est probable qu'un plus grand nombre d'entre elles franchissent le filtre basé sur les minimizers. Le comportement de K2Rmini converge alors progressivement vers celui de BTS, dans un mode de fonctionnement sans filtrage préalable.

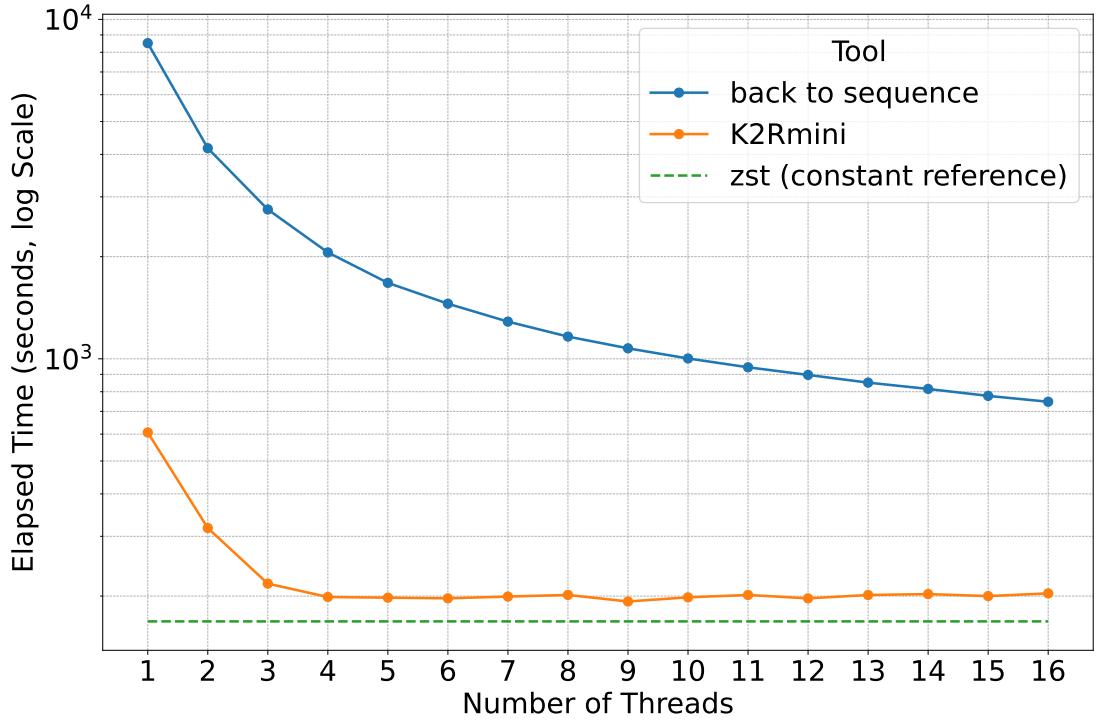
Pour compléter ce benchmark, d'autres tests ont été réalisés exclusivement sur BTS et K2Rmini, en utilisant un jeu de données Nanopore humain de reads longs, un jeu de données HiFi de reads de longueur 10 à 20 kb, ainsi qu'un jeu de données Illumina de reads courts (250 pb). Nous avons également évalué les performances sur des séquences assemblées, à partir d'unitigs et de contigs issus de LOGAN, du génome humain de référence T2T complet, caractérisé par ses très longues séquences contiguës, ainsi que de l'ensemble de la collection RefSeq de génomes bactériens, représentative par son grand nombre de génomes individuels. Les résultats

sont présentés en Table 4.2. Les résultats montrent que K2Rmini est significativement plus rapide que BTS sur l'ensemble des jeux de données testés, et plus économe en mémoire dans presque tous les cas.

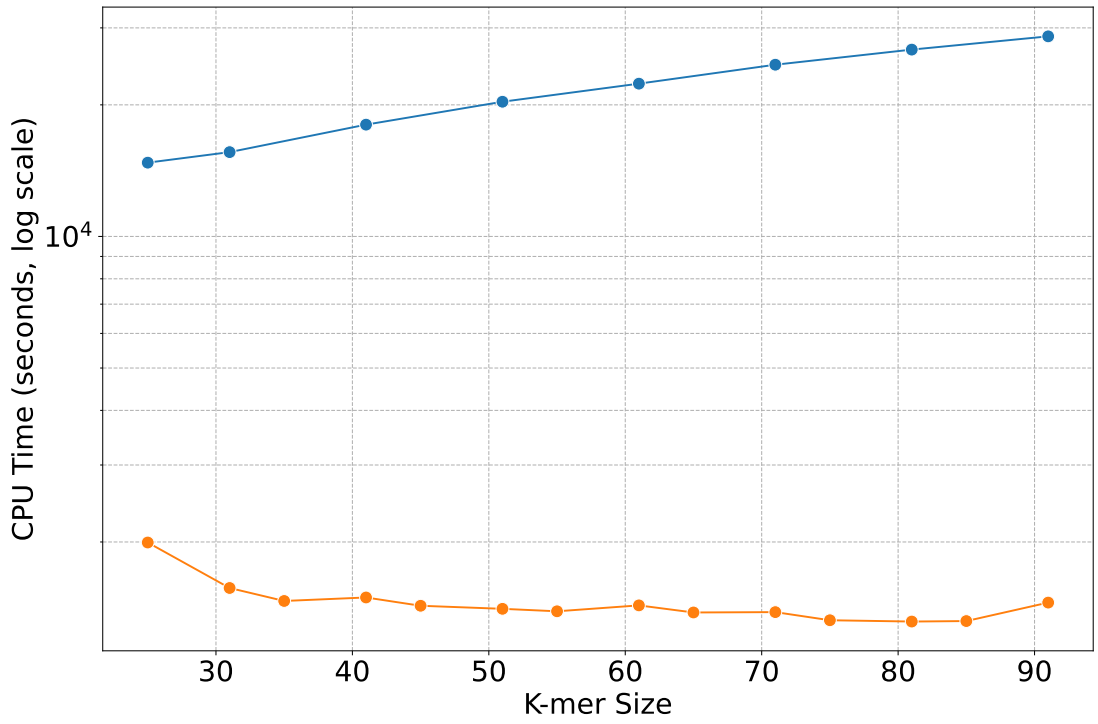
	BTS			K2Rmini		
Dataset	**CPU**	**Elapsed**	**Memory**	**CPU**	**Elapsed**	**Memory**
ONT (SRR23365080)	26,987	15m01	1,303	**1,465**	**6m20**	**103**
HiFi (SRX7897685-8)	13,363	7m26	235	**732**	**2m55**	**31**
Illumina (ERR3239454)	5,481	5m22	177	**465**	**1m36**	**34**
T2T (GCF_009914755.1)	152	2m33	**3,185**	**8**	**3**	4,112
Contigs (SRR7853572)	121	20	148	**16**	**5**	**32**
Unitigs (SRR7853572)	342	1m51	148	**94**	**38**	**36**
Refseq Bacteria	17,873	10m11	9,224	**862**	**5m5**	**1,777**

TABLE 4.2 – *Comparaison des performances de BTS et K2Rmini. Les temps sont exprimés en minutes et secondes et la mémoire en Mo. Les meilleurs résultats sont en gras.*

Nous avons finalement évalué l'influence des paramètres utilisés et notamment la longueur des *k*-mers, également pour BTS et K2Rmini (voir Figure 4.19). Une longueur de *k*-mer plus importante a tendance à réduire légèrement le temps CPU utilisé pour K2Rmini, mais augmente le temps utilisé par BTS. Ce comportement est attendu, car une taille plus importante de *k*-mer réduira le nombre de minimizers utilisés grâce à une fenêtre plus large, et donc accélère le filtrage.



(a)



(b)

FIGURE 4.19 – *Influence des paramètres (nombre de threads (4.19a) et taille des k-mers (4.19b)) sur les performances de K2Rmini et BTS en temps. Un jeu de données Humain HiFi ayant une profondeur de 50X est utilisé ici.*

Ce travail a démontré qu'une recherche reposant sur un filtrage par minimizers accéléré par SIMD est à la fois rapide et économe en ressources pour de nombreuses applications.

4.4 Résultats principaux

Les outils présentés dans ce chapitre prolongent le développement de K2R en apportant des solutions aux limites identifiées concernant la taille des index et la rapidité de requête pour OReO

et K2Rmini, ainsi qu'à un besoin applicatif supplémentaire avec ONIKA, tout en maintenant une structure en lien avec K2R.

Nous avons montré que K2Rmini, grâce à sa stratégie inversée consistant à indexer les requêtes plutôt que les jeux de données, complète efficacement K2R dans le contexte de grands volumes de reads. Il permet d'obtenir des requêtes positives comme négatives jusqu'à un ordre de grandeur plus rapides que l'état de l'art. Comparé à BTS, qui constitue une version non optimisée de cette approche, K2Rmini offre dans de nombreux cas un gain d'au moins un ordre de grandeur en temps comme en mémoire, tout en étant presque systématiquement plus performant.

OReO, pour sa part, réduit fortement la taille des index produits par K2R en réordonnant les reads en amont de l'indexation. Sur des jeux de données HiFi issus des génomes *Humain* et *A. Thaliana*, il permet une réduction de la taille finale stockée d'un facteur quatre à cinq. Nous avons également constaté qu'OReO améliore significativement l'efficacité de compresseurs classiques, avec des gains d'un ordre de grandeur en temps comme en taille de fichier, en particulier sur des données HiFi.

Enfin, ONIKA propose une méthode d'indexation fondée sur le sketching, offrant d'excellentes performances pour la comparaison de collections de séquences. Bien qu'il soit moins adapté aux cas où les séquences sont très proches, ONIKA reste plus rapide que l'état de l'art, pouvant atteindre jusqu'à trois ordres de grandeur de gain dans ses meilleurs scénarios. ONIKA bénéficie également d'une forte complémentarité avec OReO, ce dernier permettant de réduire par un facteur cinq la taille des sketchs générés grâce à son réordonnancement des reads.

Chapitre 5

Conclusion et perspectives

5.1 Retour sur le développement de K2R

L'objectif initial de cette thèse était de développer un moyen d'indexation adapté aux longs reads, capable de passer à l'échelle dans le cas de gros génomes, à l'exemple du génome humain. Cette idée est venue de plusieurs observations :

— L'analyse de novo des séquences génomiques permet d'éviter la difficulté que représente l'opération d'assemblage, complexe et fortement dépendante de la qualité des données.

— Il est intéressant de se concentrer sur les longs reads, certaines technologies permettant d'améliorer de plus en plus leur qualité en termes de taux d'erreurs. On peut penser qu'à l'avenir, ce type de reads sera plus largement utilisé.

— Après étude de l'état de l'art, nous nous sommes rendu compte de la difficulté du passage à l'échelle des solutions proposées. Les méthodes basées sur le hachage restent très coûteuses en mémoire, mais permettent l'association d'information et des requêtes simples. Les méthodes textuelles ont un coût de construction élevé, mais sont relativement compressibles. Nous avons donc naturellement pensé à développer un index qui puisse répondre à ces problématiques.

Afin de répondre à cette problématique, nous avons fait le choix de partir d'une solution extrêmement naïve, pour l'améliorer au cours du temps. Cette manière de procéder, ne cherchant pas une solution optimale immédiate, nous a permis de nous rendre compte à chaque étape de l'élément "bloquant" (mémoire RAM utilisée, temps nécessaire à l'indexation) pour mieux cibler les améliorations nécessaires.

À l'issue d'une première version fonctionnelle, résumée en Figure 3.5, nous avions déjà pu obtenir des résultats encourageants, en comparaison avec le r-index et Short Read Connector. Étant compétitifs en temps, l'utilisation mémoire restait restrictive à cause des redondances dans les identifiants de read stockés. Nous avons donc décidé de changer d'approche en combinant plusieurs maps, et en mettant en place une structure colorée préférant les minimizers aux k-mers, résumée en Figure 3.18. Nous avons cette fois-ci réussi à développer un index non seulement rapide, mais également bien moins coûteux en mémoire en évitant un maximum de redondance. Bien qu'à la base imaginé pour être utilisé majoritairement sur des longs reads, les tests réalisés sur des datasets Illumina ont aussi été concluants, en comparaison par exemple à SRC, spécialisé pour ce type de données.

5.2 Perspectives

Bien que ces trois années de thèse m'ont permis de participer à la publication de deux outils, de nombreuses questions/applications/développements sont possibles à la suite de ce travail.

Par exemple, un aspect à mon sens essentiel, du développement d'un outil tel que K2R n'a pas pu être réalisé, faute de temps : son application directe dans le cadre d'un projet concret. Plusieurs domaines peuvent être concernés par son application :

— La métagénomique, dans le but de différencier les différents génomes. L'idée étant de rechercher des séquences caractéristiques à certaines espèces dans le jeu de données, afin d'identifier celles présentes. Ce type d'analyse peut représenter un défi majeur en termes de passage à l'échelle, que ça soit en indexation comme en requête.

— Transcriptomique : il aurait également été pertinent d'évaluer K2R dans le cadre de l'analyse de l'expression génique. Un des principaux défis vient de la variabilité des niveaux d'expression : certains gènes sont très fortement exprimés, ce qui entraîne une fréquence élevée de certains k-mers, tandis que d'autres gènes faiblement exprimés génèrent des k-mers très rares. Dans ce contexte, l'utilisation des filtres d'abondance doit être très réfléchie : sans filtrage, les longues listes d'identifiants de reads qui pourraient apparaître dans les couleurs seraient prohibitives pour le passage à l'échelle. Par contre, un filtrage trop strict supprimerait de l'information cruciale.

— L'assemblage : L'indication des reads dans lesquels sont présents les k-mers peut être utilisée pour chercher les overlaps entre reads. Les k-mers présents dans les mêmes reads pourront être regroupés, puis assemblés. Les éléments ayant des minimizers identiques seront probablement successifs.

— Le mapping : K2R peut aussi aider au mapping, en réalisant une étape préliminaire de filtrage. Si une séquence courte à mapper n'a aucun minimizer commun avec la séquence cible, celle-ci peut ne plus être considérée par la suite, le dataset est donc réduit. La comparaison des minimizers présents sur la séquence avec les identifiants de reads les contenant peut aussi aider à leur localisation.

Sur le plan technique, K2R pourrait être complété par l'ajout d'une fonctionnalité de réversibilité, permettant de reconstruire le jeu de reads initial à partir de l'index. Une telle option n'est pas envisageable dans l'état actuel, en raison de l'usage des minimizers : un même minimizer peut être partagé par plusieurs k-mers, la reconstruction devient ambiguë.

Rendre la réversibilité possible nécessiterait de conserver directement l'information sur les k-mers eux-mêmes (et non les minimizers), ainsi que leur provenance (les reads contenant ces k-mers). Sur cette base, un assemblage des reads pourrait être envisagé. Néanmoins, la présence de régions répétées complique encore le problème : il faudrait alors, en plus de la provenance, enregistrer la position exacte des k-mers afin de lever les ambiguïtés. L'ajout de ces informations supplémentaires alourdirait logiquement l'utilisation mémoire, ce qui pose le défi de créer une structure capable de passer à l'échelle dans ce contexte.

Une autre fonctionnalité, déjà envisageable techniquement, consisterait à permettre la mise à jour d'un index en y intégrant de nouvelles données après sa création initiale. Cette approche réduirait considérablement les surcoûts en temps et en mémoire liés à la reconstruction répétée d'index, tout en facilitant une actualisation continue des données. De la même manière, la possibilité de fusionner plusieurs index partiels offrirait une alternative efficace à la construction d'un index unique de grande taille. En permettant la création indépendante de sous-index peu coûteux à produire, cette stratégie rendrait leur combinaison ultérieure bien plus rapide.

Actuellement, notre index est dynamique, c'est à dire que la structure peut être techniquement modifiée une fois créée. Cependant cette dynamicité a un coût : nous avons été obligés d'utiliser une structure telle que la table de hachage, qui stocke à la fois clé et valeur. Un gain en mémoire pourrait être obtenu en acceptant une structure statique, telle qu'une MPHF. La structure entière pourrait être repensée pour ce cadre statique. Par exemple, on pourrait souhaiter compresser certaines couleurs similaires entre elles, pour obtenir un gain sur cette étape.

Par ailleurs, l'évolution des technologies de séquençage introduit de nouvelles perspectives. Les dernières méthodes tendent à réduire le taux d'erreurs, notamment pour les longs reads, tout en augmentant le volume de données produites (le nombre de reads). La technologie SBX (Sequencing by Expansion) (64) en est un exemple : en "étirant" la molécule, elle évite les chevauchements de signal et améliore la lisibilité, réduisant ainsi le nombre d'erreurs générées. Cette technologie promet également une augmentation considérable du débit possible : jusqu'à 500Mb par seconde. De leur côté, les approches de séquençage "3D" `https://www.genengnews.com/gen-edge/could-single-technologies-3d-sequencing-deliver-the-10-genome/` reposent sur l'utilisation d'une matrice volumique plutôt qu'un flux 2D classique, ce qui augmente considérablement la capacité de parallélisme et donc le flux de lecture. Cette amélioration entraîne une augmentation des reads créés (jusqu'au trillion de reads), qui deviennent potentiellement plus longs, mais entraîne aussi de nouveaux défis en matière de stockage, de temps de calcul et d'optimisation. Dans ce contexte, l'usage de GPU est une piste possible : leur capacité de calcul parallèle les rend particulièrement adaptés au traitement des signaux générés lors du séquençage.

Enfin, des pistes d'amélioration basées uniquement sur l'implémentation peuvent également être envisagées. Par exemple, une combinaison entre K2R pour l'indexation et K2Rmini pour les requêtes pourrait être mise en place. En regroupant un nombre fixe de reads en leur attribuant un identifiant unique lors de l'indexation, le nombre de couleurs serait réduit, ce qui produirait des listes d'identifiants plus courtes et mieux compressibles. Dans le même temps, l'efficacité des requêtes de K2Rmini garantirait que la recherche dans ces ensembles fixes de reads reste performante.

Un autre axe d'exploration concerne l'indexation full-text, en particulier basé sur la transformée de Burrows-Wheeler. En théorie, une telle approche pourrait permettre de réaliser des filtrages de reads d'intérêt similaires à ceux proposés par K2R. Des optimisations notables ont déjà été développées pour améliorer leur passage à l'échelle : l'introduction du r-index a représenté une avancée majeure, tandis que le br-index (bidirectionnel), dérivé du *r*-index, a encore renforcé les performances de recherche.

Cependant, ces structures présentent plusieurs limites. Leur construction reste coûteuse en calcul et la gestion des erreurs demeure un point faible, puisqu'elles indexent le texte complet sans filtrage préalable. Actuellement, elles sont bien adaptées aux génomes entiers, peu bruités et de taille gérable, mais beaucoup moins aux ensembles massifs de reads longs. Pour ces derniers, les approches basées sur les *k*-mers restent indispensables. Néanmoins, avec la diminution progressive des taux d'erreurs apportée par les technologies de séquençage récentes, il est possible que ces méthodes retrouvent de la pertinence. L'impact mémoire constituerait toutefois un frein important.

Un prétraitement des données pourrait être une solution pour pallier ces difficultés. Par exemple, on pourrait limiter l'impact des erreurs en appliquant un premier filtrage. Une combinaison K2R/full-text serait alors intéressante : K2R pourrait éliminer les *k*-mers très peu ou

trop abondants, et l'index full-text serait ensuite construit sur un jeu de données déjà nettoyé. On conserverait ainsi les atouts du full-text, en particulier la possibilité d'effectuer des requêtes sur n'importe quel motif, sans contrainte de taille de *k*-mer.

Des perspectives d'évolution peuvent également concerner les outils associés au projet K2R : OReO et K2Rmini.

OReO, qui réordonne les reads d'un jeu de données afin de regrouper ceux qui se chevauchent, repose actuellement sur une étape d'assemblage à partir d'un graphe de contigs, ce qui peut s'avérer coûteux. Une alternative consisterait à remplacer cette étape par une approche plus légère. Par exemple, l'utilisation d'ONIKA, qui transforme les reads en sketchs (petit sous-ensemble fixe de *k*-mers, souvent les plus petits en valeur), pourrait être réinvestie dans ce contexte. La question de comment optimiser l'ordre des reads à partir d'une matrice de similarité est ouverte.

De son côté, K2Rmini a été conçu pour éviter l'indexation de jeux de données complets, en particulier lorsque le nombre de requêtes est limité. On peut aussi envisager son utilisation dans un objectif différent : réduire la charge liée à l'indexation de très grands ensembles de données, comme dans le cas de LOGAN (30), qui génère plusieurs centaines de téraoctets. Dans ce système, les *k*-mers sont stockés dans des filtres de Bloom, puis l'outil Back to Sequences (BTS) permet de retrouver les identifiants d'accession associés à une requête. Une alternative évidente consisterait à remplacer BTS par K2Rmini, qui pourrait offrir de meilleures performances pour ce type de tâches.

5.3 Déroulé de la thèse et perspectives personnelles

Cette thèse s'inscrit dans le prolongement de ma réorientation vers le master de bioinformatique, après l'obtention d'un master en informatique. Ma volonté d'appliquer mes compétences informatiques à un domaine qui m'intéressait particulièrement, la biologie, s'est trouvée possible avec l'ouverture de cette formation à ce moment précis. Deux stages effectués au sein du laboratoire BONSAI ont confirmé mon souhait de poursuivre en doctorat.

Initialement, ce travail de thèse se voulait résolument technique, avec une forte composante de développement. Cet aspect a d'ailleurs été, à mes yeux, le plus enrichissant, car il m'a permis d'approfondir mes connaissances en informatique tout en les mettant en œuvre dans un contexte biologique, grâce aux jeux de données analysés. Toutefois, l'intégration dans un nouveau domaine comme la biologie ne s'est pas faite sans difficultés. Je suis consciente de ne pas posséder une expertise approfondie sur certains aspects biologiques, ce qui a parfois limité mes perspectives d'application. En particulier, une fois le développement de K2R terminé, ma volonté première était de poursuivre ma thèse en l'appliquant directement sur des données biologiques. Cette étape m'aurait forcée à me former plus en profondeur en biologie, notamment pour mieux maîtriser les impacts qu'il peut y avoir sur les jeux de données. Le contact avec des biologistes aurait pu être un atout précieux dans ce sens. Peut-être que des collaborations de ce type auraient pu nous mener à d'autres idées de poursuite, que ça soit sur l'amélioration continue de K2R ou une toute autre idée d'outil.

Durant ces trois années j'ai eu l'opportunité plusieurs fois d'exposer mon travail lors de conférences. J'ai par exemple pu présenter K2R lors de *k*-mer days (méthodes *k*-mers principalement pour la transcriptomique), DSB (Data Structure in Bioinformatics), et seqBIM (journées spécialisées pour l'algorithmique des séquences) sous forme de présentations courtes. Consciente que

ce type de présentation orale est un point faible pour moi, ces oraux ont pour moi été l'occasion de m'exercer et de prendre confiance en vue de la future soutenance de thèse. Cela m'a également permis de pouvoir discuter avec la communauté de bioinformatique, et de m'intéresser aux différents travaux en cours.

Ma thèse ne s'est pas limitée au travail de recherche, mais a été également l'opportunité pour moi d'avoir une première expérience en enseignement. J'ai eu la chance de pouvoir enseigner durant 3 semestres (90 heures) à la FST (Faculté de Sciences et Technologies de Lille) à des étudiants de première année de licence Maths/Informatique ou MIASHS (Mathématiques et Informatique Appliquées aux Sciences Humaines et Sociales). Les enseignements donnés étaient portés soit sur les bases de la programmation et de l'algorithmie (en Python), soit sur le codage de l'information, avec notamment la représentation des nombres.

J'ai trouvé cette expérience extrêmement enrichissante, tant sur le plan du contact humain que sur les compétences pédagogiques que j'ai pu acquérir. J'ai cependant préféré ne pas renouveler l'expérience jusqu'en dernière année, afin de me consacrer entièrement à l'écriture de mon papier principal et sur mon manuscrit de thèse. Bien que mon nombre d'heures d'enseignement par semaine était limité, le temps nécessaire à leur préparation ainsi qu'aux corrections restait important, et aurait ajouté une difficulté supplémentaire à cette dernière année déjà très chargée.

De ce fait, il ne me conviendrait pas de poursuivre ma carrière dans l'académique. Je pense poursuivre sur un poste plus technique. Soit en développement pour continuer à me former dans ce domaine après mon premier master et le développement que j'ai pu faire durant cette thèse, soit en gestion de données à la suite de l'alternance que j'avais pu faire en 2020. Dans tous les cas, l'informatique restera au coeur de mon projet professionnel.

Bibliographie

[1] Clément Agret, Bastien Cazaux, and Antoine Limasset. Toward Optimal Fingerprint Indexing for Large Scale Genomics. In Christina Boucher and Sven Rahmann, editors, *22nd International Workshop on Algorithms in Bioinformatics (WABI 2022)*, volume 242 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 25 :1–25 :15, Dagstuhl, Germany, 2022. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. 92, 102

[2] Jarno N. Alanko, Simon J. Puglisi, and Jaakko Vuohtoniemi. Succinct k-mer set representations using subset rank queries on the spectral burrows-wheeler transform (sbwt). *bioRxiv*, 2022. 35

[3] Jarno N Alanko, Jaakko Vuohtoniemi, Tommi Mäklin, and Simon J Puglisi. Themisto : a scalable colored k-mer index for sensitive pseudoalignment against hundreds of thousands of bacterial genomes. *Bioinformatics*, 39(Supplement_1) :i260–i269, 06 2023. 43, 72

[4] Muhammad Ali, Derek B Archer, Priyanka Gorijala, Daniel Western, Jigyasha Timsina, Maria V Fernández, Ting-Chen Wang, Claudia L Satizabal, Qiong Yang, Alexa S Beiser, Ruiqi Wang, Gengsheng Chen, Brian Gordon, Tammie L S Benzinger, Chengjie Xiong, John C Morris, Randall J Bateman, Celeste M Karch, Eric McDade, Alison Goate, Sudha Seshadri, Richard P Mayeux, Reisa A Sperling, Rachel F Buckley, Keith A Johnson, Hong-Hee Won, Sang-Hyuk Jung, Hang-Rai Kim, Sang Won Seo, Hee Jin Kim, Elizabeth Mormino, Simon M Laws, Kang-Hsien Fan, M Ilyas Kamboh, Prashanthi Vemuri, Vijay K Ramanan, Hyun-Sik Yang, Allen Wenzel, Hema Sekhar Reddy Rajula, Aniket Mishra, Carole Dufouil, Stephanie Debette, Oscar L Lopez, Steven T DeKosky, Feifei Tao, Michael W Nagle, Knight Alzheimer Disease Research Center (Knight ADRC), Dominantly Inherited Alzheimer Network (DIAN), Alzheimer's Disease Neuroimaging Initiative (ADNI), ADNI-DOD, A4 Study Team, Australian Imaging Biomarkers, Lifestyle (AIBL) Study, Timothy J Hohman, Yun Ju Sung, Logan Dumitrescu, and Carlos Cruchaga. Large multi-ethnic genetic analyses of amyloid imaging identify new genes for alzheimer disease. *Acta Neuropathol Commun*, 11(1) :68, April 2023. 3

[5] Fatemeh Almodaresi, Prashant Pandey, and Robert Patro. Rainbowfish : A succinct colored de bruijn graph representation, 05 2017. 35, 54

[6] Fatemeh Almodaresi, Hirak Sarkar, Avi Srivastava, and Rob Patro. A space and time-efficient index for the compacted colored de bruijn graph. *Bioinformatics*, 34(13) :i169–i177, 06 2018. 38

[7] Stephen F. Altschul, Warren Gish, Webb Miller, Eugene W. Myers, and David J. Lipman. Basic local alignment search tool. *Journal of Molecular Biology*, 215(3) :403–410, 1990. 9

[8] Yuma Arakawa, Gonzalo Navarro, and Kunihiko Sadakane. Bi-Directional r-Indexes. In Hideo Bannai and Jan Holub, editors, *33rd Annual Symposium on Combinatorial Pattern Matching (CPM 2022)*, volume 223 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 11 :1–11 :14, Dagstuhl, Germany, 2022. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. 35, 72

[9] R. Arnold and T. Bell. A corpus for the evaluation of lossless compression algorithms. In *Proceedings DCC '97. Data Compression Conference*, pages 201–210, 1997. 32

[10] Lorraine A. K. Ayad, Gabriele Fici, Ragnar Groot Koerkamp, Grigorios Loukides, Rob Patro, Giulio Ermanno Pibiri, and Solon P. Pissis. U-index : A universal indexing framework for matching long patterns, 2025. 28

[11] Anthony Baire, Pierre Marijon, Francesco Andreace, and Pierre Peterlongo. Back to sequences : Find the origin of k-mers. *Journal of Open Source Software*, 9(101) :7066, 2024. 92, 109, 111

[12] Daniel N. Baker and Ben Langmead. Dashing : fast and accurate genomic distances with hyperloglog. *Genome Biology*, 20(1) :265, Dec 2019. 99

[13] Daniel N. Baker and Ben Langmead. Dashing 2 : genomic sketching with multiplicities and locality-sensitive hashing. *bioRxiv*, 2022. 103

[14] Anton Bankevich, Andrey V Bzikadze, Mikhail Kolmogorov, Dmitry Antipov, and Pavel A Pevzner. Multiplex de bruijn graphs enable genome assembly from long, high-fidelity reads. *Nature biotechnology*, 40(7) :1075–1081, 2022. 70

[15] Sam Behjati and Patrick S Tarpey. What is next generation sequencing ? *Arch Dis Child Educ Pract Ed*, 98(6) :236–238, August 2013. 11

[16] Gaëtan Benoit, Claire Lemaitre, Dominique Lavenier, Erwan Drezen, Thibault Dayris, Raluca Uricaru, and Guillaume Rizk. Reference-free compression of high throughput sequencing data with a probabilistic de Bruijn graph. *BMC Bioinformatics*, 16(1), September 2015. 22

[17] Bieganski, Riedl, Cartis, and Retzel. Generalized suffix trees for biological sequence data : applications and implementation. In *1994 Proceedings of the Twenty-Seventh Hawaii International Conference on System Sciences*, volume 5, pages 35–44, 1994. 28

[18] Lorenzo Bomba, Klaudia Walter, and Nicole Soranzo. The impact of rare and low-frequency genetic variants in common disease. *Genome Biology*, 18(1) :77, Apr 2017. 2

[19] James K Bonfield and Matthew V Mahoney. Compression of FASTQ and SAM format sequencing data. *PLoS One*, 8(3) :e59190, March 2013. 93

[20] Tungadri Bose, Monzoorul Haque Mohammed, Anirban Dutta, and Sharmila S Mande. BIND - an algorithm for loss-less compression of nucleotide sequence data. *J Biosci*, 37(4) :785–789, September 2012. 93

[21] Alex Bowe, Taku Onodera, Kunihiko Sadakane, and Tetsuo Shibuya. Succinct de bruijn graphs. pages 225–235, 09 2012. 35

[22] Phelim Bradley, Henk den Bakker, Eduardo Rocha, Gil McVean, and Zamin Iqbal. Ultrafast search of all deposited bacterial and viral genomic data. *Nature Biotechnology*, 37 :152–159, 02 2019. 44

[23] Nicolas L. Bray, Harold Pimentel, Páll Melsted, and Lior Pachter. Near-optimal probabilistic rna-seq quantification. *Nature Biotechnology*, 34(5) :525–527, May 2016. 21, 61

[24] Karel Břinda, Michael Baym, and Gregory Kucherov. Simplitigs as an efficient and scalable representation of de bruijn graphs. *Genome Biology*, 22(1) :96, Apr 2021. 39

[25] Andrei Z. Broder. On the resemblance and containment of documents. In *Proceedings of the Compression and Complexity of Sequences 1997*, pages 21–29. IEEE, 1997. 99

[26] Michael Burrows and David J. Wheeler. A block-sorting lossless data compression algorithm. 1994. 31, 32

[27] Shubham Chandak, Kedar Tatwawadi, and Tsachy Weissman. Compression of genomic sequencing reads via hash-based reordering : algorithm and analysis. *Bioinformatics*, 34(4) :558–567, 2018. 22

[28] Rayan Chikhi, Antoine Limasset, Shaun Jackman, Jared T Simpson, and Paul Medvedev. On the representation of de bruijn graphs. *Journal of Computational Biology*, 22(5) :336–352, 2015. 15

[29] Rayan Chikhi, Antoine Limasset, and Paul Medvedev. Compacting de bruijn graphs from sequencing data quickly and in low memory. *Bioinformatics*, 32(12) :i201–i208, June 2016. 21, 41

[30] Rayan Chikhi, Brice Raffestin, Anton Korobeynikov, Robert Edgar, and Artem Babaian. Logan : Planetary-scale genome assembly surveys life's diversity. *bioRxiv*, 2024. 109, 118

[31] Rayan Chikhi and Guillaume Rizk. Space-efficient and exact de bruijn graph representation based on a bloom filter. *Algorithms for Molecular Biology*, 8(1) :22, Sep 2013. 41

[32] Anthony J. Cox, Markus J. Bauer, Tobias Jakobi, and Giovanna Rosone. Large-scale compression of genomic sequence databases with the burrows-wheeler transform. *CoRR*, abs/1205.0192, 2012. 22, 93

[33] Andrea Cracco and Alexandru I Tomescu. Extremely fast construction and querying of compacted and colored de bruijn graphs with GGCAT. *Genome Res*, 33(7) :1198–1207, May 2023. 21, 41

[34] Nicholas D. Crosbie. grepq : A rust application that quickly filters fastq files by matching sequences to a set of regular expressions. *bioRxiv*, 2025. 111

[35] Kenny Daily, Paul Rigor, Scott Christley, Xiaohui Xie, and Pierre Baldi. Data structures and compression algorithms for high-throughput sequencing technologies. *BMC Bioinformatics*, 11(1) :514, October 2010. 93

[36] Mitra Darvish, Enrico Seiler, Svenja Mehringer, René Rahn, and Knut Reinert. Needle : a fast and space-efficient prefilter for estimating the quantification of very large collections of expression experiments. *Bioinformatics*, 38(17) :4100–4108, 2022. 46

[37] Arthur L. Delcher, Simon Kasif, Robert D. Fleischmann, Jeremy Peterson, Owen White, and Steven L. Salzberg. Alignment of whole genomes. *Nucleic Acids Research*, 27(11) :2369–2376, 01 1999. 9

[38] Gennady Denisov, Brian Walenz, Aaron L. Halpern, Jason Miller, Nelson Axelrod, Samuel Levy, and Granger Sutton. Consensus generation and variant detection by celera assembler. *Bioinformatics*, 24(8) :1035–1040, 03 2008. 15

[39] Sebastian Deorowicz and Szymon Grabowski. Compression of dna sequence reads in fastq format. *Bioinformatics*, 27(6) :860–862, 01 2011. 93

[40] Wolfgang Enard, Molly Przeworski, Simon E. Fisher, Cecilia S. L. Lai, Victor Wiebe, Takashi Kitano, Anthony P. Monaco, and Svante Pääbo. Molecular evolution of foxp2, a gene involved in speech and language. *Nature*, 418(6900) :869–872, Aug 2002. 4

[41] Otmar Ertl. Setsketch : Filling the gap between minhash and hyperloglog. *CoRR*, abs/2101.00314, 2021. 103

[42] Mikhail Karasikov et al. Efficient and accurate search in petabase-scale sequence repositories. *Nature*, 2025. Online ahead of print. 46

[43] Jason Fan, Jamshed Khan, Noor Pratap Singh, Giulio Ermanno Pibiri, and Rob Patro. Fulgor : a fast and compact k-mer index for large-scale matching and color queries. *Algorithms for Molecular Biology*, 19(1) :3, Jan 2024. 43, 72

[44] Paolo Ferragina and Giovanni Manzini. Opportunistic data structures with applications. volume 2000, pages 390–398, 02 2000. 32

[45] fulcrumgenomics. fqgrep. `https://github.com/fulcrumgenomics/fqgrep.git`, 2025. Accessed : 2025-06-02. 111

[46] Andrew Gallant. ripgrep. `https://github.com/BurntSushi/ripgrep`, 2024. Accessed : 2025-06-02. 111

[47] Jérémy Gauthier, Charlotte Mouden, Tomasz Suchan, Nadir Alvarez, Nils Arrigo, Chloé Riou, Claire Lemaitre, and Pierre Peterlongo. DiscoSnp-RAD : de novo detection of small variants for RAD-Seq population genomics. *PeerJ*, 8 :e9291, June 2020. 20

[48] Mathilde Girard, Léa Vandamme, Bastien Cazaux, and Antoine Limasset. OReO : optimizing read order for practical compression. *Bioinform Adv*, 5(1) :vbaf128, June 2025. 91, 93

[49] Szymon Grabowski, Sebastian Deorowicz, and Łukasz Roguski. Disk-based compression of data from genome sequencing. *Bioinformatics*, 31(9) :1389–1395, 2015. 22

[50] Ragnar Groot Koerkamp. A*PA2 : Up to 19× Faster Exact Global Alignment. In Solon P. Pissis and Wing-Kin Sung, editors, *24th International Workshop on Algorithms in Bioinformatics (WABI 2024)*, volume 312 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 17 :1–17 :25, Dagstuhl, Germany, 2024. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. 7

[51] Roberto Grossi, Ankur Gupta, and Jeffrey Scott Vitter. High-order entropy-compressed text indexes. In *Proceedings of the Fourteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '03, page 841–850, USA, 2003. Society for Industrial and Applied Mathematics. 34

[52] Robert S Harris and Paul Medvedev. Improved representation of sequence bloom trees. *Bioinformatics*, 36(3) :721–727, feb 2020. 44

[53] Yukun He, Yanan Chu, Shuming Guo, Jiang Hu, Ran Li, Yali Zheng, Xinqian Ma, Zhenglin Du, Lili Zhao, Wenyi Yu, et al. T2t-yao : a telomere-to-telomere assembled diploid reference genome for han chinese. *Genomics, Proteomics & Bioinformatics*, 21(6) :1085–1100, 2023. 13

[54] Guillaume Holley and Páll Melsted. Bifrost : highly parallel construction and indexing of colored and compacted de bruijn graphs. *Genome Biology*, 21(1) :249, Sep 2020. 38, 43

[55] Markus Hsi-Yang Fritz, Rasko Leinonen, Guy Cochrane, and Ewan Birney. Efficient storage of high throughput DNA sequencing data using reference-based compression. *Genome Res*, 21(5) :734–740, January 2011. 93

[56] X Huang and A Madan. CAP3 : A DNA sequence assembly program. *Genome Res*, 9(9) :868–877, September 1999. 15

[57] Florian Ingels, Léa Vandamme, Mathilde Girard, Clément Agret, Bastien Cazaux, and Antoine Limasset. Compressed inverted indexes for scalable sequence similarity. *bioRxiv*, 2025. 92

[58] Chirag Jain, Alexander Dilthey, Sergey Koren, Srinivas Aluru, and Adam M Phillippy. A fast approximate algorithm for mapping long reads to large reference databases. In *International Conference on Research in Computational Molecular Biology*, pages 66–81. Springer, 2017. 61

[59] Chirag Jain, Arang Rhie, Haowen Zhang, Claudia Chu, Brian P Walenz, Sergey Koren, and Adam M Phillippy. Weighted minimizer sampling improves long read mapping. *Bioinformatics*, 36(Suppl_1) :i111–i118, July 2020. 63

[60] Daniel C Jones, Walter L Ruzzo, Xinxia Peng, and Michael G Katze. Compression of next-generation sequencing reads aided by highly efficient de novo assembly. *Nucleic acids research*, 40(22) :e171–e171, 2012. 22

[61] Mikhail Karasikov, Harun Mustafa, Daniel Danciu, Oleksandr Kulkov, Marc Zimmermann, Christopher Barber, Gunnar Rätsch, and André Kahles. Efficient and accurate search in petabase-scale sequence repositories. *Nature*, Oct 2025. 1

[62] Jamshed Khan and Rob Patro. Cuttlefish : fast, parallel and low-memory compaction of de bruijn graphs from large-scale genome collections. *Bioinformatics*, 37(Supplement_1) :i177–i186, 07 2021. 41

[63] Ragnar Groot Koerkamp and Igor Martayan. Simdminimizers : Computing random minimizers, fast. *bioRxiv*, 2025. 109

[64] Mark Kokoris, Robert McRuer, Melud Nabavi, Aaron Jacobs, Marc Prindle, Cynthia Cech, Kendall Berg, Taylor Lehmann, Cara Machacek, John Tabone, Jagadeeswaran Chandrasekar, Lacey McGee, Matthew Lopez, Tommy Reid, Cara Williams, Salka Barrett, Alex Lehmann, Michael Kovarik, Robert Busam, Scott Miller, Brent Banasik, Brittany Kesic, Anasha Arryman, Megan Rogers-Peckham, Alan Kimura, Megan LeProwse, Mitchell Wolfin, Svetlana Kritzer, Joanne Leadbetter, Majid Babazadeh, John Chase, Greg Thiessen, William Lint, Drew Goodman, Dylan O'Connell, Nadya Lumanpauw, John Hoffman, Samantha Vellucci, Kendra Collins, Jessica Vellucci, Amy Taylor, Molly Murphy, Michael Lee, and Matthew Corning. Sequencing by expansion (sbx) – a novel, high-throughput single-molecule sequencing technology. *bioRxiv*, 2025. 117

[65] Marek Kokot, Adam Gudyś, Heng Li, and Sebastian Deorowicz. CoLoRd : compressing long reads. *Nat Methods*, 19(4) :441–444, March 2022. 93

[66] Sergey Koren, Brian P Walenz, Konstantin Berlin, Jason R Miller, Nicholas H Bergman, and Adam M Phillippy. Canu : scalable and accurate long-read assembly via adaptive k-mer weighting and repeat separation. *Genome Res*, 27(5) :722–736, March 2017. 15

[67] T. W. Lam, W. K. Sung, S. L. Tam, C. K. Wong, and S. M. Yiu. Compressed indexing and local alignment of dna. *Bioinformatics*, 24(6) :791–797, 01 2008. 27

[68] Ben Langmead and Steven L. Salzberg. Fast gapped-read alignment with bowtie 2. *Nature Methods*, 9(4) :357–359, Apr 2012. 9, 27

[69] Ben Langmead, Cole Trapnell, Mihai Pop, and Steven L Salzberg. Ultrafast and memory-efficient alignment of short DNA sequences to the human genome. *Genome Biology*, 10(3) :R25, March 2009. 9, 11

[70] Hayan Lee, James Gurtowski, Shinjae Yoo, Shoshana Marcus, W. Richard McCombie, and Michael Schatz. Error correction and assembly complexity of single molecule sequencing reads. *bioRxiv*, 2014. 14

[71] Heng Li. Minimap and miniasm : fast mapping and de novo assembly for noisy long sequences. *Bioinformatics*, 32(14) :2103–2110, 03 2016. 15

[72] Heng Li. Minimap2 : pairwise alignment for nucleotide sequences. *Bioinformatics*, 34(18) :3094–3100, 05 2018. 12, 63

[73] Heng Li. New strategies to improve minimap2 alignment accuracy. *Bioinformatics*, 37(23) :4572–4574, December 2021. 10

[74] Heng Li and Richard Durbin. Fast and accurate short read alignment with burrows–wheeler transform. *Bioinformatics*, 25(14) :1754–1760, 05 2009. 9, 11, 27

[75] Ruiqiang Li, Chang Yu, Yingrui Li, Tak-Wah Lam, Siu-Ming Yiu, Karsten Kristiansen, and Jun Wang. Soap2 : an improved ultrafast tool for short read alignment. *Bioinformatics*, 25(15) :1966–1967, 06 2009. 9

[76] Yanbo Li and Yu Lin. Kmer2snp : reference-free snp calling from raw reads based on matching. *bioRxiv*, 2020. 20

[77] Antoine Limasset, Jean-François Flot, and Pierre Peterlongo. Toward perfect reads : self-correction of short reads via mapping on de Bruijn graphs. *Bioinformatics*, 36(5) :1374–1381, 02 2019. 21

[78] Antoine Limasset, Guillaume Rizk, Rayan Chikhi, and Pierre Peterlongo. Fast and scalable minimal perfect hashing for massive key sets, 2017. 38

[79] Antoine Limasset, Guillaume Rizk, Rayan Chikhi, and Pierre Peterlongo. Fast and Scalable Minimal Perfect Hashing for Massive Key Sets. In Costas S. Iliopoulos, Solon P. Pissis, Simon J. Puglisi, and Rajeev Raman, editors, *16th International Symposium on Experimental Algorithms (SEA 2017)*, volume 75 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 25 :1–25 :16, Dagstuhl, Germany, 2017. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. 72

[80] Yuansheng Liu, Zuguo Yu, Marcel E Dinger, and Jinyan Li. Index suffix–prefix overlaps by (w, k)-minimizer to generate long contigs for reads compression. *Bioinformatics*, 35(12) :2066–2074, 2019. 22

[81] Hengyun Lu, Francesca Giordano, and Zemin Ning. Oxford nanopore MinION sequencing and genome assembly. *Genomics Proteomics Bioinformatics*, 14(5) :265–279, September 2016. 11

[82] Nicolas Maillet, Guillaume Collet, Thomas Vannier, Dominique Lavenier, and Pierre Peterlongo. Commet : Comparing and combining multiple metagenomic datasets. In *2014 IEEE International Conference on Bioinformatics and Biomedicine (BIBM)*, pages 94–98, 2014. 21

[83] Udi Manber and Gene Myers. Suffix arrays : a new method for on-line string searches. In *Proceedings of the First Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '90, page 319–327, USA, 1990. Society for Industrial and Applied Mathematics. 30

[84] Camille Marchet et al. Reindeer2 : practical abundance index at scale. *bioRxiv*, 2025. Preprint. 45

[85] Camille Marchet, Zamin Iqbal, Daniel Gautheret, Mikaël Salson, and Rayan Chikhi. Reindeer : efficient indexing of k-mer presence and abundance in sequencing datasets. *Bioinformatics*, 36(Supplement_1) :i177–i185, 2020. 21, 45

[86] Camille Marchet, Mael Kerbiriou, and Antoine Limasset. Blight : efficient exact associative structure for k-mers. *Bioinformatics*, 37(18) :2858–2865, 04 2021. 38, 43

[87] Camille Marchet, Lolita Lecompte, Antoine Limasset, Lucie Bittner, and Pierre Peterlongo. A resource-frugal probabilistic dictionary and applications in bioinformatics. *Discrete Applied Mathematics*, 274 :92–102, 2020. 51, 72

[88] Igor Martayan, Léa Vandamme, Bede Constantinides, Bastien Cazaux, Charles Paperman, and Antoine Limasset. Accelerating k-mer-based sequence filtering. *bioRxiv*, 2025. 92, 109

[89] Edward M. McCreight. A space-economical suffix tree construction algorithm. *J. ACM*, 23(2) :262–272, April 1976. 28

[90] Qingxi Meng, Shubham Chandak, Yifan Zhu, and Tsachy Weissman. Reference-free lossless compression of nanopore sequencing reads using an approximate assembly approach. *Scientific Reports*, 13(1) :2082, February 2023. 93

[91] Monzoorul Haque Mohammed, Anirban Dutta, Tungadri Bose, Sudha Chadaram, and Sharmila S. Mande. Deliminate—a fast and efficient method for loss-less compression of genomic sequences : Sequence analysis. *Bioinformatics*, 28(19) :2527–2529, 07 2012. 93

[92] Martin D Muggli, Alexander Bowe, Noelle R Noyes, Paul S Morley, Keith E Belk, Robert Raymond, Travis Gagie, Simon J Puglisi, and Christina Boucher. Succinct colored de bruijn graphs. *Bioinformatics*, 33(20) :3181–3187, October 2017. 35

[93] Taher Mun, Alan Kuhnle, Christina Boucher, Travis Gagie, Ben Langmead, and Giovanni Manzini. Matching reads to many genomes with the r-index. *J Comput Biol*, 27(4) :514–518, March 2020. 35, 51

[94] Eugene W Myers. The fragment assembly string graph. *Bioinformatics*, 21 Suppl 2 :ii79–85, September 2005. 15

[95] Joong Chae Na, Alberto Apostolico, Costas S. Iliopoulos, and Kunsoo Park. Truncated suffix trees and their application to data compression. *Theoretical Computer Science*, 304(1) :87–101, 2003. 29

[96] S B Needleman and C D Wunsch. A general method applicable to the search for similarities in the amino acid sequence of two proteins. *J Mol Biol*, 48(3) :443–453, March 1970. 6

[97] Takaaki Nishimoto and Yasuo Tabei. Optimal-time queries on bwt-runs compressed indexes. *arXiv preprint arXiv :2006.05104*, 2020. 35

[98] Sergey Nurk, Sergey Koren, Arang Rhie, Mikko Rautiainen, Andrey V Bzikadze, Alla Mikheenko, Mitchell R Vollger, Nicolas Altemose, Lev Uralsky, Ariel Gershman, et al. The complete sequence of a human genome. *Science*, 376(6588) :44–53, 2022. 13

[99] Brian D. Ondov, Todd J. Treangen, Páll Melsted, Adam B. Mallonee, Nicholas H. Bergman, Sergey Koren, and Adam M. Phillippy. Mash : fast genome and metagenome distance estimation using minhash. *Genome Biology*, 17(1) :132, Jun 2016. 21, 100

[100] Fatih Ozsolak. Third-generation sequencing techniques and applications to drug discovery. *Expert Opin Drug Discov*, 7(3) :231–243, February 2012. 11

[101] Prashant Pandey, Fatemeh Almodaresi, Michael A Bender, Michael Ferdman, Rob Johnson, and Rob Patro. Mantis : A fast, small, and exact Large-Scale Sequence-Search index. *Cell Syst*, 7(2) :201–207.e4, June 2018. 42, 43, 54, 55

[102] Prashant Pandey, Michael Bender, Rob Johnson, and Rob Patro. A general-purpose counting filter : Making every bit count. pages 775–787, 05 2017. 45

[103] Rob Patro and Carl Kingsford. Data-dependent bucketing improves reference-free compression of sequencing reads. *Bioinformatics*, 31(17) :2770–2777, 2015. 22

[104] David Pellow, Lianrong Pu, Barış Ekim, Lior Kotlar, Bonnie Berger, Ron Shamir, and Yaron Orenstein. Efficient minimizer orders for large values of k using minimum decycling sets. *Genome Research*, 33(7) :1154–1161, 2023. 58

[105] Pierre Peterlongo, Chloé Riou, Erwan Drezen, and Claire Lemaitre. Discosnp++ : de novo detection of small variants from raw unassembled read set(s). *bioRxiv*, 2017. 20

[106] N Tessa Pierce, Luiz Irber, Taylor Reiter, Phillip Brooks, and C Titus Brown. Large-scale sequence comparisons with sourmash. *F1000Res*, 8 :1006, July 2019. 21

[107] Jan Platos and Jiri Dvorsky. Word-based text compression, 2008. 34

[108] powturbo. Turbopfor-integer-compression. `https://github.com/powturbo/TurboPFor-Integer-Compression.git`, 2013-2019. 56

[109] Amatur Rahman and Paul Medvedev. Representation of k-mer sets using spectrum-preserving string sets. *bioRxiv*, 2020. 39

[110] Anthony Rhoads and Kin Fai Au. PacBio sequencing and its applications. *Genomics Proteomics Bioinformatics*, 13(5) :278–289, November 2015. 11

[111] Kristoffer Sahlin, Thomas Baudeau, Bastien Cazaux, and Camille Marchet. A survey of mapping algorithms in the long-reads era. *Genome Biology*, 24(1) :133, Jun 2023. 9

[112] Leena Salmela and Eric Rivals. Lordec : accurate and efficient long read error correction. *Bioinform.*, 30(24) :3506–3514, 2014. 21

[113] Coulson AR Sanger F, Nicklen S. Dna sequencing with chain-terminating inhibitors. *Proc Natl Acad Sci U S A*, December 1977. 11

[114] Ariya Shajii, Deniz Yorukoglu, Yun William Yu, and Bonnie Berger. Fast genotyping of known snps through approximate k-mer matching. *Bioinformatics*, 32(17) :i538–i544, 08 2016. 20

[115] Wei Shen, Botond Sipos, and Liuyang Zhao. Seqkit2 : A swiss army knife for sequence and alignment processing. *iMeta*, 3(3) :e191, 2024. 111

[116] T F Smith and M S Waterman. Identification of common molecular subsequences. *J Mol Biol*, 147(1) :195–197, March 1981. 5

[117] Brad Solomon and Carl Kingsford. Fast search of thousands of short-read sequencing experiments. *Nature Biotechnology*, 34(3) :300–302, Mar 2016. 44

[118] Brad Solomon and Carl Kingsford. Improved search of large transcriptomic sequencing databases using split sequence bloom trees. *J Comput Biol*, 25(7) :755–765, March 2018. 44

[119] Martin Šošic and Mile Šikic. Edlib : a C/C ++ library for fast, exact sequence alignment using edit distance. *Bioinformatics*, 33(9) :1394–1395, May 2017. 8

[120] E. Ukkonen. On-line construction of suffix trees. *Algorithmica*, 14(3) :249–260, Sep 1995. 28

[121] Raluca Uricaru, Guillaume Rizk, Vincent Lacroix, Elsa Quillery, Olivier Plantard, Rayan Chikhi, Claire Lemaitre, and Pierre Peterlongo. Reference-free detection of isolated SNPs. *Nucleic Acids Research*, 43(2) :e11–e11, 11 2014. 20

[122] Léa Vandamme, Bastien Cazaux, and Antoine Limasset. K2r : Tinted de bruijn graphs implementation for efficient read extraction from sequencing datasets. *bioRxiv*, 2024. 46, 89

[123] Alexander B. Veretennikov. Proximity full-text search with a response time guarantee by means of additional indexes with multi-component keys. *CoRR*, abs/1812.07640, 2018. 28

[124] Rongjie Wang, Junyi Li, Yang Bai, Tianyi Zang, and Yadong Wang. Bdbg : a bucket-based method for compressing genome sequencing data with dynamic de bruijn graphs. *PeerJ*, 6 :e5611, 2018. 22

[125] Xiang Wang, Yang Hong, Harry Chang, KyoungSoo Park, Geoff Langdale, Jiayu Hu, and Heqing Zhu. Hyperscan : A fast multi-pattern regex matcher for modern {CPUs}. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 631–648, 2019. 111

[126] Peter Weiner. Linear pattern matching algorithm. pages 1–11, 11 1973. 28, 32

[127] W Timothy J White and Michael D Hendy. Compressing DNA sequence databases with coil. *BMC Bioinformatics*, 9(1) :242, May 2008. 22, 93

[128] Ryan R Wick and Kathryn E Holt. Benchmarking of long-read assemblers for prokaryote whole genome sequencing. *F1000Res*, 8 :2138, December 2019. 19

[129] Derrick E. Wood and Steven L. Salzberg. Kraken : ultrafast metagenomic sequence classification using exact alignments. *Genome Biology*, 15(3) :R46, Mar 2014. 21

[130] Vladimir Yanovsky. ReCoil - an algorithm for compression of extremely large datasets of dna data. *Algorithms for Molecular Biology*, 6(1) :23, October 2011. 22, 93

[131] Ye Yu, Djamal Belazzougui, Chen Qian, and Qin Zhang. Memory-efficient and ultra-fast network lookup and forwarding using othello hashing. *IEEE/ACM Transactions on Networking*, PP :1–14, 04 2018. 45

[132] Ye Yu, Jinpeng Liu, Xinan Liu, Yi Zhang, Eamonn Magner, Erik Lehnert, Chen Qian, and Jinze Liu. SeqOthello : querying RNA-seq experiments at scale. *Genome Biology*, 19(1) :167, October 2018. 45

[133] Mohsen Zakeri, Nathaniel K. Brown, Omar Y. Ahmed, Travis Gagie, and Ben Langmead. Movi : A fast and cache-efficient full-text pangenome index. *iScience*, 27(12), Dec 2024. 35, 72

[134] Yongpeng Zhang, Linsen Li, Jun Xiao, Yanli Yang, and Zexuan Zhu. Fqzip : Lossless reference-based compression of next generation sequencing data in fastq format. In Hisashi Handa, Hisao Ishibuchi, Yew-Soon Ong, and Kay-Chen Tan, editors, *Proceedings of the 18th Asia Pacific Symposium on Intelligent and Evolutionary Systems - Volume 2*, pages 127–135, Cham, 2015. Springer International Publishing. 93

[135] Jianshu Zhao, Xiaofei Zhao, Both Jean Pierre, and Konstantinos Konstantinidis. Bindash 2.0 : New minhash scheme allows ultra-fast and accurate genome search and comparisons, 03 2024. 103

[136] XiaoFei Zhao. Bindash, software for fast genome distance estimation on a typical personal laptop. *Bioinformatics*, 35(4) :671–673, 07 2018. 21, 99