Mémoire de stage

Méthodes de *mapping* de *reads* avec indexation des *reads*

Master IGIS spécialité ITA, 2ème année

18 janvier 2017

Pierre Morisse

Encadrants : M. Thierry Lecroq et M. Arnaud Lefebvre

Table des matières

Introduction

Cadre

Ce mémoire a été rédigé dans le cadre d'un stage visant à conclure ma seconde année de Master, ainsi qu'à constituer une amorce à mon sujet de thèse. Ce stage s'est déroulé du 4 avril au 1^{er} juillet 2016, au sein de l'équipe TIBS du laboratoire LITIS, et plus précisément dans le bâtiment Théodore Monod, situé à Mont-Saint-Aignan. Les travaux de cette équipe portent principalement sur la recherche, l'indexation et l'extraction d'informations pertinentes dans des données biologiques et des systèmes d'information en santé, et offrent donc de nombreuses applications dans ces domaines. Ce stage a été financé par le LITIS, grâce à des fonds alloués par l'Université de Rouen, destinés à permettre l'accueil de stagiaires de seconde année de Master Recherche.

Contexte

Depuis le milieu des années 2000 et le développement des séquenceurs à très haut débit (*Next Generation Sequencing*), la biologie doit faire face au traitement d'énormes quantités de données, formées par des millions de très courtes séquences appelées *reads*. Ces *reads* sont notamment utilisés pour résoudre des problèmes de *mapping* ou d'assemblage, et doivent souvent subir une procédure de correction avant utilisation, afin de diminuer le taux d'erreurs de séquençage qu'ils présentent. La nécessité d'indexer les *reads* afin de résoudre ces problèmes a été identifiée dans [1], où les 7 requêtes suivantes, donnant lieu à de nombreuses potentielles applications, et un index les supportant ont également été introduits, pour une séquence f de longueur k donnée :

1. Dans quels *reads* f apparaît ?
2. Dans combien de *reads* f apparaît ?
3. Quelles sont les occurrences de f ?
4. Quel est le nombre d'occurrences de f ?

5. Dans quels *reads* f n'apparaît qu'une fois ?
6. Dans combien de *reads* f n'apparaît qu'une fois ?
7. Quelles sont les occurrences de f dans les *reads* où f n'apparaît qu'une fois ?

De nombreuses méthodes permettant l'indexation des *reads*, que cela soit à l'aide de structures de données classiques, de tables de hachage, ou de structures de données plus évoluées, existent et sont utilisées pour résoudre les différents problèmes de *mapping* et de correction de *reads*, ainsi que pour traiter les 7 requêtes précédemment décrites. Ces différentes méthodes sont décrites dans l'état de l'art, en Section 3.

Chapitre 1

Définitions et notations

Afin de rendre claire la lecture du reste de ce document, nous introduisons ici quelques définitions et notations, qui seront valables tout au long de celui-ci.

L'alphabet utilisé est l'alphabet ADN, $\Sigma = \{A, C, G, T\}$.

Assemblage :	Problème consistant à aligner et à fusionner des *reads* entre eux afin de reconstituer la longue séquence dont ils sont originalement issus
Contig :	Séquence générée par l'assemblage de plus courtes séquences se chevauchant
Gb :	Gigabases
Indels. :	Insertions et suppressions
k-mer :	Facteur de longueur k d'une séquence
Mapping :	Problème consistant à aligner des *reads* sur une séquence de référence
Polymorphisme :	Concept désignant la coexistence de plusieurs allèles, non pathogènes, pour un même gène
Read :	Séquence produite par un séquenceur à très haut débit. Un bref descriptif de ces séquenceurs est donné en Section 2
Séquence :	Mot sur l'alphabet Σ^*
Subs. :	Substitutions

Chapitre 2

Séquenceurs à très haut débit

Différentes technologies et plateformes sont disponibles pour le séquençage de *reads*. Elles se différencient principalement par la longueur des *reads* qu'elles produisent, par leur débit, ainsi que par le taux et le type d'erreurs qu'elles introduisent le plus fréquemment. Nous dressons, dans cette section, un bref descriptif de ces technologies et plateformes.

2.1 Séquençage de *reads* courts

La technologie Illumina repose sur le séquençage par synthèse basé sur des ADN polymérases. Elle propose diverses plateformes, produisant des *reads* de longueur différentes, avec un débit et un coût différents, en fonction des projets à réaliser. Par exemple, elle permet de produire 3 milliards de *reads* de longueur comprise entre 36 et 100, avec une précision supérieure à 99%, en 2 à 11 jours, soit un débit d'environ 600 Gb par *run*, avec sa plateforme HiSeq 2500/1500, pour un coût de 740 000$. Sa plateforme MiSeq, d'un coût de 125 000$, quant à elle, permet de produire 17 millions de *reads* de longueur comprise en 25 et 250, avec une précision supérieure à 99%, en 4 à 27 heures, soit un débit d'environ 8,5 Gb par *run*. Quelle que soit la plateforme utilisée, les erreurs les plus fréquentes dans les *reads* séquencés par cette technologie sont des erreurs de substitutions, et elle sont, le plus souvent, situées à la fin des *reads*.

La technologie Roche repose sur le pyroséquençage, et propose, elle aussi, différentes plateformes. Par exemple, elle permet de produire un million de *reads* de longueur 700, avec une précision de 99,997%, en 23 heures, soit un débit de 0,7 Gb par *run*, avec sa plateforme 454 GS FLX+, d'un coût de 450 000$. Sa plateforme 454 GS Junior, d'un coût de 108 000$, quant à elle, permet de produire un million de *reads* de longueur 400, avec une précision supérieure à 99%, en 10 heures, soit un débit de 0,4 Gb par *run*. Quelle que

soit la plateforme utilisée, les erreurs les plus fréquentes dans les *reads* séquencés par cette technologie sont des erreurs d'insertions et de suppressions.

La technologie ABI Life Technologies propose différentes méthodes de séquençage des *reads*. Par exemple, sa plateforme 5500xl SOLiD, d'un coût de 595 000$, repose sur le séquençage par ligature. Elle permet ainsi de produire 2,8 millions de *reads* de longueur 75, avec une précision de 99,99%, en 7 jours, soit un débit de 180 Gb par *run*. Sa plateforme Ion Proton Chip I/II, d'un coût de 243 000$, quant à elle, repose sur le séquençage par détection de protons. Elle permet ainsi de produire 60 à 80 millions de *reads* de longueur jusqu'à 200, avec une précision supérieure à 99%, en 2 heures, soit un débit d'environ 10 à 100 Gb par *run*. Quelle que soit la plateforme utilisée, les erreurs les plus fréquentes dans les *reads* séquencés par cette technologie sont des erreurs d'insertions et de suppressions.

2.2 Séquençage de *reads* longs

Depuis peu, de nouvelles technologies se développent et permettent de séquencer des *reads* de plus en plus longs, ouvrant ainsi les portes à de nombreuses potentielles applications jusqu'ici impossibles, mais engendrent cependant un taux d'erreurs de séquençage bien plus important.

La technologie Pacific Bioscience repose sur le séquençage de simple molécule en temps réel. Sa plateforme PacBio RS, d'un coût de 750 000$, permet de produire environ 50 000 *reads* de longueur moyenne 3 000, avec une faible précision de seulement 85%, en 2 heures, soit un débit d'environ 0,135 Gb par *run*. Les erreurs les plus fréquentes dans les *reads* séquencés par cette technologie sont des erreurs d'insertions et de suppressions.

La technologie Oxford Nanopore reposait sur le séquençage par exonucléase par nanopore lors de l'étude des différents documents, mais une nouvelle version a cependant été développée récemment, et le séquençage repose désormais sur une translocation par nanopore. Cette technologie permet, par exemple, avec sa plateforme GridION, de produire 4 à 10 millions de *reads* de quelques dizaines de milliers de bases de longueur, avec une précision atteignant les 96%, en un temps variable en fonction de l'expérience, mais offrant cependant un débit de quelques dizaines de Gb par *run*, pour un coût également variable en fonction de l'expérience. Sa plateforme MinION, d'un coût inférieur à 1 000$, et pouvant être utilisée en la connectant simplement via USB à un ordinateur classique, quant à elle, permet de produire en moyenne 70 000 *reads* de quelques dizaines de milliers de bases de longueur, avec une très faible précision, de seulement environ 70% en 48

heures, soit un débit d'environ 0,132 Gb par *run*. Les erreurs les plus fréquentes dans les *reads* séquencés par cette technologie sont des erreurs d'insertions et de suppressions, et ce quelle que soit la plateforme utilisée. De plus, cette technologie permet de séquencer deux différents types de *reads*. En effet, lorsque les deux brins de la molécule sont correctement lus, un consensus est construit pour obtenir un *read* plus long et plus précis, appelé *read* 2D. Dans le cas contraire, lorsque seul un brin est correctement lu, le *read* séquencé est alors plus court, moins précis, et est appelé *read* 1D.

2.3 Tableau récapitulatif

Un récapitulatif des différentes technologies de séquençage, de leurs différentes plateformes, de leurs coûts, et de leurs propriétés est donné Table 2.1.

On remarque, sur ce tableau, que de nombreuses plateformes de séquençage sont disponibles, et que chacune d'elle offre des propriétés différentes, permettant ainsi de traiter différents types de problèmes de génomique. On remarque également que, au prix du débit, le coût de ces séquenceurs a considérablement diminué, l'exemple le plus flagrant étant la plateforme MinION, proposant un séquençage accessible à tous, puisque pouvant fonctionner par simple connexion à un ordinateur classique, le tout pour un prix inférieur à 1 000$.

Technologie	Technique de séquençage	Plateforme	Nombre de *reads*	Longueur	Précision	Temps	Débit	Coût	Erreurs
Illumina	Synthèse, basé sur ADN polymérases	HiSeq 2500/1500 MiSeq	3 milliards 17 millions	36 - 100 25 - 250	99 >99	2 - 11 jours 4 - 27 heures	600 8,5	740 000 125 000	Subs.
Roche	Pyroséquençage	454 GS FLX+ 454 GS Junior	1 million 1 million	700 400	99,997 >99	23 heures 10 heures	0,7 0,4	450 000 108 000	Indels.
ABI Life Technologies	Ligature Détection de protons	5500xl SOLiD Ion Proton Chip I/II	2,8 millions 60 - 80 millions	75 jusqu'à 200	99,99 >99	7 jours 2 heures	180 10 - 100	595 000 243 000	Indels.
Pacific Bioscience	Simple molécule en temps réel	PacBio RS	50 000	3 000 en moyenne	85	2 heures	13	750 000	Indels.
Oxford Nanopore	Exonucléase par Nanopore	GridION MinION	4 - 10 millions 70 000	> 10 000 > 10 000	96 70	variable 48 heures	quelques dizaines 0,132	variable 1 000	Indels.

TABLE 2.1 – Récapitulatif des différentes technologies de séquençage. La précision est donnée en %, le débit en Gb (Gigabases) par *run*, et le coût en $.

Chapitre 3

État de l'art

Au vu des nombreuses différences entre les diverses technologies et plateformes de séquençage, de multiples méthodes permettant d'indexer un ensemble de *reads* ont été développées, aussi bien pour permettre le traitement des 7 requêtes, que pour résoudre des problèmes de *mapping* ou de correction, la diversité de ces structures prenant auquel cas tout son sens, les erreurs de séquençage n'étant pas les mêmes en fonction de la technologie utilisée.

3.1 Structures de données classiques

Il est possible d'utiliser des structures de données classiques généralisées afin d'indexer un ensemble de *reads*. Nous décrivons ci-dessous diverses méthodes adoptant cette approche.

3.1.1 SHREC - HybridSHREC

SHREC (*SHort-Read Error Correction method*), développé en 2009 dans [2], et utilisé pour la correction de *reads* courts, repose sur un arbre des suffixes [3] généralisé, construit pour le texte obtenu par concaténation de tous les *reads* de l'ensemble, séparés par des caractères spéciaux n'appartenant pas à l'alphabet, et permet d'indexer un ensemble de 1 090 946 *reads* de longueur 70 sur une machine classique, en utilisant 1,5 Go de mémoire. Les ressources demandées par une telle structure sont cependant tellement importantes, et augmentent tellement rapidement en fonction du nombre et de la longueur des *reads* qu'il est impossible d'indexer un ensemble de *reads* provenant d'un génome humain, même sur une machine disposant de très importantes ressources.

SHREC ne permet de traiter que des erreurs de substitutions. Pour corriger les *reads*, il réalise un parcours en profondeur de l'arbre des suffixes et, pour chaque nœud v, vérifie si l'un de ses fils w a un poids, calculé en fonction du nombre de fils du nœud, plus faible qu'un seuil fixé. Si c'est le cas, il tente alors d'identifier w à l'un de ses frères, afin de les fusionner, et de corriger les *reads* erronés décrits par w.

Ainsi, SHREC permet de corriger, en moyenne, 88,56% des *reads* erronés, et traite la correction de 1 090 946 *reads* de longueur 70 en 108 à 264 minutes, en fonction du taux d'erreurs présenté par les *reads*.

Une version améliorée de SHREC, nommée HybridSHREC, a été développée en 2010 dans [4], et permet de corriger également les erreurs liées à des insertions ou à des suppressions, en suivant la même méthode que SHREC. De par la prise en compte des insertions et des suppressions, les résultats offerts sont bien meilleurs, et HybridSHREC permet ainsi de corriger, en moyenne, 98,39% des *reads* erronés. De plus, HybridSHREC se montre également plus rapide que SHREC, et traite la correction de 1 120 000 *reads* de longueur comprise entre 75 et 125, et disposant d'un taux d'erreurs de 3%, en 40 minutes.

3.1.2 HiTEC

HiTEC (*High Throughput Error Correction*), développé en 2011 dans [5], également utilisé pour la correction de *reads* courts, repose sur une table des suffixes [6] permettant de simuler l'arbre des suffixes généralisé, tout en réduisant la consommation en mémoire. Il est ainsi possible d'indexer un ensemble de 1 090 946 *reads* de longueur 70 sur une machine classique, en utilisant cette fois seulement 754 Mo de mémoire. Cette réduction de consommation en mémoire s'atténue cependant à mesure que le nombre et que la longueur des *reads* à indexer augmente, et la solution présentée ici ne permet donc toujours pas d'indexer un ensemble de *reads* provenant d'un génome humain, même avec d'importantes ressources.

HiTEC ne permet de traiter que des erreurs de substitutions. Pour corriger les *reads*, il se sert de séquences témoins, qui sont des séquences d'une longueur fixée, ne contenant pas de caractère spécial. De cette façon, si par exemple, dans l'ensemble de *reads*, la séquence témoin u est le plus souvent suivie d'un A, et que, dans un certain *read* de l'ensemble, la séquence u est suivie d'un B, ce B sera considéré comme une erreur, et le *read* pourra être corrigé, en remplaçant le B par un A.

Ainsi, HiTEC permet de corriger, en moyenne, 94,43% des *reads* erronés, et traite la correction de 1 090 946 *reads* de longueur 70 en 18 à 40 minutes, en fonction du taux d'erreurs présenté par les *reads*.

3.1.3 Fiona

Fiona, développé en 2014 dans [7], lui aussi utilisé pour la correction de *reads* courts, utilise une table des suffixes partielle pour réduire encore davantage l'espace nécessaire à l'indexation. Il est alors possible d'indexer des ensembles de *reads* plus importants, provenant de génomes plus conséquents. Par exemple, il est possible d'indexer intégralement un ensemble de 186 millions de *reads* de longueur 177 provenant d'un génome humain à l'aide de cette structure, mais bien que la complexité en espace des solutions précédentes soit nettement réduite, une telle indexation demande tout de même encore 244 Go de mémoire.

Fiona permet de traiter des erreurs de substitutions, d'insertions, et de suppressions. Pour cela, il parcourt l'arbre simulé par la table des suffixes, et lorsqu'un nœud v a un fils w ayant un poids plus faible qu'un seuil fixé, ce dernier est considéré comme erroné. Chaque nœud du sous arbre de racine w est alors comparé aux nœuds des autres fils, corrects, de v, et un vote majoritaire est appliqué, afin de déterminer quelle correction appliquer à w et à ces fils, et donc aux *reads* décrits par ces nœuds.

Ainsi, Fiona permet de corriger, en moyenne, 66,76% des *reads* erronés, et traite la correction d'un million de *reads* de longueur 178 en environ 15 minutes.

3.2 Tables de hachage

Il est également possible d'utiliser des tables de hachage afin d'indexer un ensemble de *reads*. Nous décrivons ci-dessous diverses méthodes adoptant cette approche.

3.2.1 MAQ

MAQ (*Mapping and Assembly with Quality*), développé en 2008 dans [8] permet de résoudre des problèmes de *mapping* de *reads* courts. Pour cela, il utilise des *templates*, qui sont des mots sur l'alphabet binaire, de la même longueur que les *reads*. Les *reads* sont comparés aux *templates*, et les nucléotides ayant une position à 1 dans ces *templates* sont hachés en un entier de 24 bits, qui est ajouté à une liste, avec l'identifiant du *read*. La liste est ensuite triée selon les valeurs de hachage, afin de grouper en mémoire les *reads* partageant cette valeur, puis chaque entier, ainsi que l'ensemble de *reads* correspondant, sont stockés dans une table de hachage, avec l'entier pour clé. Une table de hachage différente est utilisée pour chaque *template*, et un *template* possède toujours un complémentaire, qui est traité en parallèle lors de l'indexation. Par défaut, MAQ utilise 6 *templates*, et donc 6 tables de hachage. Une fois le processus d'indexation pour une paire de *templates* terminé, celui-ci est alors répété avec la paire suivante, jusqu'à ce que tous les *templates* aient été

traités. Il est ainsi possible d'indexer un ensemble d'un million de *reads* de longueur 44, en utilisant 1,2 Go de mémoire.

Le génome de référence est ensuite scanné en avant et en arrière, et chaque facteur de la longueur des *reads* est recherché, après conversion en un entier de 24 bits, fonction des différents *templates*, dans les tables de hachage correspondantes, afin de trouver les potentiels alignements, cette phase de conversion et de recherche étant effectuée pour l'ensemble de tous les *templates*.

MAQ prend en compte les erreurs de substitutions, d'insertions et de suppressions lors du *mapping*, et permet, par exemple, de traiter un ensemble d'un million de *reads* de longueur 35, en mappant effectivement 93,2% de ces *reads*, en 331 minutes.

3.2.2 MrsFAST - MrsFAST-Ultra

MrsFAST (*Micro-Read (Substitutions only) Fast Alignment and Search Tool*), développé en 2010 dans [9], permet de résoudre des problèmes de *mapping* de *reads* courts à l'aide d'une table de hachage. Pour cela, il indexe l'ensemble des k-mers des *reads*, ainsi que du génome de référence sur lequel ceux-ci doivent être *mappés*, dans une table de hachage, afin de permettre un traitement rapide. L'indexation d'un ensemble de *reads* provenant d'un génome humain, et du génome de référence, par cette méthode, demande alors 20 Go de mémoire et est effectuée en 26 minutes. Une version améliorée de cette structure, nommée MrsFAST-Ultra, a été développée en 2014 dans [10], et permet l'indexation d'un ensemble de *reads* provenant d'un génome humain, et du génome de référence, en seulement 8 minutes, en n'utilisant cette fois plus que 2 Go de mémoire. De plus, cette version peut également prendre en compte le polymorphisme lors du *mapping* des *reads* sur le génome de référence, afin de produire de meilleurs résultats. Comme leurs noms l'indiquent, ces deux outils ne prennent pas en compte les erreurs d'insertions et de suppressions lors du *mapping*.

Ainsi, MrsFAST permet de mapper 87,27% des *reads* d'un ensemble d'un million de *reads* de longueur 100 sur un génome de référence, en tolérant un maximum de 6 erreurs de substitutions, en 169 minutes. MrsFAST-Ultra, de son côté, permet de mapper jusqu'à 90,55% des *reads* d'un ensemble de 2 millions de *reads* de longueur 100 sur un génome de référence, avec le même nombre maximum d'erreurs de substitutions, en seulement 57 minutes. De plus, MrsFAST-Ultra permet de reporter jusqu'à 100 occurrences de chaque *read* de cet ensemble dans le génome de référence en 58 minutes.

3.2.3 Coral

Coral (*CORrection with ALignments*), développé en 2011 dans [11], utilise l'alignement multiple de *reads* courts afin de résoudre des problèmes de correction de *reads* courts. Pour ce faire, il indexe l'ensemble des *k*-mers de tous les *reads* à l'aide d'une table de hachage, associant à chaque *k*-mer la liste des *reads* dans lesquels il apparait. Ainsi, il permet l'indexation d'un ensemble de 3,4 millions de *reads* de longueur 47 en utilisant 2,6 Go de mémoire.

Coral permet de corriger des erreurs de substitutions, d'insertions, et de suppressions. Pour cela, un *read* dit de base est choisi, et tous les *reads* partageant un *k*-mer commun avec celui-ci sont alignés dessus. Une séquence consensus est ensuite déduite de l'alignement des tous les *reads*, et ceux ayant des positions ne s'accordant pas avec la séquence consensus sont corrigés en fonction de celle-ci. L'opération est ensuite répétée, jusqu'à ce que chaque *read* ait été choisi comme *read* de base.

Ainsi, Coral permet de corriger, en moyenne, 92,88% des *reads* erronés, et traite la correction d'un ensemble d'un million de *reads* de longueur 178 en environ 5 minutes.

3.2.4 RACER

RACER (*Rapid and Accurate Correction of Errors in Reads*), développé en 2013 dans [12] permet également de résoudre des problèmes de correction de *reads* courts. Pour cela, il stocke l'ensemble des *k*-mers des *reads*, chacun représenté comme un entier de 64 bits, dans une table de hachage, et associe à chacun de ces *k*-mers un ensemble de 8 compteurs. Ainsi, il se montre compétitif aux structures disponibles à l'époque, en termes de mémoire, et permet l'indexation d'un ensemble de 2 119 404 *reads* de longueur 47, en utilisant 1,437 Go de mémoire.

RACER ne permet de traiter que des erreurs de substitutions, bien que la prise en compte des insertions et suppressions soit prévue pour de futures versions. Pour corriger les *reads*, il compte dénombre les différents *k*-mers de l'ensemble de *reads*, et ceux au dessous d'un certain seuil sont corrigés, par une méthode non détaillée, ni dans l'article, ni dans le matériel supplémentaire.

Ainsi, RACER permet de corriger, en moyenne, 76,65% des *reads* erronés, et traite la correction de 2 119 404 *reads* de longueur 75 en environ 23 minutes. RACER a également été testé un ensemble de 101 548 652 *reads* Illumina de longueur 457 595 provenant d'un génome de mouche, et a pu corriger 42,95% des *reads* erronées de cet ensemble, en 104 minutes en exécution multithreads, le tout en utilisant environ 41,7 Go de mémoire.

3.2.5 BLESS

BLESS (*BLoom-filter-based Error correction Solution for high-throughput Sequencing reads*), développé en 2014 dans [13], permet lui aussi de résoudre des problèmes de correction de *reads* courts. Pour cela, il se sert de différents fichiers, dans lesquels il stocke les *k*-mers de tous les *reads* en fonction d'une valeur de hachage. Les différents *k*-mers sont ensuite comptés à l'aide d'une table de hachage, et ceux apparaissant plus souvent qu'un seul fixé sont considérés comme corrects, et sont ajoutés à un filtre de Bloom [14], consommant peu de mémoire, et permettant un test d'appartenance rapide. Ainsi, BLESS se montre extrêmement efficace concernant la complexité en espace, et ne consomme en moyenne que 2,5% de la mémoire utilisée par les autres méthodes. De ce fait, il est par exemple possible d'indexer un ensemble de 20 millions de *reads* de longueur 36 en utilisant seulement 14 Mo de mémoire.

BLESS permet de traiter des erreurs de substitutions, d'insertions, et de suppressions. Pour cela, il compare les *k*-mers erronés, apparaissant moins souvent qu'un seuil fixé, à un ensemble de *k*-mers corrects voisins, à l'aide du filtre de Bloom, et corrige les *k*-mers erronés en fonction de ces derniers. Les *k*-mers erronés produits à cause des faux positifs du filtre de Bloom sont, quant à eux, restaurés à leur valeur initiale en fin de traitement, en les comptant, et en comparant le nombre obtenu au seuil précédemment fixé.

Ainsi, BLESS permet de corriger, en moyenne, 84,38% des *reads* erronés, et traite la correction d'un million de *reads* de longueur 101, et disposant d'un taux d'erreurs de 2,1%, en environ 6 minutes.

3.3 Structures de données évoluées

Au vu de l'incapacité des solutions présentées précédemment à traiter les *reads* longs et l'ensemble des 7 requêtes, le besoin de trouver des structures de données et des méthodes plus adaptées, et donc plus efficaces, s'est vite révélé urgent. Des solutions plus évoluées ont alors été développées, et permettent de traiter ces *reads* longs, ou de répondre à la liste de requêtes, tout en indexant les *reads* aussi ou plus efficacement que les méthodes précédentes. Nous décrivons ces diverses structures et méthodes ci-dessous.

3.3.1 GkA

La première de ces structures a été développée en 2011 dans [1], et se nomme GkA (*Gk Arrays*). Elle repose sur le tri selon l'ordre lexicographique de tous les *k*-mers des *reads*, et sur trois tableaux. Le premier est une table des suffixes modifiée, et permet de stocker la position de début de chaque *k*-mer, le second est un tableau inverse, donnant le rang

lexicographique d'un k-mer à partir de sa position de début dans un *read*, et le dernier est un tableau associant un k-mer à son nombre d'occurrences. Cette structure s'est montrée plus efficace en termes de mémoire et plus rapide que l'indexation par tables de hachage ou par tables des suffixes, comparée aux méthodes disponibles à l'époque. Elle permet l'indexation de 42,4 millions de *reads* de longueur 75 provenant d'un génome humain en utilisant 20 Go de mémoire, et le traitement de la liste des 7 requêtes, pour une longueur k de f fixée.

3.3.2 CGkA

Une version compressée de la structure précédente, nommée CGkA (*Compressed Gk Arrays*), a été développée en 2013 dans [15]. Elle repose sur une table des suffixes échantillonnée, construite pour le texte obtenu par la concaténation de tous les *reads* de l'ensemble de départ, séparés par des caractères spéciaux n'appartenant pas à l'alphabet, et sur trois vecteurs de bits. Elle permet ainsi de réduire de 40 à 90% la taille des GkA classiques, tout en répondant à l'ensemble des requêtes, avec une complexité en temps cependant plus importante. À l'aide de cette structure, il est donc possible d'indexer 42,4 millions de *reads* de longueur 75 provenant d'un génome humain, en n'utilisant plus qu'entre 3 et 7 Go de mémoire, en fonction du degré d'échantillonnage choisi, et de répondre aux 7 requêtes, pour différentes longueurs k de f.

3.3.3 PgSA

Une autre structure, nommée PgSA (*Pseudogenomic Suffix Array*), a été développée en 2015 dans [16], et repose également sur une table des suffixes échantillonnée, cette fois construite pour le pseudogénome obtenu à partir des *reads* de l'ensemble de départ, concaténés et imbriqués en prenant en compte les chevauchements, afin de réduire la longueur de la séquence à indexer, et donc la taille de la table. En complément de la table des suffixes échantillonnée, PgSA repose également sur une table auxiliaire, apportant des informations sur les indices et les *offsets* des *reads* dans le pseudogénome, et sur les occurrences des k-mers. Cette structure se montre compétitive aux deux précédentes en termes de mémoire. En effet, elle permet d'indexer 42,4 millions de *reads* de longueur 75 provenant d'un génome humain, en utilisant seulement entre 1 et 4 Go de mémoire, en fonction du degré d'échantillonnage choisi, tout en supportant le traitement des 7 requêtes, pour différentes longueurs k de f.

Avec cette structure, la complexité en temps du traitement des requêtes 1 et 3 s'est montrée nettement plus faible que celle des CGkA de [15], et celle des requêtes 2 et 4, légèrement plus importante. Comparée aux GkA de [1], cette complexité s'est montrée

légèrement plus importante, et ce pour l'ensemble des requêtes 1 à 4. Cependant, la complexité en temps des requêtes 5 à 7 n'a pas pu être comparée, celles-ci n'ayant pas été implémentées dans les versions de GkA et de CGkA utilisées lors de la rédaction de l'article introduisant PgSA.

3.3.4 LoRDEC

LoRDEC (*Long Read DBG Error Correction*), développé en 2014 dans [17], est un outil utilisé pour résoudre des problèmes de correction de *reads* longs. Principalement conçu pour une utilisation sur des *reads* de la technologie Pacific Bioscience, disposant donc d'un taux d'erreurs d'environ 15%, il permet la correction à l'aide d'un ensemble de *reads* courts, disposant d'un plus faible taux d'erreurs, pour lequel est construit un graphe de de Bruijn [18]. Prenant en compte les récents développements pour la représentation compacte des graphes de de Bruijn, il est ainsi possible d'indexer un ensemble de 2 316 613 *reads* courts de longueur 100 sous forme de graphe, en utilisant 960 Mo de mémoire.

LoRDEC permet de traiter des erreurs de substitutions, d'insertions, et de suppressions. Pour cela, il construit le graphe de de Bruijn des k-mers corrects, apparaissant plus souvent qu'un seuil fixé, des *reads* courts. Il corrige alors les régions erronées des *reads* longs en cherchant un chemin optimal dans le graphe de de Bruijn, partant d'un k-mer source, situé à gauche d'une région erronée, vers un k-mer cible, situé à droite d'une région erronée. La séquence des k-mers se chevauchant le long du chemin fournit alors une correction. Si la région erronée se situe à une extrémité du *read* long, LoRDEC tente de construire une extension, allant de la région erronée jusqu'à la fin (respectivement jusqu'au début) du *read*, et aligne la séquence associée au chemin construit sur la région erronée, par programmation dynamique, pour déduire le suffixe (respectivement le préfixe) optimal à utiliser en tant que correction.

Ainsi, LoRDEC permet de corriger, en moyenne, 85,783% des *reads* erronés, et traite la correction de 33 360 *reads* de longueur moyenne 2 938, et de longueur maximale 14 949, en environ 10 minutes.

3.4 Tableaux récapitulatifs

Nous dressons ici différents tableaux récapitulatifs, permettant de comparer les performances des différentes méthodes et structures de données présentées précédemment, en regroupant ces dernières en fonction du type de problème qu'elles permettent de résoudre.

3.4.1 Correction de *reads*

Un récapitulatif des différentes méthodes de correction de *reads*, des structures de données sur lesquelles elles reposent, et de leur efficacité est donné Table 3.1.

Outil	Structure de données	Erreurs corrigées	Nombre de *reads* (longueur)	Espace mémoire	Temps	*Reads* corrigés
SHREC	Arbre des suffixes	subs.	1 090 946 (70)	1 500	183	88,56
HybridSHREC	Arbre des suffixes	subs. + indels	977 971 (178)	15 000	28	98,39
HiTEC	Table des suffixes	subs.	1 090 946 (70) 4 639 675 (70)	757 3 210	28 125	94,43
Fiona	Table des suffixes partielle	subs. + indels	977 971 (178) 2 464 690 (142)	2 000 3 000	15 32	66,76
Coral	Table de hachage	subs. + indels	977 971 (178)	8 000	5	92,88
RACER	Table de hachage	subs.	2 119 404 (75) 101 548 652 (457 595)	1 437 41 700	23 104	76,65 42,95
BLESS	Filtres de Bloom	subs. + indels	1 096 140 (101)	11	6	84,38
LoRDEC	Graphe de de Bruijn	subs. + indels	33 360 *reads* longs (2 938) et 2 316 613 *reads* courts (100)	960	10	85,78

TABLE 3.1 – Récapitulatif des différentes méthodes de correction des *reads*. L'espace mémoire est donnée en Mo. Le temps est donné en minutes. Les *reads* corrigés sont donnés en %, et la valeur indiquée est une moyenne obtenue à partir de différents ensembles de données.

On remarque que l'espace mémoire occupé par SHREC / HybridSHREC et HiTEC augmente très rapidement en fonction du nombre et de la longueur des *reads* à indexer, bien que celui occupé par HiTEC reste relativement raisonnable, mais que ces outils produisent de bons résultats. HiTEC se montre également plus rapide que les deux méthodes susmentionnées, bien que son temps d'exécution augmente rapidement en fonction de la taille des données. Fiona occupe un espace mémoire plus raisonnable, n'augmentant pas trop rapidement en fonction du nombre et de la longueur des *reads*, et se montre plus rapide que les solutions précédentes, mais produit des résultats très peu satisfaisants. Coral, au contraire, se montre assez gourmand en mémoire, mais produit de bons résultats, en un temps encore plus faible que Fiona. RACER se montre légèrement plus efficace que Fiona, aussi bien en termes d'efficacité de la correction que d'espace mémoire occupé, au prix d'une complexité en temps légèrement plus importante, mais produit toujours des résultats assez peu satisfaisants. BLESS s'impose aisément comme l'outil le plus efficace en terme d'espace mémoire occupé, se montre également compétitif aux autres solutions en termes de temps d'exécution, et produit des résultats acceptables. LoRDEC, quant à lui, est le seul outil permettant de corriger des *reads* longs en utilisant une structure d'index sur les *reads* étudié, et ne peut donc pas être comparé à d'autres solutions.

3.4.2 *Mapping* de *reads*

Un récapitulatif des différentes méthodes de *mapping* de *reads*, et de leur efficacité est donné Table 3.2.

Outil	Structure de données	Erreurs prises en compte	Nombre de *reads* (longueur)	Espace mémoire	Temps	*Reads mappés*
MAQ	Table de hachage	subs. + indels	1 000 000 (44)	1 200	331	92,53
MrsFAST	Table de hachage	subs.	1 000 000 (100)	20 000	169	90,70
MrsFAST-Ultra	Table de hachage	subs.	2 000 000 (100)	2 000	57	91,41

TABLE 3.2 – Récapitulatif des différentes méthodes de *mapping* de *reads*. L'espace mémoire est donnée en Mo. Le temps est donné en minutes. Les *reads mappés* sont donnés en %, et la valeur indiquée est une moyenne obtenue à partir de différents ensembles de données.

On remarque que MAQ demeure l'outil le plus efficace en termes de *reads* effectivement *mappés*, mais que son temps d'exécution est bien trop important pour être utilisé en pratique. MrsFAST réduit nettement le temps d'exécution, permettant de traiter des *reads* plus de 2 fois plus longs en presque 2 fois moins de temps, mais souffre d'une consommation de mémoire bien trop importante, presque 16 fois plus grande que celle de MAQ. MrsFAST-Ultra, quant à lui, réduit encore davantage le temps d'exécution, mappant 2 fois plus de *reads* que son prédécesseur, en presque 3 fois moins de temps, et produit de très bons résultats, mappant en moyenne seulement 1% de *reads* de moins que MAQ, en utilisant un espace mémoire seulement légèrement supérieur à ce dernier. MrsFAST-Ultra semble donc s'imposer comme l'outil le plus efficace parmi les trois présentés.

Peu d'outils sont présentés ici, mais de nombreuses méthodes de *mapping* de *reads*, indexant le génome de référence plutôt que les *reads*, existent cependant et produisent de bons résultats, aussi bien en espace et en temps, qu'en qualité de *mapping*. Nous présentons brièvement quelques uns de ces outils ci-dessous.

SOAP, développé en 2008 dans [19], indexe le génome de référence à l'aide d'une table de hachage, utilisant par exemple 14,7 Go de mémoire pour l'indexation d'un génome humain, et permet de mapper 93,6% des *reads* d'un ensemble d'un million de *reads* de longueur 44 sur le génome de référence, en environ 280 minutes, et en prenant en compte les erreurs de substitutions, d'insertions, et de suppressions lors du *mapping*. Sa version améliorée, SOAP2, développée en 2009 dans [20], utilise la transformée de Burrows-Wheeler [21] pour indexer le génome de référence, et réduit alors l'espace nécessaire à l'indexation d'un génome humain à 5,4 Go, tout en conservant la même qualité de *mapping* et en pouvant traiter des *reads* de longueur jusqu'à 1024. De plus, le *mapping* de l'ensemble d'un million de *reads* de longueur 44 ne demande plus qu'environ 11 minutes, soit environ 25 fois moins de temps qu'avec SOAP.

Bowtie, développé en 2009 dans [22], utilise un FM-index [23], basé sur la transformée de Burrows-Wheeler, pour indexer le génome de référence, utilisant par exemple 1,3 Go de mémoire pour l'indexation d'un génome humain, et permet de traiter environ 29,5 millions de *reads* par heure, en mappant effectivement, en moyenne, 71,9% de ces *reads* sur le génome de référence, et en ne prenant en compte que les erreurs de substitutions lors du *mapping*. Sa version améliorée, Bowtie2, développée en 2012 dans [24], étend Bowtie afin de permettre la prise en compte des erreurs d'insertions et de suppressions lors du *mapping*, à l'aide de programmation dynamique. L'espace mémoire est alors affecté, et l'alignement de *reads* sur un génome humain provoque un pic de mémoire à 3,24 Go durant l'exécution. Cependant, Bowtie2 se montre bien plus efficace que Bowtie, en alignant, en moyenne, 95% des *reads* sur le génome de référence, tout en étant légèrement plus rapide, malgré l'utilisation de programmation dynamique en complément.

3.4.3 Traitement des 7 requêtes

Un récapitulatif des différentes méthodes permettant de répondre aux 7 requêtes, des structures de données sur lesquelles elles reposent, et de leur efficacité est donné Table 3.3.

Outil	Structure de données	Nombre de *reads* (longueur)	Espace mémoire	Temps R1	Temps R2	Temps R3	Temps R4
GkA	Table des suffixes modifiée + Table des suffixes modifiée inverse + Table associant *k*-mer - nombre d'occurrences	42 400 000 (75)	20	16	25	25	0,1
CGkA	Table de suffixes échantillonnée + 3 vecteurs de bits	42 400 000 (75)	3 - 7	1203	28	1278	28
PgSA	Table des suffixes échantillonnée + Table auxiliaire d'information sur les *reads* et *k*-mers	42 400 000 (75)	1 - 4	70	58	70	58

TABLE 3.3 – Récapitulatif des différentes méthodes permettant de traiter les 7 requêtes. L'espace mémoire est donné en Go. Le temps est donné en millisecondes. Les requêtes 5-7 sont exclues du comparatif, car non implémentées dans GkA et CGkA lors des tests réalisés dans [16].

On remarque que, concernant la complexité en temps, GkA se montre plus efficace que les deux autres solutions, mais demande cependant un espace mémoire beaucoup plus important. CGkA réduit nettement l'espace mémoire, en offrant une complexité en temps similaire pour la requête 2, mais beaucoup plus importante pour les requêtes 1, 3 et 4. PgSA offre, quant à lui, une complexité en temps similaire pour le traitement des 4 requêtes, bien qu'environ trois fois plus importante que celle de GkA, en compressant cependant de 5 à 20 fois la taille de l'index utilisé par ce dernier. Ainsi, PgSA semble donc offrir le meilleur compromis temps / mémoire pour le traitement des 7 requêtes.

Chapitre 4

Méthode alternative à la correction de *reads* longs : Les *reads* NaS

Les *reads* longs permettent de résoudre des problèmes d'assemblage longs et complexes, qu'il aurait été impossible de résoudre à l'aide de *reads* courts. De plus, comme nous l'avons vu dans la Section 2, séquencer de tels *reads* est devenu relativement rapide et peu coûteux, et des outils tels que MinION, disponible pour moins de 1 000$, permettent de séquencer facilement ces *reads* longs. Cependant, utiliser ces *reads*, en particulier dans le cas de problèmes d'assemblage, est difficile, car ils disposent d'un important taux d'erreurs, avoisinant notamment les 30% dans le cas des *reads* séquencés par MinION. De plus, la correction de ces *reads* par des solutions classiques n'est pas aussi efficace que la correction de *reads* courts, comme le montre la Table 3.1, et il s'est donc révélé urgent de proposer une méthode alternative permettant d'appliquer un traitement correctif aux *reads* longs avant leur utilisation.

Ainsi, une solution envisagée a été la création de *reads* longs dits synthétiques, générés à partir d'une approche hybride utilisant des *reads* longs, disposant donc de forts taux d'erreurs, comme *templates* et des *reads* courts, étant eux bien plus précis, sans pour autant tenter d'apporter une réelle correction aux *reads* longs. Ces nouveaux *reads* ainsi obtenus, appelés *reads* NaS (Nanopore *Synthetic-long*), car synthétisés à partir de *reads* longs de la technologie Nanopore, ont été introduits en 2015 dans [25]. Ils peuvent atteindre une longueur de 60 000, disposent d'une précision de 99,99%, et peuvent donc s'aligner intégralement et sans erreurs. Ils représentent ainsi la première solution efficace permettant d'apporter un traitement correctif aux *reads* longs, afin de pouvoir plus facilement utiliser ces derniers dans divers problèmes de *mapping* ou d'assemblage.

Nous présentons ici deux méthodes permettant d'obtenir de tels *reads*. La première

nécessite d'aligner les *reads* courts sur les *templates*, mais également entre eux, tandis que la deuxième tente de déduire le maximum d'informations à partir de l'alignement des *reads* courts sur les *templates* uniquement.

4.1 Jeu de données utilisé

Pour illustrer la pertinence de la synthèse de *reads* NaS, nous avons utilisé un jeu de données composé de cinq ensembles de *reads* longs de la bactérie *Acinetobacter baylyi ADP1*, séquencés par MinION. Ces cinq ensembles totalisent 66 492 *reads* dont 13% de *reads* 2D, représentant 42% de la taille totale des données, et indiquant donc une importante différence entre la longueur des *reads* 1D et des *reads* 2D. En effet, les *reads* 1D présentent une longueur moyenne de 2 052 tandis que les *reads* 2D affichent une longueur moyenne atteignant 10 033. Une description des cinq ensembles de *reads* longs utilisés, du pourcentage de *reads* 2D qu'ils contiennent et de la taille totale des données que ceux-ci représentent est donnée Table 4.1.

Ensemble	Nombre de *reads*	% *reads* 2D	% taille totale
1	9 241	6,5	14,6
2	3 990	13,6	27,1
3	6 052	43,3	57,1
4	11 957	11,6	42,7
5	35 252	9,7	44,6

TABLE 4.1 – Description des ensembles de *reads* MinION du jeu de donnée utilisée. La dernière colonne indique la taille totale des données représentée par les *reads* 2D.

Pour obtenir une valeur témoin, ces *reads* longs ont été alignés sur le génome de référence avant traitement, à l'aide de LAST [26]. 83,2% des *reads* 2D, et seulement 16,6% des *reads* 1D, soit un total de 25% de l'ensemble de tous les *reads*, ont ainsi été effectivement alignés, avec une identité moyenne de 56,5% pour les *reads* 1D, et de 74,5% pour les *reads* 2D. Comme indiqué dans la description des séquenceurs Nanopore, Section 2.2, ces résultats montrent que les *reads* 2D du jeu de données utilisé ici disposent en effet d'un taux d'erreurs de 25,5%, moins important que le taux d'erreurs des *reads* 1D, atteignant 43,5%.

Le jeu de données se compose également de deux ensembles de *reads* courts provenant, quant à eux, d'une bibliothèque Illumina. Ces ensembles sont tous deux composés

de 5 984 858 *reads*, mais en ne conservant que les *reads* d'une longueur supérieure ou égal à 250, ces deux ensembles ne présentent plus que 5 915 778 et 5 542 347 *reads*, respectivement, après filtrage.

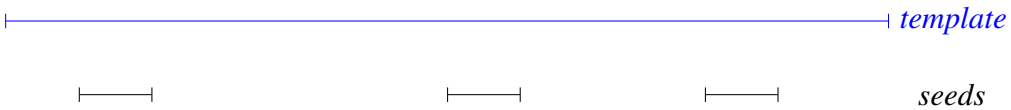
4.2 Première méthode

Une première méthode permettant d'obtenir de tels *reads* a donc été développée en 2015 dans [25], au côté de l'introduction de la notion de *reads* NaS. Elle nécessite l'alignement des *reads* courts sur les *reads* longs, mais également l'alignement des *reads* courts entre eux. Nous décrivons ci-dessous cette méthode, ainsi que le principal outil sur lequel elle repose.

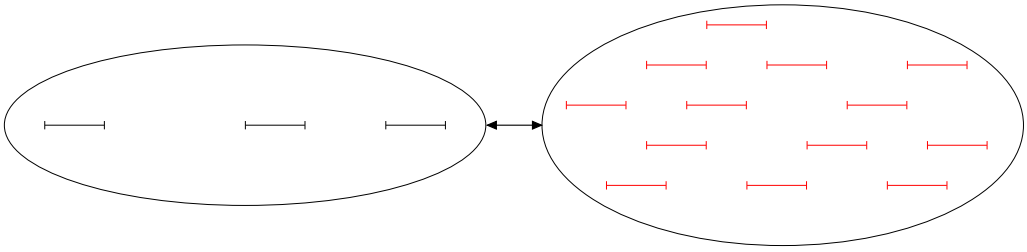
4.2.1 Présentation de la méthode

Les *reads* courts sont tout d'abord alignés sur les *reads* longs *templates*, afin de trouver les *reads* s'alignant totalement, et pouvant ainsi servir de *seeds*. De nouveaux *reads*, similaires aux *seeds*, et n'ayant pas été alignés précédemment, sont ensuite recrutés en alignant les *reads* courts entre eux, ce processus de recrutement étant itéré jusqu'à obtention d'une couverture suffisante des *templates*, afin de produire des assemblages locaux par la suite, et ainsi, obtenir des *reads* synthétiques de haute qualité. Le processus de recrutement de *reads*, et donc de synthèse de *reads* NaS, étant engendré par les *seeds*, il est clairement nécessaire qu'au moins un *seed* soit aligné sur un *read* long *template* donné pour que celui-ci puisse produire un *read* NaS. Une illustration des quatre étapes nécessaires à la synthèse d'un *read* NaS est donnée Figure 4.1.

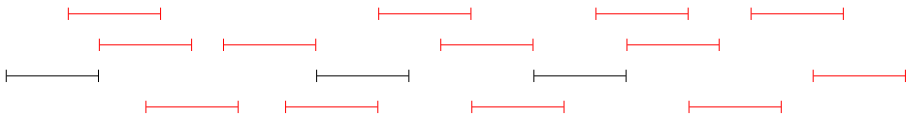
1. Alignement des *reads* courts sur le *read* long *template*, afin de trouver les *seeds*



2. Recrutement de nouveaux *reads* similaires aux *seeds*, en alignant les *reads* courts entre eux



3. Micro-assemblage de l'ensemble de *reads* obtenu



4. Obtention d'un contig

FIGURE 4.1 – Procédure de synthèse d'un *read* NaS. Le contig obtenu en fin de traitement est considéré comme un *read* NaS s'il est unique.

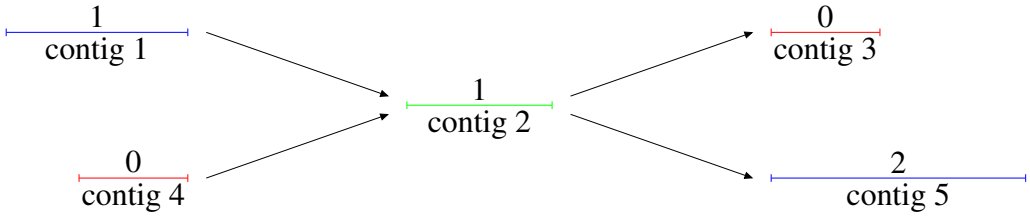
Le point 2. est le plus crucial de cette méthode, car il permet de retrouver des *reads* courts correspondant à des régions de faible qualité du *read* long utilisé comme *template*. Nous décrivons la relation de similarité entre les *seeds* et les autres *reads* courts et détaillons la méthode de recrutement utilisée en Section 4.2.2.

En général, un seul contig est produit par cette méthode, mais dans les régions répétitives, quelques *reads* ne venant pas de régions correctes peuvent être recrutés, et ainsi produire des contigs erronés ne devant pas être associés au *template*. Pour résoudre ce

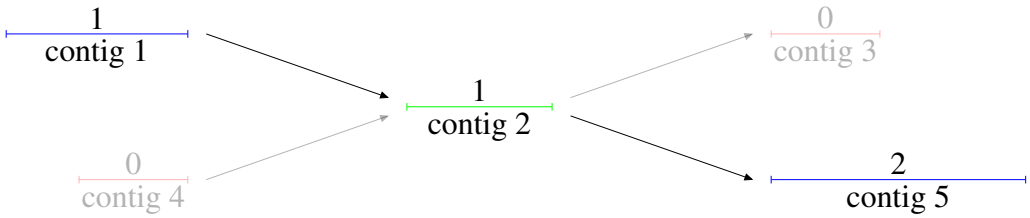
problème et ne produire qu'un seul contig en sortie, et donc un *read* NaS, il suffit, à la fin du traitement, de construire explicitement le graphe des contigs, pondéré par le degré de couverture des contigs par les *seeds*. Une fois le graphe construit, il suffit alors de choisir le chemin passant par les contigs ayant le plus haut degré de couverture par les *seeds*, par exemple à l'aide de l'algorithme de Floyd-Warshall. Le contig ainsi produit est ensuite vérifié par alignement des *reads* courts, et est considéré comme correct, et donc comme un *read* NaS, si la couverture est suffisante. Une illustration d'un tel cas est donnée Figure 4.2.

1. Obtention de plusieurs contigs

2. Construction du graphe des contigs, pondéré par le degré de couverture des contigs par les *seeds*



3. Sélection du chemin optimal, passant par les contigs ayant le plus haut degré de couverture par les *seeds*



4. Vérification du contig produit par alignement des *reads* courts, et acceptation si la couverture est suffisante

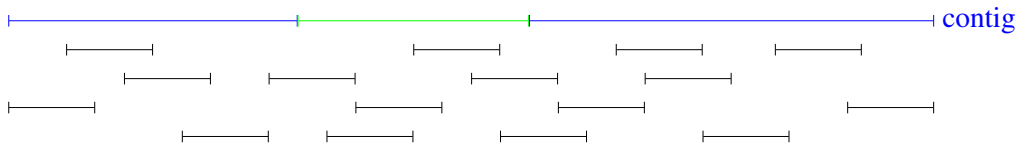


FIGURE 4.2 – Procédure de production d'un unique contig, dans le cas où le micro-assemblage a généré de multiples contigs.

4.2.2 Méthode de recrutement des *reads* : Commet

Comme indiqué précédemment, l'étape la plus cruciale, mais aussi la plus longue de la méthode, étant en moyenne responsable de 70% du temps de traitement, est le recrutement de *reads*. Cette étape nécessite l'alignement de l'ensemble des *reads* courts entre eux, et utilise à ces fins Commet (*COmpare Multiple METagenomes*), un outil introduit en 2014 dans [27].

Commet permet de comparer un ensemble de *reads* non assemblés les uns aux autres, à l'aide d'une stratégie d'indexation efficace. Pour cela, les résultats sont stockés sous la forme de vecteurs de bits, une représentation bien plus compacte des fichiers de *reads*, qui permet également de filtrer les *reads* mais aussi de combiner les différents sous-ensembles de *reads* facilement, à l'aide des opérations logiques. Les *reads* sont comparés en fonction de leur similarité, deux *reads* étant considérés comme similaires s'ils partagent au moins t k-mers distincts, ne se chevauchant pas. La méthode permettant de comparer deux ensembles de *reads* A et B consiste alors à trouver tous les *reads* de A étant similaires à au moins un *read* de B (ou inversement). Elle repose sur une opération dirigée, notée $A \overset{\rightsquigarrow}{\cap} B$, et fournissant donc les *reads* de A similaires à des *reads* de B, mais pas l'inverse. Cette opération est une heuristique, et se calcule comme suit :

1. Les k-mers de B sont indexés dans un Bloom Filter Trie (BFT) [28]
2. Les k-mers des *reads* de A ne se chevauchant pas sont recherchés dans le BFT

L'étape 2. ne vérifie cependant pas si les k-mers des *reads* de A apparaissent dans un seul et même *read* de B, et peut donc produire de faux positifs. Ainsi, pour comparer deux ensembles de *reads* A et B, tout en limitant les faux positifs, mais également la taille de l'index, les opérations suivantes sont réalisées :

1. Calcul de $A \overset{\rightsquigarrow}{\cap} B$
2. Calcul de $B \overset{\rightsquigarrow}{\cap} (A \overset{\rightsquigarrow}{\cap} B)$
3. Calcul de $A \overset{\rightsquigarrow}{\cap} (B \overset{\rightsquigarrow}{\cap} (A \overset{\rightsquigarrow}{\cap} B))$ (Noté $A \overset{\rightsquigarrow}{\Cap} B$)

Comparer les ensembles A et B à l'aide de trois opérations $\overset{\rightsquigarrow}{\cap}$ successives permet en effet de limiter l'effort d'indexation, seul le premier calcul indexant l'ensemble B complet, et les suivant n'indexant que des sous-ensembles de A et de B. Le résultat intéressant de ces calculs est alors $A \overset{\rightsquigarrow}{\Cap} B$, indiquant les *reads* de A similaires à des *reads* de B. De façon similaire, il est possible d'obtenir les *reads* de B similaires à des *reads* de A, en réalisant l'opération symétrique $B \overset{\rightsquigarrow}{\Cap} A$. Il est également possible de comparer plusieurs ensembles

de *reads* aisément, en généralisant cette méthode. Sur un jeu de données composé de 28 ensembles de 10 000 *reads*, avec $t = 2$, et des k-mers de longueur 35, Commet a permis de calculer l'ensemble des 756 intersections en 35 minutes. Cependant, bien qu'efficace, cet outil se montre trop lent pour l'usage désiré dans la synthèse de *reads* NaS.

4.2.3 Résultats

La synthèse de *reads* NaS a été testée sur les 66 492 *reads* MinION du génome de la bactérie *Acinetobacter baylyi ADP1* du jeu de données précédemment décrit, à l'aide de plusieurs sous-ensembles de *reads* Illumina de longueur 250. À partir de ces données, 11 275 *reads* NaS, d'une longueur maximale de 59 863, ont été produits, 50 *reads* de cet ensemble ayant une longueur supérieure à 50 000. Certains *reads* produits sont plus longs que leur *template* de référence, notamment à cause de l'étape de recrutement, retrouvant des *reads* situés en dehors du *template*, et à la longueur des *reads* sélectionnés de la bibliothèque Illumina. On remarque donc que seulement 17% des *templates* MinION ont produit un *read* NaS, ce qui est dû au fort taux d'erreurs des *reads* MinION, et en accord avec les 25% de *reads* MinION non traités, mappés sur le génome de référence lors du test préliminaire. Plus précisément, 76,4% des *templates* 2D ont produit un *read* NaS, contre seulement 8,1% des *templates* 1D, ce qui est également en accord avec le plus fort taux d'erreurs observé dans les *reads* 1D lors du test d'alignement préliminaire.

Le temps de traitement d'un *read* MinION, et donc la production d'un *read* NaS, est de moins d'une minute en moyenne, soit un temps total d'environ 7 jours pour la synthèse des 11 275 *reads* NaS obtenus en fin de traitement, la majorité de ce temps étant due à la méthode peu efficace de recrutement des *reads*.

Après la production des *reads* NaS, leur qualité a été inspectée en les alignant sur le génome de référence avec BWA, un aligneur reposant sur la transformée de Burrows-Wheeler, introduit en 2009 dans [29]. Les *reads* NaS ainsi alignés couvrent 99,96% du génome, avec une identité moyenne de 99,99%, 97% de ces *reads* s'alignant sans aucune erreur, et 99,2% s'alignant avec seulement une erreur, les 50 *reads* d'une longueur supérieure à 50 000 produits s'alignant notamment parfaitement. À titre de comparaison, après correction des *reads* MinION à l'aide de Proovread, un outil développé en 2014 dans [30], et destiné à la correction de *reads* longs séquencés par la technologie Pacific Bioscience, disposant donc d'un plus faible taux d'erreurs, seulement 63,35% des *reads* corrigés ont été effectivement *mappés* avec BWA, avec une identité moyenne de 71,6%.

Ainsi, la production de *reads* NaS se montre en effet très efficace, semble offrir une excellente alternative à la correction de *reads* MinION par des méthodes plus classiques,

et pourrait se révéler extrêmement utile, notamment en permettant de traiter plus aisément des problèmes concernant de larges génomes répétitifs.

4.3 Notre méthode

Le travail réalisé durant la rédaction de ce mémoire s'est donc principalement porté sur le développement d'une nouvelle méthode de production de *reads* synthétiques. Notre méthode conserve le principe de base de la solution précédente, et repose toujours sur une approche hybride, utilisant des *reads* longs et bruités comme *templates*, sur lesquels des *reads* courts, plus précis, sont alignés. Cependant, notre méthode vise à ne déduire des informations qu'à partir de cet alignement des *reads* courts sur les *reads* longs, et ne nécessite donc pas l'alignement des *reads* courts entre eux.

4.3.1 Présentation de la méthode

Comme pour la méthode précédente, nous alignons tout d'abord les *reads* courts sur les *reads* longs *templates*, afin de trouver les *reads* courts s'alignant totalement, et pouvant servir de *seeds*. Cependant nous désirons aussi, pour cette méthode, récupérer les *reads* courts dont seulement un préfixe ou un suffixe, d'au moins une longueur fixée *lmin*, s'est aligné. Ici, les *reads* courts sont alignés sur les *reads* longs *templates* à l'aide d'une version modifiée de BLAT [31], un outil permettant le *mapping* de *reads* en indexant le génome de référence, et donc dans notre cas les *reads* longs, s'étant révélé plus efficace que les autres outils pour cette utilisation. Plus exactement, nous utilisons une version modifiée de PBLAT, une version de BLAT permettant l'exécution en multithreads. Le fonctionnement de BLAT et les modifications apportées sont brièvement décrites en Section 4.3.2. Cette procédure d'alignement nous permet donc de récupérer les différents ensembles de *reads* courts désirés, en différenciant les *reads* :

— Totalement alignés
— Avec un préfixe aligné
— Avec un suffixe aligné

Pour chaque *read* long, les différents ensembles de *reads* courts, obtenus par l'alignement et par le classement précédent, sont ajoutés à trois différentes listes, qui sont ensuite triées en fonction des positions de début d'alignement préalablement calculées. La liste des *reads* totalement alignés représente ainsi les *seeds*, et les listes des *reads* avec seulement

un préfixe ou seulement un suffixe aligné, les recrues potentielles. La structure de données nécessaire au stockage des *reads* courts dans les listes est décrite en Section 4.3.3.

Une fois les listes crées et triées, notre méthode se décompose en deux différentes étapes :

— La première parcourt les trois listes en parallèle, afin de recruter de nouveaux *reads* similaires aux *seeds*, et met à jour la liste des *seeds* en fonction des recrutements effectués, afin d'étendre ces derniers, et ainsi obtenir des contigs couvrant davantage le *read* long *template* considéré. Nous décrivons la relation de similarité entre les *seeds* et les *reads* courts partiellement alignés, ainsi que la méthode de recrutement des *reads* similaires en Section 4.3.4.

— La seconde étape, quant à elle, parcourt à nouveau les trois listes en parallèle afin d'étendre de nouveau les contigs obtenus par la mise à jour des *seeds* à l'étape précédente, et de couvrir davantage le *read* long *template* considéré, en recrutant de nouveaux *reads* partiellement alignés, mais en s'affranchissant cette fois de la relation de similarité. Nous décrivons la méthode de recrutement des *reads*, pour cette seconde étape, en Section 4.3.5.

4.3.2 Version modifiée de PBLAT

Nous présentons tout d'abord brièvement le fonctionnement de BLAT, le programme sur lequel PBLAT est basé, et n'ajoute qu'une surcouche permettant l'exécution en multi-threads.

BLAT commence par construire un index de tous les *k*-mers du génome de référence ne se chevauchant pas, et de leurs positions, en excluant ceux apparaissant plus souvent qu'un certain seuil fixé. La taille de l'index construit est relativement faible et permet son stockage en RAM sur des machines classiques, l'indexation d'un génome humain demandant par exemple seulement 0,9 Go de mémoire. Les *reads* à aligner sont ensuite parcourus, et pour chaque *read*, l'ensemble de ses *k*-mers, se chevauchant sur une certaine longueur, est recherché dans l'index. Une liste des *hits* ainsi découverts, contenant les positions de ces *hits* dans le génome de référence et dans le *read* est alors construite et triée, afin de déterminer les régions du génome de référence homologues au *read*. Une fois ces régions détectées, une phase d'alignement est réalisée, et la liste préalablement créée est alors parcourue afin d'étendre le plus possibles les *hits* précédemment obtenus, en les fusionnant s'ils se chevauchent, dans le but de produire l'alignement final du *read* sur le génome de référence, s'il existe. BLAT se montre très efficace en termes de temps,

et permet, par exemple, de *mapper* 1 000 *reads* provenant d'un génome de souris sur un génome de référence en seulement 37 secondes.

Comme indiqué précédemment, nous indexons ici un ensemble de *reads* longs et non un génome, et avons besoin de récupérer non seulement les *reads* courts complètement alignés sur les *reads* longs, mais également les *reads* courts dont seulement un préfixe ou un suffixe, de longueur suffisante, s'est aligné.

PBLAT ne permettant pas de produire de tels résultats nativement, nous avons été amenés à étudier et à modifier son code source, afin de lui permettre de fournir l'ensemble des alignements désirés. En particulier, nous avons modifié la fonction permettant de filtrer et d'enregistrer les alignements. Nous y avons notamment ajouté des filtres supplémentaires, permettant, si l'alignement trouvé pour un *read* donné n'est pas total, de vérifier si un préfixe ou un suffixe de ce *read* s'aligne sur une longueur supérieure ou égale au seuil *lmin* préalablement fixé, et le cas échéant, d'enregistrer l'alignement. Nous avons également modifié le format de sortie de PBLAT, afin d'obtenir en fin de traitement un fichier plus concis, permettant un traitement des données d'alignement plus aisé en aval.

Ce format se compose d'une ligne par alignement d'un *read* court sur un *read* long découvert, et chacune de ces lignes comporte les informations suivantes, séparées par des tabulations : type d'alignement (total, préfixe, ou suffixe), longueur du *read* court, longueur de l'alignement, position de début de l'alignement, ID du *read* long, longueur du *read* long, et séquence ADN du *read* court.

4.3.3 Structure de données

Comme décrit précédemment, le traitement des données par notre méthode nécessite l'ajout des différents ensembles de *reads*, totalement ou partiellement alignés, dans des listes. Pour cela, il s'est révélé nécessaire de développer une structure de données permettant de stocker les informations concernant les alignements des *reads* courts sur les *reads* longs, fournies en sortie par notre version de PBLAT. Pour un alignement d'un *read* court sur un *read* long donné, les différentes informations le concernant sont donc stockées dans la structure suivante :

```
struct alignement {
  int rlen;
  int mlen;
  int pos;
  char* seq;
}
```

Où `rlen` représente la longueur du *read* court, `mlen` la longueur de la partie du *read* court effectivement alignée sur le *read* long, `pos`, la position de début d'alignement sur le *read* long, et `seq`, la séquence ADN du *read* court dont l'alignement est décrit.

4.3.4 Première étape : Recrutement de *reads* similaires

Nous considérons ici qu'un *seed* et qu'un *read* court, dont seul un préfixe, de longueur au moins *lmin*, s'est aligné sont similaires si le préfixe correctement aligné du *read* court chevauche le suffixe du *seed* sur une longueur supérieure ou égale au seuil défini par *lmin*. Symétriquement, un *seed* et un *read* court, dont seul un suffixe, de longueur au moins *lmin*, s'est aligné sont similaires si le suffixe correctement aligné du *read* court chevauche le préfixe du *seed* sur une longueur supérieure ou égale à ce même seuil *lmin*. Une illustration de la similarité entre *seeds* et *reads* courts partiellement alignés est donnée Figure 4.3.

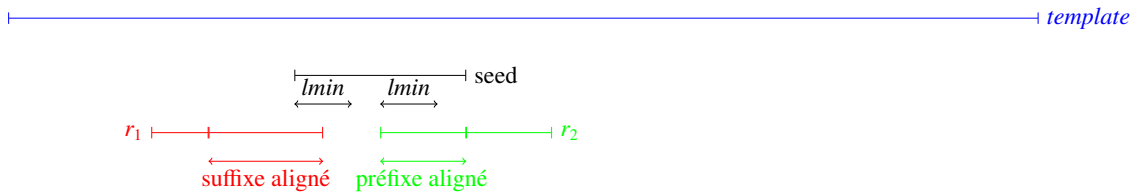


FIGURE 4.3 – Illustration de la similarité entre *seeds* et *reads* courts partiellement alignés. *lmin* représente le seuil permettant de déterminer cette similarité. Ici, le suffixe aligné du *read* r_1 ne chevauche pas le préfixe du *seed* sur une longueur suffisante, puisque bien inférieure au seuil *lmin* fixé, le *read* r_1 et le *seed* ne sont donc pas similaires. En revanche, le préfixe aligné du *read* r_2 chevauche le suffixe du *seed* sur une longueur légèrement supérieure au seuil *lmin* fixé, et r_2 et le *seed* sont donc bien similaires.

Parmi ces *reads* similaires aux *seeds*, nous ne recrutons cependant un *read* que s'il permet d'étendre le *seed* auquel il est similaire. De plus, s'il est possible de recruter plusieurs *reads* pour un même *seed*, nous choisirons toujours le *read* permettant d'étendre au maximum le *seed*. Ainsi, s'il est possible de recruter plusieurs *reads* dont un suffixe s'est correctement aligné, nous choisirons toujours celui ayant la position de début d'alignement la plus petite, car il permettra d'étendre le *seed* au maximum, à gauche. Symétriquement, s'il est possible de recruter plusieurs *reads* dont un préfixe s'est correctement aligné, nous choisirons toujours celui ayant la position de début d'alignement la plus grande, car il permettra d'étendre le *seed* au maximum, à droite. Lors d'un recrutement, le *seed* considéré est alors mis à jour afin de prendre en compte l'extension provoquée par le recrutement

et un contig est ainsi obtenu. Une illustration de la procédure de recrutement de *reads* similaires, est donnée Figure 4.4.

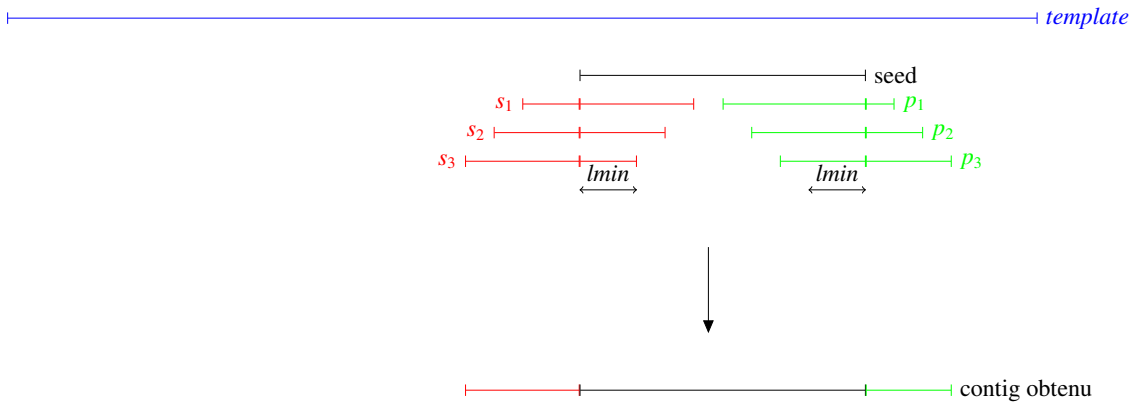


FIGURE 4.4 – Illustration du processus de recrutement de *reads* similaires pour un *seed* donné. Tous les *reads* partiellement alignés sont ici similaires au *seed*, et peuvent être recrutés. Les *reads* s_3 et p_3 permettent d'étendre le *seed* au maximum, à gauche et à droite respectivement, et sont donc effectivement recrutés. Le *seed* est alors mis à jour avec les parties ne le chevauchant pas des *reads* recrutés, et un contig est obtenu.

L'algorithme détaillé du parcours parallèle des listes, pour le recrutement des *reads* similaires, pour le traitement d'un *read* long *template* fixé, est donné Figure 4.5.

```
RECRUTEMENT(seeds[], n, suffs[], m, suffs[], p, lmin)
▹ Entrée : seeds[] : liste des seeds, n : taille de seeds[],
▹          suffs[] : liste des reads avec un suffixe aligné, m : taille de suffs[],
▹          prefs[] : liste des reads avec un préfixe aligné, p : taille de prefs[],
▹          lmin : seuil déterminant la similarité
1  Début
2    QSORT(seeds[])
3    QSORT(prefs[])
4    QSORT(suffs[])
5    FUSION(seeds[])
6    k ← 0
7    j ← 1
8    pour i ← 0 à n faire
9      tantque k < m et suffs[k].pos + suffs[k].rlen < seeds[i].pos + lmin faire
10       k ← k + 1
11     fin tantque
12     si k < m et suffs[k].pos < seeds[i].pos alors
13       seeds[i].seq ← CONCATENATION(suffs[k].seq[0 .. seeds[i].pos − suffs[k].pos],
         seeds[i].seq)
14       seeds[i].rlen ← seeds[i].pos + seeds[i].rlen − suffs[k].pos
15       seeds[i].pos ← suffs[k].pos
16       k ← k + 1
17     fin si
18     tantque j < p et prefs[j].pos ≤ seeds[i].pos + seeds[i].rlen − lmin faire
19       j ← j + 1
20     fin tantque
21     si prefs[j − 1].pos > seeds[i].pos
         et prefs[j − 1].pos ≤ seeds[i].pos + seeds[i].rlen − lmin
         et prefs[j − 1].pos + prefs[j − 1].rlen > seeds[i].pos + seeds[i].rlen alors
22       seeds[i].seq ← CONCATENATION(seeds[i].seq,
         prefs[j − 1].seq[seeds[i].pos + seeds[i].rlen − prefs[j − 1].pos .. prefs[j − 1].rlen])
23       seeds[i].rlen ← prefs[j − 1].pos + prefs[j − 1].rlen − seeds[i].pos
24     fin si
25   fin pour
26   FUSION(seeds[])
27 Fin
```

FIGURE 4.5 – Algorithme de recrutement de *reads* similaires, pour le traitement d'un *read* long *template* donné.

Les lignes 2 - 4 trient des listes contenant les *seeds* et les *reads* partiellement alignés, selon la position de début d'alignement. La ligne 5 fusionne les *seeds* se chevauchant déjà avant traitement. La boucle à la ligne 8 permet d'itérer le processus de recrutement pour tous les *seeds*. La boucle aux lignes 9 - 11 permet de rechercher le *read* avec un suffixe aligné, similaire au *seed* en cours de traitement, permettant de maximiser, à gauche, l'extension du *seed*. La condition à la ligne 12 permet de vérifier si un tel *read* a été trouvé, et les lignes 13 - 15 recrutent effectivement ce *read* et mettent à jour les informations du *seed* pour prendre en compte l'extension le cas échéant. La boucle aux lignes 18 - 20 permet de rechercher le *read* avec un préfixe aligné, similaire au *seed* en cours de traitement, permettant de maximiser, à droite, l'extension du *seed*. La condition à la ligne 21 permet de vérifier si un tel *read* a été trouvé, et les lignes 22 - 23 recrutent effectivement ce *read* et mettent à jour les informations du *seed* pour prendre en compte l'extension le cas échéant. Enfin, la ligne 26 fusionne les contigs se chevauchant, une fois le traitement terminé.

4.3.5 Deuxième étape : Extension des contigs obtenus

Comme indiqué précédemment, pour cette deuxième étape de recrutement, visant à étendre les contigs produits par la première étape, nous nous affranchissons de la relation de similarité entre les *seeds* et les *reads* partiellement alignés. À la place, nous définissons un seuil *lmax*, et recrutons un *read* partiellement aligné afin d'étendre un contig donné si le préfixe (respectivement le suffixe) correctement aligné de ce *read* chevauche le contig sur une longueur inférieure ou égale au seuil *lmax* fixé. De plus, comme précédemment, s'il est possible de recruter plusieurs *reads* pour un même contig, nous choisirons toujours le *read* permettant d'étendre au maximum le contig. Ainsi, s'il est possible de recruter plusieurs *reads* dont un suffixe s'est correctement aligné, nous choisirons toujours celui ayant la position de début d'alignement la plus petite, car il permettra d'étendre le contig au maximum, à gauche. Symétriquement, s'il est possible de recruter plusieurs *reads* dont un préfixe s'est correctement aligné, nous choisirons toujours celui ayant la position de début d'alignement la plus grande, car il permettra d'étendre le contig au maximum, à droite. Lors d'un recrutement, le contig considéré est alors mis à jour afin de prendre en compte l'extension provoquée par le recrutement. Une illustration de la procédure d'extension des contigs, par recrutement de *reads*, est donnée Figure 4.6.

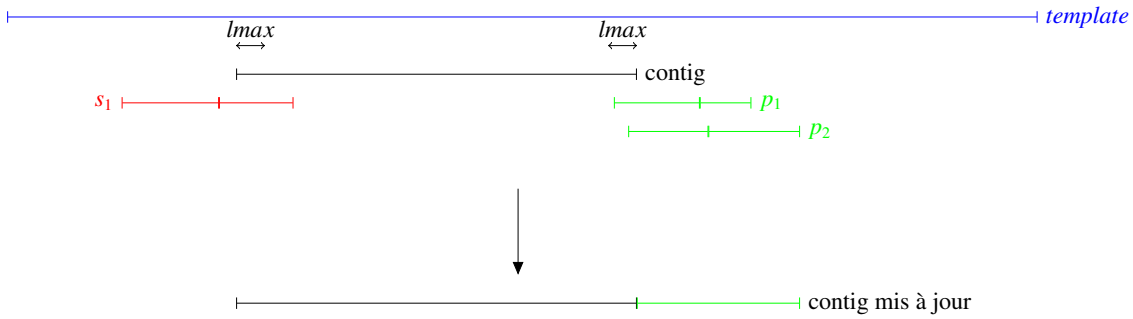


FIGURE 4.6 – Illustration du processus d'extension pour un contig donné. Le *read* s_1 chevauche le contig sur un longueur supérieure à *lmax* et par conséquent, ne peut pas être recruté. Les *reads* p_1 et p_2 chevauchent tous les deux le contig sur une longueur inférieure ou égale à *lmax* et peuvent donc être recrutés. Cependant, le *read* p_2 permet d'étendre le contig, à droite, sur une longueur plus importante que le *read* p_1, et p_2 est donc recruté. Le contig est alors mis à jour avec la partie ne le chevauchant pas du *read* recruté, afin d'étendre sa longueur.

L'algorithme détaillé du parcours parallèle des listes, pour l'extension des contigs, pour le traitement d'un *read* long *template* fixé, est donné Figure 4.7.

```
RECRUTEMENT(contigs[], n, suffs[], m, prefs[], p, lmax, nb)
▹ Entrée : contigs[] : liste des contigs obtenus à l'étape précédente,
▹          n : taille de contigs[],
▹          suffs[] : liste des reads avec un suffixe aligné, m : taille de suffs[],
▹          prefs[] : liste des reads avec un préfixe aligné, p : taille de prefs[],
▹          lmax : seuil utilisé pour le recrutement,
▹          nb : nombre d'itérations souhaitées
1  Début
2    pour e ← 0 à nb faire
3      k ← 0
4      j ← 1
5      pour i ← 0 à n faire
6        tantque k < m et suffs[k].pos + suffs[k].rlen < contigs[i].pos faire
7          k ← k + 1
8        fin tantque
9        si k < m et suffs[k].pos + suffs[k].rlen ≤ contigs[i].pos + lmax alors
10         contigs[i].seq ← CONCATENATION(suffs[k].seq[0 .. contigs[i].pos − suffs[k].pos],
           contigs[i].seq)
11         contigs[i].rlen ← contigs[i].pos + contigs[i].rlen − suffs[k].pos
12         contigs[i].pos ← suffs[k].pos
13         k ← k + 1
14       fin si
15       tantque j < p et prefs[j].pos ≤ contigs[i].pos + contigs[i].rlen faire
16         j ← j + 1
17       fin tantque
18       si prefs[j − 1].pos ≥ contigs[i].pos + contigs[i].rlen − lmax
           et prefs[j − 1].pos ≤ contigs[i].pos + contigs[i].rlen alors
19         contigs[i].seq ← CONCATENATION(contigs[i].seq,
           prefs[j − 1].seq[contigs[i].pos + contigs[i].rlen − prefs[j − 1].pos .. prefs[j − 1].rlen])
20         contigs[i].rlen ← prefs[j − 1].pos + prefs[j − 1].rlen − contigs[i].pos
21       fin si
22     fin pour
23     FUSION(contigs[])
24   fin pour
25 Fin
```

FIGURE 4.7 – Algorithme d'extension des contigs, pour le traitement d'un *read* long *template* donné.

Les listes ayant déjà été triées, et les contigs se chevauchant ayant déjà été fusionnés lors de l'étape précédente, il n'est pas nécessaire de réaliser à nouveau ces deux opérations en début de traitement. La boucle à la ligne 2 permet d'itérer le processus d'extension le nombre de fois désiré. La boucle à la ligne 5 permet d'appliquer le processus d'extension à tous les contigs. La boucle aux lignes 6 - 8 permet de rechercher le *read* avec un suffixe aligné, permettant de maximiser, à gauche, l'extension du contig en cours de traitement. La condition à la ligne 9 permet de vérifier si un tel *read* a été trouvé, et les lignes 10 - 12 recrutent effectivement ce *read* et mettent à jour les informations du contig pour prendre en compte l'extension le cas échéant. La boucle aux lignes 15 - 17 permet de rechercher le *read* avec un préfixe aligné, permettant de maximiser, à droite, l'extension du contig en cours de traitement. La condition à la ligne 18 permet de vérifier si un tel *read* a été trouvé, et les lignes 19 - 20 recrutent effectivement ce *read* et mettent à jour les informations du contig pour prendre en compte l'extension le cas échéant. Enfin, la ligne 23 fusionne les contigs se chevauchant, une fois le traitement d'une itération terminé.

4.3.6 Résultats

Nous présentons ici les différents résultats obtenus lors des tests notre méthode sur le jeu de donné présenté en Section 4.1.

Pour enclencher le processus de recrutement de *reads*, et donc de synthèse d'un *read* NaS, pour un *read* long donné, il est nécessaire qu'au moins un *seed* s'aligne sur celui-ci. Nous présentons, Table 4.2, les statistiques concernant le nombre de *reads* long 1D et 2D des différents ensembles de notre jeu de données, sur lesquels au moins un *seed* a pu être aligné.

Ensemble	*reads* 1D	*reads* 1D avec au moins un *seed*	*reads* 2D	*reads* 2D avec au moins un *seed*
1	8 639	508 (5,88%)	602	401 (66,61%)
2	3 447	463 (13,43%)	543	322 (59,30%)
3	3 433	312 (9,09%)	2 619	1 616 (61,70%)
4	10 567	492 (4,66%)	1 390	943 (67,84%)
5	31 825	1 713 (5,38%)	3 427	2 871 (87,78%)

TABLE 4.2 – Répartition des *seeds* pour les différents ensembles du jeu de données utilisé.

Ces résultats montrent que 6% des *reads* 1D et 71,7% des *reads* 2D, soit au total 9 641 *reads* de l'ensemble des *reads* longs du jeu de données, peuvent potentiellement produire un *read* NaS à l'aide de notre méthode, ce qui est légèrement plus faible que les 11 275 *reads* NaS produits par la première méthode. Cette différence s'explique par l'ajout de filtres à notre version de PBLAT, permettant de différencier les alignements totaux des

alignements locaux de préfixes ou de suffixes, et tolérant un plus faible nombre d'erreurs lors du *mapping*, produisant ainsi moins de *seeds*.

Les résultats obtenus, après exécution de notre méthode sur les différents ensembles de *reads* longs précédents, sur lesquels au moins un *seed* a pu être aligné, sont décrits Table 4.3.

Ensemble	*reads*	Contigs / *read*	Longueur moyenne	Précision moyenne	Plus long contig (précision)	Temps
1	1D	1,52	604	89,60	7 791 (94%)	54 s
	2D	2,23	1 407	88,14	10 806 (94%)	3 min 06 s
2	1D	2,42	623	87,90	3 728 (95%)	1 min 20 s
	2D	3,17	1 495	87,79	17 797 (94%)	4 min 32 s
3	1D	1,97	589	89,09	5 506 (94%)	37 s
	2D	3,03	2 139	87,91	22 305 (94%)	1 h 57 min
4	1D	2,68	665	88,65	7 372 (92%)	1 min 26 s
	2D	2,82	2 717	88,06	28 059 (93%)	1 h 01 min 50 s
5	1D	2,89	742	87,94	15 223 (91%)	15 min 35 s
	2D	2,41	4 349	88,83	33 317 (91%)	11 h

TABLE 4.3 – Résultats obtenus après exécution de notre méthode et alignement des contigs produits à l'aide de BWA. Les paramètres de notre méthode ont été fixés comme suit : $lmin = 100$, $lmax = 10$, et 10 itérations pour la deuxième phase d'extension. Les paramètres de BWA ont été fixés aux valeurs par défaut. La deuxième colonne représente le nombre moyen de contigs obtenus par *read* long *template*.

Notre méthode permet donc de produire rapidement des contigs relativement longs, et plus précis que leurs *templates* d'origine. Le temps de traitement d'un *read* long est en effet de moins de dix secondes en moyenne, et l'ensemble des 9 641 *reads* sur lesquels au moins un *seed* a pu être aligné a été traité en un peu moins de 14 h 30 min au total. Ainsi, notre méthode se montre environ 11,6 fois plus rapide que la méthode précédente.

Nous obtenons, en moyenne, 2,296 contigs de longueur 645 et de précision 88,636% à partir d'un *read* 1D, tandis que les *reads* 1D initiaux affichent une longueur moyenne de 2 052 et une précision de 56,5%. Concaténés, les contigs produits couvrent donc en moyenne 72,17% de leur *template* 1D d'origine. De même, à partir d'un *read* 2D, nous obtenons en moyenne, 2,732 contigs de longueur 2 421 et de précision 88,186%, tandis que les *reads* 2D initiaux affichent une longueur moyenne de 10 033, et une précision de 74,5%. Concaténés, les contigs produits couvrent donc en moyenne seulement 65,93% de leur *template* 2D d'origine. Nous remarquons cependant que, concaténés, les contigs produits à partir des *reads* 1D de l'ensemble 5 atteignent, en moyenne, une longueur plus importante que leur *template* d'origine. Cela s'explique par le fait que notre méthode,

comme la précédente, peut recruter des *reads* courts pour étendre les *seeds* ou les contigs, même si ces *reads* se situent en dehors du *template*.

L'application de notre méthode a donc permis de réduire le taux d'erreurs des contigs produits à moins de 12% en moyenne, aussi bien pour les contigs obtenus à partir de *reads* 2D, qui présentaient à la base un taux d'erreurs de 25,5%, que pour les contigs obtenus à partir de *reads* 1D, qui eux, présentaient à la base un taux d'erreurs de 43,5%.

Le faible taux de couverture des *templates* d'origine observé, quant à lui, peut notamment être expliqué par la synthèse de contigs courts, produisant une nette diminution de ce taux. Ces contigs courts sont en fait, dans la plupart des cas, des *seeds* qu'il a été impossible d'étendre, aussi bien par la première que par la deuxième étape de notre méthode. Ainsi, il serait intéressant de réaliser de nouveaux tests, en ajustant les paramètres *lmin* et *lmax*, afin observer s'ils permettent d'étendre d'avantage les *seeds*, et donc d'obtenir une meilleure couverture des *templates*.

Ainsi, l'application de notre méthode semble déjà constituer un prétraitement efficace pour la production de *reads* NaS. Il serait maintenant nécessaire, en plus d'ajuster les paramètres, d'étudier plus en détails les contigs produits, afin d'observer d'éventuels chevauchements permettant de les fusionner, et de produire, pour chaque *read* long, un unique contig de longueur plus importante, et donc un éventuel *read* NaS.

Conclusion et perspectives

Ce mémoire a donc permis de dresser l'état de l'art concernant les différentes technologies et plateformes de séquençage, et les solutions, utilisant une structure d'index sur les *reads*, permettant de résoudre des problèmes de *mapping*, de correction, ou de traitement des 7 requêtes définies dans l'introduction. Nous avons ainsi pu observer l'évolution rapide des plateformes de séquençage, notamment concernant leur prix et la longueur des *reads* qu'elles permettent de produire, mais également l'intérêt d'utiliser des structures de données évoluées plutôt que des structures de données classiques lors de l'indexation des *reads*, particulièrement lorsque les études se portent sur des génomes de taille importante.

Nous avons plus particulièrement étudié les *reads* longs, se développant depuis peu, et se montrant très utiles, permettant notamment le traitement de problèmes d'assemblage longs et complexes, jusqu'ici impossible à résoudre à l'aide des *reads* courts. Ces *reads* longs sont cependant très bruités, affichant un taux d'erreurs de séquençage pouvant atteindre 30%, et sont donc difficiles à utiliser. Nous avons pu voir que les techniques de corrections classiques ne sont que très peu efficaces sur de tels *reads*, et avons de ce fait introduit une méthode alternative permettant d'appliquer un traitement correctif à ces *reads* avant utilisation, à travers les *reads* NaS.

Nous avons décrit une première méthode de synthèse de tels *reads*, produisant des contigs longs et uniques, et étant très efficace en termes de précision, mais très peu en termes de temps. Celle-ci a permis d'ouvrir la porte au développement de notre méthode, quant à elle bien plus efficace en termes de temps, mais produisant plusieurs contigs au lieu d'un unique *read* synthétique de longueur plus importante, et donc des résultats moins satisfaisants, que nous n'avons cependant pas eu le temps d'étudier en détails. Bien que moins satisfaisants, les résultats obtenus par notre méthode semblent tout de même indiquer que celle-ci constitue un prétraitement efficace à la production de *reads* NaS.

Il serait donc nécessaire, dans un premier temps, de réaliser de nouveaux tests, en ajustant les paramètres de notre méthode, pour tenter d'obtenir de meilleurs résultats. Par

la suite, une analyse en profondeur des résultats fournis pourrait permettre d'en déduire d'avantage d'informations, et ainsi éventuellement rendre possible la production d'uniques contigs, de longueur plus importante, pour chaque *read* long, par exemple en assemblant les multiples contigs actuellement produits, s'ils se chevauchent.

Bien que sortant du cadre du sujet de ce mémoire, il serait également intéressant de dresser l'état de l'art des méthodes d'assemblage de *reads* utilisant une structure d'index sur ces *reads*, afin de réaliser un tour d'horizon complet des solutions aux principaux problèmes concernant les *reads*, et utilisant une structure d'index sur ces derniers.

Bibliographie

[1] N. Philippe, M. Salson, T. Lecroq, M. Leonard, T. Commes, and E. Rivals. Querying large read collections in main memory : a versatile data structure. *BMC bioinformatics*, 12(1) :242, 2011.

[2] J. Schröder, H. Schröder, S. J. Puglisi, R. Sinha, and B. Schmidt. SHREC : A short-read error correction method. *Bioinformatics*, 25(17) :2157–2163, 2009.

[3] P. Weiner. Linear pattern matching algorithms. *Switching and Automata Theory, 1973. SWAT '08. IEEE Conference Record of 14th Annual Symposium on*, pages 1–11, 1973.

[4] L. Salmela. Correction of sequencing errors in a mixed set of reads. *Bioinformatics*, 26(10) :1284–1290, 2010.

[5] L. Ilie, F. Fazayeli, and S. Ilie. HiTEC : Accurate error correction in high-throughput sequencing data. *Bioinformatics*, 27(3) :295–302, 2011.

[6] U. Manber and G. Myers. Suffix Arrays : A New Method for On-line String Searches. *Proceedings of the First Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '90 :319–327, 1990.

[7] M. H. Schulz, D. Weese, M. Holtgrewe, V. Dimitrova, S. Niu, K. Reinert, and H. Richard. Fiona : A parallel and automatic strategy for read error correction. *Bioinformatics*, 30(17) :356–363, 2014.

[8] H. Li, J. Ruan, R. Durbin, H. Li, J. Ruan, and R. Durbin. Mapping short DNA sequencing reads and calling variants using mapping quality scores Mapping short DNA sequencing reads and calling variants using mapping quality scores. *Genome Research*, pages 1851–1858, 2008.

[9] F. Hach, F. Hormozdiari, C. Alkan, F. Hormozdiari, I. Birol, E. E. Eichler, and S. C. Sahinalp. mrsFAST : a cache-oblivious algorithm for short-read mapping. *Nature Methods*, 7(8) :576–577, 2010.

[10] F. Hach, I. Sarrafi, F. Hormozdiari, C. Alkan, E. E. Eichler, and S. C. Sahinalp. MrsFAST-Ultra : A compact, SNP-aware mapper for high performance sequencing applications. *Nucleic Acids Research*, 42(W1) :494–500, 2014.

[11] L. Salmela and J. Schröder. Correcting errors in short reads by multiple alignments. *Bioinformatics*, 27(11) :1455–1461, 2011.

[12] L. Ilie and M. Molnar. RACER : Rapid and accurate correction of errors in reads. *Bioinformatics*, 29(19) :2490–2493, 2013.

[13] Y. Heo, X. L. Wu, D. Chen, J. Ma, and W. M. Hwu. BLESS : Bloom filter-based error correction solution for high-throughput sequencing reads. *Bioinformatics*, 30(10) :1354–1362, 2014.

[14] B. H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7) :422–426, 1970.

[15] N. Välimäki. Scalable and Versatile k -mer Indexing for High-Throughput Sequencing Data. *Proceedings of the 9th International Symposium on Bioinformatics Research and Applications*, pages 237–248, 2013.

[16] T. Kowalski, S. Grabowski, and S. Deorowicz. Indexing arbitrary-length k-mers in sequencing reads. *PLoS ONE*, 10(7) :1–14, 2015.

[17] L. Salmela and E. Rivals. LoRDEC : Accurate and efficient long read error correction. *Bioinformatics*, 30(24) :3506–3514, 2014.

[18] N.G. de Bruijn. A combinatorial problem. *Proceedings of the Section of Sciences of the Koninklijke Nederlandse Akademie van Wetenschappen te Amsterdam*, 49(1946)(7) :758–764, 1946.

[19] R. Li, Y. Li, K. Kristiansen, and J. Wang. SOAP : Short oligonucleotide alignment program. *Bioinformatics*, 24(5) :713–714, 2008.

[20] R. Li, C. Yu, Y. Li, T. W. Lam, S. M. Yiu, K. Kristiansen, and J. Wang. SOAP2 : An improved ultrafast tool for short read alignment. *Bioinformatics*, 25(15) :1966–1967, 2009.

[21] M. Burrows and D. Wheeler. A block-sorting lossless data compression algorithm. *Research report*, 1994.

[22] B. Langmead, C. Trapnell, M. Pop, and S. L. Salzberg. Ultrafast and memory-efficient alignment of short DNA sequences to the human genome. *Genome biology*, 10(3) :R25, 2009.

[23] P. Ferragina and G. Manzini. Opportunistic data structures with applications. *Proceedings of the 41st Annual Symposium on Foundations of Computer Science*, FOCS '00 :390–398, 2000.

[24] B. Langmead and S. L. Salzberg. Fast gapped-read alignment with Bowtie 2. *Nat Methods*, 9(4) :357–359, 2012.

[25] M.-A. Madoui, S. Engelen, C. Cruaud, C. Belser, L. Bertrand, A. Alberti, A. Lemainque, P. Wincker, and J.-M. Aury. Genome assembly using Nanopore-guided long and error-free DNA reads. *BMC Genomics*, 16 :327, 2015.

[26] S. M. Kielbasa, R. Wan, K. Sato, S. M. Kiebasa, P. Horton, and M. C. Frith. Adaptive seeds tame genomic sequence comparison. *Genome Research*, pages 487–493, 2011.

[27] N. Maillet, G. Collet, T. Vannier, D. Lavenier, and P. Peterlongo. Commet : Comparing and combining multiple metagenomic datasets. *Proceedings - 2014 IEEE International Conference on Bioinformatics and Biomedicine*, IEEE BIBM(November) :94–98, 2014.

[28] G. Holley, R. Wittler, and J. Stoye. Bloom Filter Trie – A Data Structure for Pan-Genome Storage. *Algorithms in Bioinformatics : 15th International Workshop, WABI 2015*, pages 217–230, 2015.

[29] H. Li and R. Durbin. Fast and accurate short read alignment with Burrows-Wheeler transform. *Bioinformatics*, 25(14) :1754–1760, 2009.

[30] T. Hackl, R. Hedrich, J. Schultz, and F. Förster. Proovread : Large-scale high-accuracy PacBio correction through iterative short read consensus. *Bioinformatics*, 30(21) :3004–3011, 2014.

[31] W. J. Kent. BLAT — The BLAST -Like Alignment Tool. *Genome research*, 12 :656–664, 2002.