Étude des différentes approches d'indexation : du génome au pan-génome Application aux génomes de riz

Clément Agret

THÈSE POUR OBTENIR LE GRADE DE DOCTEUR DE MONTPELLIER SUPAGRO

En Génétique et Amélioration des plantes

École doctorale : GAIA - Biodiversité, Agriculture, Alimentation, Environnement, Terre, Eau

Unité de recherche UMR AGAP - Cirad & MAB - Lirmm

Étude des différentes approches d'indexation : du génome au pan-génome Application aux génomes de riz

Présentée par Clément AGRET

Le 13 octobre 2020

Sous la direction de Manuel RUIZ, Annie CHATEAU et Alban MANCHERON

Devant le jury composé de

Dr. Thérèse COMMES	Professeur à l'Université de Montpellier	**Présidente du jury**
Dr. Thierry LECROQ	Professeur à l'Université de Rouen Normandie	Rapporteur
Dr. Jacques VAN HELDEN	Professeur à l'Université d'Aix-Marseille	Rapporteur
Dr. Mikaël SALSON	Maître de conférences à l'Université de Lille	Examinateur
Dr. Hélène CHIAPELLO	Ingénieur de recherche à l'INRAE	Examinatrice
Dr. Manuel RUIZ	Chercheur au Cirad	Co-directeur
Dr. Annie CHATEAU	Maître de conférences à l'Université de Montpellier	Co-directrice
Dr. Alban MANCHERON	Maître de conférences à l'Université de Montpellier	Co-encadrant
Dr. François SABOT	Directeur de recherche à l'IRD	Invité

UNIVERSITÉ DE MONTPELLIER

Montpellier SupAgro

Étude des différentes approches d'indexation : du génome au pan-génome

Application aux génomes de riz

Different indexing approaches study: From Genome to Pan-genome

Application to rice genome

Clément Agret

Université de Montpellier

Clément Agret

Étude des différentes approches d'indexation : du génome au pan-génome
Application aux génomes de riz

174 (IV+xvi+154) pages

Ce document a été préparé avec LaTeX2e et le package `hdrUM` version (None). Le package `hdrUM` est disponible à l'adresse : `https://gitlab.info-ufr.univ-montp2.fr/doccy/latex-hdrUM/`.

Impression : output.pdf – 26/10/2020 – 08:49.

Remerciements

À mon neveu Augustin, Cédric, Clément AGRET et ses cousins à venir.

Avant toute chose, je souhaite remercier les familles AGRET et AUDEMARD de m'avoir conçu, élevé et considéré mais également pour la confiance, l'amour et l'attention qu'ils ont su m'apporter. Je suis chanceux d'avoir une si belle famille.
Je remercie mes très chers parents, Bernard et Véronique, qui ont toujours été là pour moi. Merci à mon deuxième papa, Bertrand qui m'a élevé comme son fils. Je remercie mon unique frère Florian d'avoir été et d'être une source d'inspiration, et mes sœurs Léa, Clara et Anyliane pour leur amour, soutien et encouragements inconditionnels. Je n'oublie pas ma belle sœur Dr. Julie qui a toujours été là pour moi et qui a mis au monde (objectivement et à cette date) le plus beau bébé du monde. Sans oublier Michèle, plus connue sous le nom de Mamiette, qui a toujours su subvenir à mes besoins en me préparant de bons petits plats en quantité suffisante (autrement dit pour 10).
Un grand merci à mon parrain Jean-Christophe, dit Christophe, et à ses abeilles pour leur bon miel, à mon oncle Bruno d'avoir toujours été à l'écoute très attentive et curieux de ce que je pouvais bien lui raconter, merci à ma tante tata-caco pour son énergie folle et les goûters d'anniversaire de ses filles débordant de bonnes choses, merci à mamie, papi, Domi et tonton Michel pour tous les bons moments vécus ensemble. Merci Isa de m'avoir expliqué les mathématiques, plus précisément comment résoudre l'équation $ax^2 + bx + c = 0$.
Un très grand merci à ma grande famille.

Je suis conscient que la thèse est un travail d'équipe et de collaboration. Je n'aurais jamais pu réaliser ce travail ainsi que cette formation doctorale sans le soutien d'un grand nombre de personnes. Ces personnes dont la générosité, la bonne humeur et l'intérêt manifestés à mon égard m'ont permis de progresser dans cette phase délicate de passage d'étudiant à jeune chercheur (ou de jeune Padawan à Jedi).

Dans ce périple, les acteurs ont été multiples et j'espère n'avoir oublié personne (comme j'en ai la fâcheuse habitude). Sachez par avance qu'il n'y a personne que je ne souhaite pas remercier, sauf peut-être toutes les personnes motorisées qui roulent sur les pistes cyclables et qui doublent d'un peu trop près les cyclistes. Sachez également que si le directeur veut retenir l'élite en baisse, il risque de transformer délires en peines.

En parlant de directeur, je tiens à remercier en premier lieu Docteur Manuel RUIZ, Docteur Annie CHATEAU et Docteur Alban MANCHERON.
Manuel pour la confiance qu'il m'a accordée, pour toutes les heures consacrées à diriger ma recherche et pour ces années à essayer de faire de moi un vrai chercheur. Merci Manuel tu m'as beaucoup apporté et appris, tu as beaucoup compté pour moi et je suis fier de voir le chemin que l'on a accompli.
Annie pour sa patience, sa patience, sa patience et surtout sa patience. Merci Annie pour ta diplomatie, ton temps et d'avoir su comment tempérer Alban qui pour moi restera un mystère.
Alban pour sa grande, immense, que dis-je gigantesque disponibilité, pour ses innombrables conseils, ses centaines de milliers de lignes de code de corrections et son respect à toute épreuve des délais serrés de relecture des documents que je lui ai adressés. Je pense pouvoir écrire une thèse sur tout ce que Alban m'a apporté (humainement et scientifiquement). Merci Alban.
J'aimerais également dire à quel point j'ai apprécié leurs qualités humaines d'écoute et de compréhension tout au long de ce travail doctoral. J'ai réussi et c'est en grande partie grâce

à eux trois.

Merci à mes rapporteurs Pr. Thierry Lecroq et Pr. Jacques Van Helden d'avoir accepté de relire mon manuscrit et de m'avoir fait un retour très constructif sur cette relecture. Merci à l'ensemble des membres du jury d'avoir été présents pour ma soutenance et d'avoir permis une discussion bienveillante et très intéressante.

Merci à mon cher voisin Dr André M. pour ses relectures et nombreux conseils. Merci JLP d'être là depuis le début, tu comptes beaucoup pour moi. Merci Dr. C. Bessiere père et fille pour m'avoir mis sur le chemin de la recherche en informatique. Merci Éric ainsi qu'à tous les québécois (Diane et Yves si vous me lisez). Un merci ainsi qu'un bonjour à l'équipe ID Salut Fred, Maryline, Jean-François, Gautier et Gaétan. Un grand merci à Dr. Marion D. et Dr. Aurélien C. d'avoir été mes acolytes, mais également à tous les doctorants d'AGAP, merci Clara, Aurélie, Kelly, Lauriane et Benjamin d'être toujours partants pour aller boire un verre.
Merci au Cirad et au Lirmm de m'avoir accueilli chaleureusement, merci à l'UMR AGAP et à l'équipe MAB pour les pauses cafés et bières.

Un immense merci à Marc Chakiachvili, alias l'homme le plus classe du monde ainsi qu'à sa femme et ses merveilleux enfants, d'avoir les même références que moi et de me permettre de me sentir moins seul dans ce monde de merde.
Merci à Dr. Jérôme A., Cerise B., Étienne B., Dr. Marianne B., Robin B., et Dr. David F. d'être mes amis et d'être toujours là même après X années (l'ordre est alphabétique). Je ne compte même plus les vacances passées ensemble, les longues soirées à refaire le monde et tout simplement tous les moments où nous étions réunis. Merci Balgorr, Boub et Zooby d'être mes témoins si jamais je devais me marier.
Merci à mes amis Beret et Maz qui, en plus d'être des moteurs sportif (Tous les semi-marathons, marathons et triathlon ça compte), sont mes camarades de beuverie. Merci à la Tennis United, aux 11 présents en Chine et un merci tout particulier à Mario qui lira mes remerciements à Téo (big louche).
Merci Julie, Claire et MAB, d'avoir accepté de me relire et d'avoir corrigé les quelques milliers de fautes que mon clavier avait laissé paraître!!

Je ne remercierai jamais assez le tord-boyaux, *Hearthstone* et *League of Legends* pour son soutien affectif sans faille dans la Faille de l'Invocateur.
Merci également à Orelsan, Lorage, Lomepal, Dooz Kawa, Hugo TSR, Scylla, Nekfeu, l'ODC, Melan et Gaël Faye de m'avoir procuré de bons sons, morceaux ou de bons concerts permettant un échappatoire certain.

Merci au Palmashow, la Classe Américaine, Dikkenek, La Cité de la peur, Oss117 1 et 2, Astérix et Obélix mission Cléopatre, la stratégie de l'echec, Waynes world, Dumb et dumber et la multitude de films nuls ou des nuls qui m'ont forgé mon sens de l'humour absurde.
Merci à René Coty d'avoir été mon porte bonheur et permis de rencontrer ma conjointe.
Si tu me lis, merci Michel Hazanavicius, j'ai adoré voir, revoir, rerevoir et même (re)^{n}voir et lire la Classe américaine.

Pour finir, merci Claire pour ton soutien dans ces derniers moments dits « de rush ultime ». Merci d'avoir su et pu être mon pilier et la partenaire idéale pour la rédaction de ce manuscrit, la préparation de la présentation et les démarches administratives!!!
Merci Claire d'avoir été celle qui a déposé mon dossier sans lequel je n'aurais jamais pu

soutenir et défendre cette thèse.

Il n'y a pas que le travail qui mène au titre de docteur, il y a également la passion et l'amour de ce qu'on fait. Si j'ai toujours réussi à finir ce que j'ai entrepris c'est uniquement grâce à vous tous.

Les paysages sont désolés, pourtant les hommes n'ont pas d'excuses.
Ken Samaras - Nekfeu

La connaissance s'acquiert par l'expérience, tout le reste n'est que de l'information.
Albert Einstein

Sommaire

— Pages liminaires —

— Corps du document —

— Annexes —

— Pages annexées —

Introduction

L'évolution rapide des technologies de séquençage permettant d'accéder aux informations biologiques contenues dans l'ADN a révolutionné le monde de la biologie. Les nouvelles technologies dites NGS (Next-Generation Sequencing) produisent aujourd'hui la séquence d'un génome humain, d'une taille moyenne d'environ $3 \cdot 10^9$ paires de bases, pour moins de 1000€.
Ces faibles coûts, couplés à l'évolution des techniques d'assemblage de génomes, permettent l'établissement d'un nombre croissant de génomes de référence pour différentes espèces. De nombreux projets de comparaison de ces génomes ont donc vu le jour au cours de ces dernières années.
Ces génomes de référence sont utilisés comme un guide pour étudier les génomes d'une population, cartographier les séquences de génomes nouvellement séquencés pour une espèce donnée, identifier et annoter les différentes régions du génome.
La disponibilité de génomes de référence pour différentes espèces n'est qu'un premier pas vers la compréhension des relations entre le patrimoine héréditaire d'un individu, correspondant à son *génotype*, et l'ensemble des caractères visibles de cet individu, correspondant à son *phénotype*. Des études vont ainsi permettre de comprendre les rôles et les fonctions de différents gènes dans certaines maladies.

Pour pouvoir étudier efficacement la question de l'indexation des données génomiques, les équipes ID (Intégration des Données) de l'UMR AGAP, et MAB (Méthodes et Algorithmes pour la Bioinformatique) du LIRMM se sont associées pour définir et co-encadrer cette thèse.

L'unité de recherche AGAP, Amélioration Génétique et Adaptation des Plantes méditerranéennes et tropicales, est une unité mixte de recherche placée sous la tutelle du Cirad, l'Inrae et Montpellier SupAgro. Cette unité développe une recherche scientifique en biologie des plantes et génétique végétale sur un grand nombre de plantes d'intérêt agronomique. Une partie importante de mon travail de thèse s'est notamment déroulée au Cirad, le Centre de coopération internationale en recherche agronomique pour le développement qui est l'organisme français de recherche agronomique et de coopération internationale pour le développement durable des régions tropicales et méditerranéennes. À ce titre, il met son expertise scientifique et institutionnelle au service des politiques publiques de ces pays et des débats internationaux sur les grands enjeux de l'agriculture. Il apporte son soutien à la diplomatie scientifique de la France.
Le Laboratoire d'Informatique, de Robotique et de Microélectronique de Montpellier (LIRMM) est une unité mixte de recherche, dépendant conjointement de l'Université de Montpellier et du Centre National de la Recherche Scientifique (CNRS). Ses activités de recherche sont centrées sur le domaine des sciences et technologies de l'information, de la communication et des systèmes.

Ce travail s'inscrit donc dans un contexte transdisciplinaire, le cas échéant traversant les frontières entre les disciplines biologique, informatique et bioinformatique.

Ce travail d'indexation d'une large collection de génomes similaires s'intègre dans le cadre

d'un projet plus large, le projet étendard *GenomeHarvest*. Ce dernier, financé par Agropolis Fondation (2016-2020), regroupe de nombreuses équipes de la communauté scientifique d'Agropolis. Il repose sur l'acquisition récente de nombreuses séquences de génomes de référence [D'Hont et al., 2012, Wu et al., 2014, Garsmeur et al., 2018].
Ces données disponibles couplées à la possibilité d'obtention de données de reséquençage massives ouvrent la voie pour une meilleure compréhension de l'organisation et de la dynamique des génomes et donne les premières clés pour une exploitation plus efficace de leur diversité dans les programmes de sélection variétale.
Cette nouvelle perspective soulève également de nouveaux défis pour une exploitation optimale de ces données, notamment le développement de nouveaux concepts, méthodes et outils de biomathématique/bioinformatique.

Le projet *GenomeHarvest* se concentre d'une part, sur les aspects liés aux évènements inter(sous)spécifiques les plus fréquents impliqués dans l'histoire des plantes cultivées et, d'autre part, sur le développement de méthodologies pour décrypter la structure inter(sous)spécifique des génomes de plantes et son impact sur la transmission de caractères. Ce projet *GenomeHarvest* aborde une approche pluridisciplinaire intégrant biomathématiques, bioinformatique, génomique, et génétique pour le développement de méthodologies exploitant les données issues des séquenceurs à haut débit, dites NGS [1].
L'étroite collaboration au sein de ce projet a pour objectif de caractériser principalement trois points : premièrement, les génomes issus d'hybridations inter(sous)spécifiques ; deuxièmement, les variations structurales de grande taille ; troisièmement, l'impact des structures des génomes sur la ségrégation chromosomique et sur l'expression des gènes.

Les modèles biologiques choisis pour mener à bien cette étude et cette collaboration sont le bananier, le caféier, les agrumes et le riz. Ces modèles de culture ont été choisis car ils couvrent une large gamme d'événements inter(sous)spécifiques avec un nombre d'événements faible pour les agrumes jusqu'à un nombre très élevé pour le riz, qui ont conduit à des structures mosaïques pour des génomes de taille très variables. La structure mosaïque du génome d'un individu va donc correspondre à la combinaison non uniforme du matériel génétique de ses différents ancêtres, après différents évènements d'hybridations inter(sous)spécifiques.

En ce qui concerne le riz, ces dernières années, le nombre de génomes séquencés a augmenté de manière considérable. Ainsi, l'International Rice Research Institute (IRRI), dans le cadre du consortium GRISP (Global Rice Science Partnership), a initié un programme de séquençage de l'ensemble des variétés de riz, et aujourd'hui plus de 3000 génomes sont déjà disponibles [Li et al., 2014]. Dans le cadre du projet *GenomeHarvest*, nous cherchons à prendre en compte de manière systématique les micro-réarrangements du génome entre les différentes variétés de riz car ceci est essentiel pour pouvoir projeter les résultats de séquençage des variétés dans différentes analyses : cartographie génétique haute résolution, GWAS (Genome-wide association study), sélection génomique, analyse de diversité, etc.

Pour cela, nous avons besoin d'organiser de manière optimisée en matière d'espace de stockage et de rapidité de requête, les zones communes à tous les génomes et l'information sur les variations structurales elles-mêmes. Il est donc nécessaire d'avoir une structure de données qui permette l'indexation d'une large collection de génomes similaires, à l'échelle des 3000 génomes du riz [2].

1. NGS pour next-generation sequencing regroupe un ensemble de méthodes permettant de séquencer des millions de fragments simultanément.

2. Un génome de riz occupe environ 500 mégaoctets d'espace disque donc 3000 génomes occupent environ 1,5 téraoctets d'espace disque.

En outre, ce type de structure de données pourrait être utilisée pour des analyses de type pangénomique. En effet, un pan-génome peut-être défini comme un ensemble de séquences génomiques devant être analysées ou servir de référence conjointement, et qui intègre l'information sur les variations structurales génomiques entre les différentes variétés, groupe d'individus ou individus d'une même espèce.

Les pan-génomes permettent de visualiser des ensembles de génomes, en regroupant pour la majorité les points communs et en n'affichant que les différences entre eux, telles que les variations de nucléotide unique (SNV) ou les variations structurelles (SV). Comme de nouveaux génomes sont ajoutés au pan-génome, le nombre et la complexité de ces variations augmente rapidement. La détection automatique de variations fournirait une grande quantité d'informations sur la variation au sein d'une population et éclairerait la relation entre ces variations et un certain phénotype.

La stratégie qui semble la plus évidente est qu'au lieu d'assembler le génome séquencé et le comparer avec la séquence de référence, on assemble le génome de différents individus de la même espèce et on compare les séquences entre elles.

Les défis majeurs de cette approche sont les difficultés à produire des assemblages de haute qualité auquel s'ajoute la variation des résultats de la prédiction des gènes.

Aujourd'hui, des assemblages de génomes peuvent être produits, même pour des espèces complexes, relativement rapidement et à faible coût. Cependant, ces assemblages reflètent rarement la teneur totale en gènes comme pour les régions répétitives qu'on a tendance à perdre pendant le processus d'assemblage, ne laissant qu'un sous-ensemble des gènes.
Parfois les différences observées sont dues à des erreurs d'assemblage et non à des raisons biologiques. Le processus d'annotation consiste à assigner à chaque séquence une étiquette indiquant sa nature et sa fonctionnalité. Ce processus peut aussi introduire des erreurs dans les analyses du pan-génome ; ceci peut être observé lorsqu'un gène semble manquer dans le contenu du génome annoté chez un individu alors que la comparaison de séquences de génomes montre qu'il est présent mais qu'il n'est juste pas annoté.

Ce travail de thèse porte sur l'établissement d'une méthode et l'implémentation d'un outil pour répondre au problème de l'indexation d'une collection de génomes.

Première partie

Dans cette partie, après une brève introduction au contexte scientifique de cette thèse, seront présentés le vocabulaire et les concepts essentiels à la compréhension de la problématique traitée (chapitre 1). Dans le chapitre 2, nous aborderons l'aspect méthodologique des approches par k-mers appliquées à la génomique, et notamment l'existant en matière d'outils de comptage et d'indexation de k-mers, potentiellement applicables à des collections de génomes. Enfin, le chapitre 3 est consacré à la pan-génomique, ses concepts et ses représentations de collections de génomes à l'aide de structures d'indexations.

Chapitre 1 Dans ce chapitre, l'objectif est de présenter le contexte biologique et informatique de ma thèse. Sont introduits partiellement les connaissances relatives au riz, la génomique, et la bioinformatique. Une majeure partie de ce chapitre définit le vocabulaire utilisé dans ce manuscrit.

Chapitre 2 Il est de plus en plus facile d'obtenir des génomes complets, ou au moins un ensemble de séquences représentant des génomes entiers, grâce à l'utilisation intensive du

séquençage haut débit, donc de nouveaux besoins émergent, et de nouveaux outils voient le jour. Dans ce chapitre nous introduirons ces outils et les technologies en émergence. Nous introduirons également des concepts plus anciens mais toujours d'actualité comme les structures succinctes.
Ce deuxième chapitre est consacré à la pan-génomique et aux méthodes existantes pour l'indexation de données biologiques. Il révèle la difficulté de penser et de concevoir des algorithmes pour répondre aux questions biologiques posées étant donnée l'immensité des données relatives à ce domaine.

Chapitre 3 Dans ce chapitre, nous aborderons comment répondre aux défis qui émergent d'une nouvelle ère génomique, centrée sur l'analyse de gènes et de séquences spécifiques, mais aussi sur des études manipulant de dix à plusieurs milliers de génomes par espèce. Une telle collection est généralement appelée un pan-génome. Dans le pan-génome, de grandes parties du génome sont partagées entre les individus.
Partant de cette idée, cette troisième partie présente les structures de données existantes pour indexer une collection de génomes similaires. Une approche sans référence ni alignement évite la perte d'informations sur la variation génétique qui n'aurait pas été trouvée dans le cas de l'alignement direct des lectures des séquences sur un génome de référence.

Seconde partie

Dans cette partie, seront présentés nos études préliminaires et les résultats obtenus. Dans le chapitre 5, nous aborderons l'aspect méthodologique et logiciel des approches par comptage de k-mers appliquées à la génomique. Nous introduirons la librairie *gkampi* et sa stratégie de comptage et d'indexation pour un génome. Dans le chapitre 6 nous présenterons notre contribution majeure : *RedOak*, un logiciel d'indexation de k-mers pensé pour être appliqué à de grandes collections de génomes. Enfin, le chapitre 7 est consacré aux résultats obtenus et à leur validation biologique.

Chapitre 4 Dans ce chapitre est introduit le cheminement qui nous a mené à la structure d'indexation que nous avons implémentée. Dans un premier temps, nous rappellerons les besoins qui nous ont poussé à chercher dans cette direction, puis les solutions déjà existantes ainsi que celles que nous avons envisagées pour pallier aux problèmes que nous avons rencontrés.
Pour ce faire nous présenterons dans un premier temps l'étude qui a permis l'analyse combinatoire des 8 premiers génomes de riz assemblés et disponibles en 2016.
Cette étude préliminaire nous mène sur la deuxième partie de ce chapitre qui porte sur notre premier outil *LEGO* (pour whoLE Genomes cOmparaison), basé sur les comptages de k-mers et qui permet de créer des diagrammes d'ensembles. Nous aborderons ensuite *PICTURES*, un outil que nous avons développé en C, qui à partir d'une matrice de présence/absence de k-mers, permet soit de créer une image compressible soit de créer des digrammes de Venn.
Nous verrons finalement comment nous avons imaginé un algorithme permettant de compresser des données très éparses ce qui a donné lieu à *RUBIKS* (pour RRR Update for Bit Indexing k-mer Structure).

Chapitre 5 Nous présentons dans ce chapitre un algorithme d'indexation de k-mers issus de séquences biologiques, massivement parallèle, et son implémentation en C++ sous forme d'une librairie : `libGkArrays-MPI`. Nous présenterons ensuite un premier logiciel basé sur cette librairie permettant d'indexer et de faire des requêtes sur des séquences biologiques : *gkampi*.

Les performances de *gkampi* seront présentées dans le contexte de son utilisation sur un cluster de calcul comme sur un ordinateur portable.

Chapitre 6 Dans ce chapitre nous décrivons *RedOak*, un outil également basé sur le `libGkArrays-MPI` permettant d'indexer et d'interroger les k-mers issus d'une collection de génomes. Ce chapitre se découpe en quatre parties, nous décrivons dans la première notre approche. Dans la seconde est décrit l'algorithme principal de *RedOak*, notre structure d'indexation. La troisième partie présente les requêtes que notre structure autorise. Pour finir, la dernière partie traite de la complexité en temps et en espace de notre structure d'indexation.

Chapitre 7 Dans ce chapitre sont présentées les performances du logiciel *RedOak*, décrit dans le chapitre 6. Une comparaison sera faite sur le temps et l'espace nécessaire pour l'indexation d'une large collections de génomes ainsi que sur le temps nécessaire pour effectuer un certain nombre de requêtes sur un exemple biologique concret. Dans la dernière partie de ce chapitre, nous montrons la comparaison des résultats obtenus avec *RedOak* à ceux d'autres outils.

Conclusion et discussions

Cette dernière partie présente une synthèse des travaux détaillés dans le manuscrit. De cette synthèse seront présentées plusieurs perspectives liées au travail effectué au cours de cette thèse à moyen et long terme.

Première partie

État de l'art

Dans cette partie, après une brève introduction au contexte scientifique de la thèse, seront présentés le vocabulaire et les concepts essentiels à la compréhension de la problématique de la thèse (chapitre 1). Dans le chapitre 2, nous aborderons l'aspect méthodologique des approches par k-mers appliquées à la génomique, et notamment l'existant en matière d'outils de comptage et d'indexation de k-mers, potentiellement applicables à des collections de génomes. Enfin, le chapitre 3 est consacré à la pan-génomique, ses concepts et ses représentations de collections de génomes à l'aide de structures d'indexations, en lumière avec les besoins de la pan-génomique.

Chapitre 1

Contexte

Dans ce premier chapitre, le contexte scientifique de ce manuscrit de thèse est décrit. Il s'agit ici de présenter les notions essentielles de la génomique permettant la compréhension du document.
Nous aborderons dans un premier temps la génomique et dans un second temps, la terminologie ainsi que toutes les notions associées à la bioinformatique utilisés dans ce manuscrit.

1.1 La génomique

La *génomique* est la science qui a pour objet l'étude des génomes. Elle vise à comprendre comment ils se sont formés, se sont maintenus, et quelles sont leurs fonctions dans le cadre d'un organisme donné, d'une maladie, ou d'un événement biologique.

La génomique peut se diviser en deux parties :

i/ La *génomique structurale*, qui met en lien les données issues du séquençage, de l'organisation, et de la structure physique du génome, à différentes échelles : ensemble du génome, chromosomes (translocations et autres remaniements), régions particulières (suppressions, duplications, etc.), ensemble de gènes, etc.

ii/ La *génomique fonctionnelle*, qui vise à déterminer la fonction et l'expression des gènes. C'est la caractérisation de l'expression du génome et l'intégration de cette expression dans l'élaboration des grandes fonctions métaboliques qui est étudiée.

Pour présenter la génomique nous aurons besoin des notions suivantes : ADN, gène, génome. Nous introduirons également notre organisme modèle : le riz.

1.1.1 L'ADN

L'acide désoxyribonucléique qui est plus communément appelé ADN est une macromolécule qui sert de support pour stocker le génome de toutes les cellules et certain virus. L'ADN est constitué d'acides nucléiques (*i.e.*, nucléotides). Il existe quatre bases nucléiques qui se séparent en deux catégories, les *purines* et les *pyrimidines*. Les purines regroupent l'Adénine et la Guanine et les pyrimidines regroupent la Cytosine et la Thymine. Ces quatre bases sont nommées selon leur initiale (respectivement A, G, C et T) depuis la norme IUPAC (International Union of Pure and Applied Chemistry) [IUPAC IUBMB JCMB, 1970].

Il existe des liaisons entre ces molécules. L'Adénine et la Thymine s'apparient au moyen de deux liaisons hydrogène, tandis que la Guanine et la Cytosine s'apparient au moyen de trois liaisons hydrogène (voir figure 1.1). Les bases qui s'apparient sont dites complémentaires. Lorsque les séquences des deux brins sont complémentaires, ces brins peuvent s'apparier en formant une structure bicaténaire hélicoïdale caractéristique que l'on appelle double hélice d'ADN. Cette double hélice est adaptée au stockage de l'information génétique. Elle s'organise en *Chromosomes* dont la composition, le nombre et la structure sont des caractéristiques des espèces.

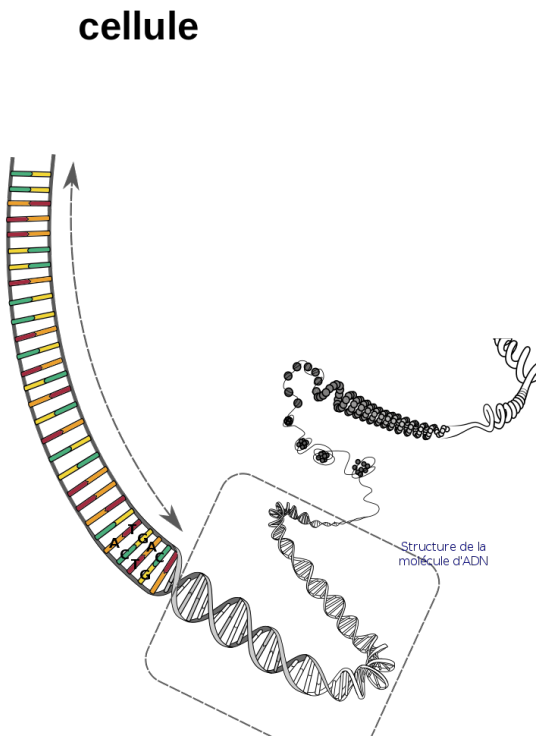


Figure 1.1: Schéma d'appariement des bases de l'ADN qui vont s'enrouler sur des histones pour former des nucléosomes et donner naissance aux chromosomes dans la cellule eucaryote

Source: National Human Genome Research Institute

1.1.2 Le gène

En 1909, Wilhelm Johannsen définit le *gène* comme l'unité de base de l'hérédité, qui détermine un trait précis pour un organisme vivant. Nous retiendrons ici qu'un gène est une unité d'information génétique.

Sur la molécule d'ADN, un gène est représenté par une séquence de nucléotides, caractérisé à la fois par sa position et par l'ordre de ses bases azotées. Il s'agit d'un langage codé en séquence. On dit ainsi que l'ADN est le support de l'information génétique.
Les chromosomes sont transmis de la cellule mère aux cellules filles par un procédé appelé la *mitose*. Cette action permet une duplication fidèle du matériel génétique. L'ensemble du matériel génétique d'une espèce constitue son génome.

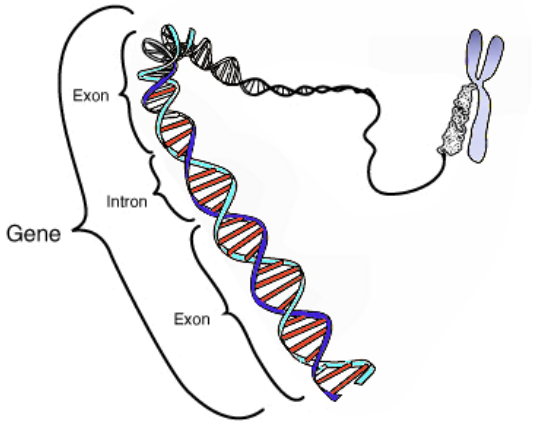


Figure 1.2: Illustration d'un enchaînement exon-intron dans un gène.

Source: National Human Genome Research Institute

1.1.3 Le génome

Le *génome*, est l'ensemble du matériel génétique d'une espèce codé dans son acide désoxyribonucléique (ADN) ; il peut pour certaines espèces, comme c'est le cas de certains virus, être constitué d'acide ribonucléique (ARN).

Il contient en particulier tous les gènes codants pour des protéines ou correspondant à des ARN structurés (gènes non codants). Il se décompose donc en séquences codantes (transcrites en ARN messagers et traduites en protéines) et non codantes (non transcrites, ou transcrites en ARN, mais non traduites).

Le génome est constitué d'un ou plusieurs *chromosomes* dont le nombre total dépend de l'espèce considérée, chaque chromosome étant constitué d'une unique molécule d'ADN, linéaire chez les eucaryotes, et le plus souvent circulaire chez les procaryotes.

Chaque chromosome peut être présent en un ou plusieurs exemplaires. Pour les organismes diploïdes (organismes dont les chromosomes des cellules sont présents par paires), un exemplaire provient de la mère et l'autre du père. Plus généralement, les génomes polyploïdes sont des génomes possédant plusieurs copies d'un même gène.
Le caryotype est l'arrangement des chromosomes d'une cellule.

La taille du génome se mesure en nombre de nucléotides, ou bases. Dans ce manuscrit, nous utiliserons la notation *pb* (pour paire de bases) parce que les génomes étudiés seront constitués de doubles brins d'ADN.
Nous utilisons souvent les multiples *Kpb* (pour kilo-paire-de-bases, abusivement abrégé en kilobase), *Mpb* (mégabase) ou *Gpb* (gigabase), qui valent respectivement 1 000, 1 000 000 et 1 000 000 000 [paires de] bases. La taille du génome est variable. Elle s'étend de quelques kilobases chez les virus à plusieurs centaines de *Gpb* chez certains eucaryotes. La quantité d'ADN n'est pas proportionnelle à la complexité apparente d'un organisme. Une notion

importante est *la valeur C*, qui représente la taille d'un génome, exprimée en paire de bases, ou en picogrammes. Cette valeur C peut être mesurée pour chaque espèce. En 1971, C.Thomas à regroupé les valeurs de C issues de différentes espèces et en a conclu que la valeur de C n'est pas significative du niveau de la complexité de l'organisme considéré ; c'est le paradoxe de la valeur C [Thomas, 1971].

1.2 La bioinformatique

La définition exacte du terme *bioinformatique* reste toujours une source de discussion au sein de la communauté scientifique.

Deux définitions peuvent être relevées : la première consiste à concevoir la bioinformatique comme une science. La seconde associe cette discipline à un ensemble d'outils mis à la disposition des biologistes pour valider des expériences biologiques.

Les défenseurs de la première approche considèrent que l'informatique bouleverse fondamentalement la recherche en biologie moléculaire. Elle permet d'aller beaucoup plus loin dans l'exploration du monde biologique, en apportant un nouveau paradigme de recherche défini par la conception de modèles mathématiques sur lesquels peuvent se mener des expériences *in silico*. Ce néologisme a été créé par analogie avec le terme latin *in vivo* pour désigner des simulations numériques.

La deuxième approche consiste à concevoir l'informatique comme un outil d'analyse de données adapté aux besoins des biologistes. Cette discipline conçoit et développe des méthodes et des logiciels pour le stockage et le traitement de données biologiques.

1.2.1 La bioinformatique et les séquences biologiques

Le flot récent de données correspondant aux séquences issues du séquençage du génome et de la génomique fonctionnelle a donné naissance à un nouveau domaine, la bioinformatique, qui combine des éléments de la science du vivant et de l'informatique [Luscombe et al., 2001].

La bioinformatique conceptualise la biologie en termes de macromolécules (au sens physico-chimique) puis applique des techniques informatiques qui sont directement dérivées de disciplines scientifiques. Ces disciplines telles que les mathématiques appliquées, l'informatique théorique et les statistiques permettent de comprendre et d'organiser à grande échelle les informations associées à ces molécules.

Les analyses en bioinformatique se concentrent principalement sur trois types de grands ensembles de données disponibles en biologie moléculaire : les structures macromoléculaires, les séquences du génome et les résultats des expériences de génomique fonctionnelle (par exemple les données d'expression).

Des informations supplémentaires incluent le texte des articles scientifiques et des données sur les relations provenant des voies métaboliques, des arbres de taxonomie et des réseaux d'interactions protéine-protéine.

Nous enquêtons sur certaines applications représentatives, telles que la recherche de séquences homologues [1], la conception de médicaments et la réalisation de recensements à grande échelle [Koonin, 2005].

1. Deux séquences sont dites homologues lorsqu'elles partagent une origine ancestrale commune

La bioinformatique est une science jeune qui n'est pas l'intersection de la biologie et de l'informatique, qui n'est pas l'informatique au service de la biologie mais qui englobe en partie la biologie et l'informatique et qui a ses méthodes bien à elle. Elle joue un rôle prépondérant avec la conception d'outils de stockage et de calcul d'un grand nombre de données. Ces méthodes vont de l'analyse génétique à l'analyse d'images en passant par la génomique comparative avec l'alignement de séquences, par la modélisation de l'évolution d'une population dans un environnement, par la modélisation moléculaire, par la transcriptomique [2], par le stockage et l'indexation des séquences génomiques et la reconstruction d'arbres phylogénétiques, etc.

La bioinformatique utilise un large éventail de techniques de calcul, notamment l'alignement de séquences et de structures, la conception de bases de données et l'exploration de données, la géométrie macromoléculaire, la construction d'arbres phylogénétiques, la prédiction de la structure et de la fonction des protéines, la recherche de gènes et le regroupement de données d'expression. L'accent est mis sur les approches intégrant une variété de méthodes de calcul et des sources de données hétérogènes.

Toutes ces méthodes ont apporté et vont encore apporter des avancées considérables à la connaissance des organismes vivants.

1.2.2 Les notations et définitions relatives aux séquences

L'analyse des séquences biologiques est souvent réalisée *in vitro*, par les chercheurs en biologie. Cependant, depuis le séquençage de l'ADN et surtout depuis la naissance des séquenceurs haut débit (on parle alors de NGS, acronyme de *Next-generation sequencing*), il est de plus en plus fait appel à une analyse *in silico*, c'est-à-dire, avec des moyens informatiques. Lorsque l'on séquence le génome d'un individu d'une espèce, on obtient un très grand nombre de petits fragments de génome appelés *reads* ; il s'agit d'une quantité énorme de pièces de puzzle que l'on essaie de reconstituer. En général, et lorsque c'est possible, on se sert du génome de référence comme support pour arriver à assembler ces pièces de la façon la plus correcte possible. Il arrive que parfois on trouve des variations entre les pièces dont on dispose et la zone du génome de référence correspondante ; certaines variations peuvent avoir une explication biologique d'autres non, et on se retrouve aussi quelques fois avec des pièces de puzzle manquantes et d'autres en trop. Les quantités de données générées sont telles que le seul moyen de les traiter est d'utiliser des outils informatiques [Luscombe et al., 2001].

Prenons, par exemple, le cas du coronavirus SARS-CoV-2 (responsable de la maladie COVID-19) : un nombre exponentiel de séquences de génomes est généré, dont l'annotation reste à effectuer. C'est typiquement à ce niveau que la bioinformatique a un intérêt majeur en permettant une analyse *in silico* des données à traiter. La bioinformatique aborde les études sous un aspect plus formel : ainsi certaines théories développées dans le cadre de l'algorithmique du texte trouvent une application dans le traitement de ces séquences. En outre, il ne s'agit pas simplement d'appliquer des théories aux problèmes biologiques, mais de les adapter, d'en développer de nouvelles plus spécifiques.

Dans le cadre du séquençage à haut débit, le modèle que nous allons étudier est le riz. La première difficulté consiste à organiser une masse considérable d'informations et à la rendre facilement accessible à l'ensemble de la communauté des chercheurs. La deuxième difficulté consiste à concevoir et développer des outils d'analyse capables de gérer toute cette quantité de données.

2. La transcriptomique est l'étude, par la quantification, de l'ensemble des molécules produites lors du processus de transcription d'un génome

Pour répondre à un besoin de traitements d'un nombre considérable de données, des méthodes bioinformatiques basées sur des résultats issus de l'algorithmique du texte ont été développées. Ces méthodes étant formelles, nous avons besoin de définir des notions élémentaires d'algorithmique du texte qui seront utilisées tout au long de ce manuscrit. En effet, les séquences biologiques sont des portions d'ADN que nous considérerons comme du texte. La première définition que nous allons introduire est la notion d'alphabet.

Définition 1.1. Un *alphabet* Σ est un ensemble fini de symboles. Le cardinal σ de cet ensemble est le nombre de symboles qui le composent.

Dans le cas de l'ADN, on utilise classiquement $\Sigma = \{A,C,G,T\}$ et $\sigma = 4$. On parlera également dans ce cas de lettre ou encore de nucléotide pour désigner les symboles de l'alphabet.

À partir d'un alphabet Σ, une suite finie et ordonnée de symboles de Σ est appelée une *chaîne*. Lorsque la chaîne représente une séquence biologique, donc une suite de nucléotides, par abus de langage on utilisera le mot *séquence*. Le terme *mot* est employé pour désigner une chaîne dans un cadre abstrait. Par opposition, le terme *motif* ou séquence est utilisé pour exprimer une connotation biologique. Le *mot vide*, qui ne contient aucun caractère est noté ϵ.

Nous allons présentement définir ce que sont une séquence, un préfixe, un suffixe et finalement la notion de facteur.

Définition 1.2. Une *séquence* w de taille m est un mot composé d'une suite finie de symboles pris dans l'ensemble Σ tel que, $w = w[0], w[1], ..., w[m-1]$ où $w[0]$ est le premier symbole et $w[m-1]$ le dernier symbole de w.

Définition 1.3. L'opération de *concaténation* est une opération binaire sur les mots qui consiste à accoler le deuxième mot au premier. La concaténation des mots u et v est notée uv.

Définition 1.4. Un mot p est un *préfixe* d'un mot w s'il existe un mot v tel que $w = pv$. L'ensemble des préfixes de w est noté $Pref(w)$.

Définition 1.5. Un mot s est un *suffixe* d'un mot w s'il existe un mot u tel que $w = us$. L'ensemble des suffixes de w est noté $Suff(w)$.

Définition 1.6. Un mot f est un *facteur* d'un mot w s'il existe deux mots u et v tels que $w = ufv$. L'ensemble des facteurs de w est noté $Fact(w)$.

Définition 1.7. Le terme k-mer est employé pour désigner un facteur de longueur k.

Remarque 1. Un facteur est le préfixe d'un suffixe ou encore le suffixe d'un préfixe.

Exemple 1.1 (Le préfixe, le suffixe et le facteur)**.**

Soit w = GATTACA une séquence d'ADN, alors :
$Pref(w)$ = {ϵ, G, GA, GAT, GATT, GATTA, GATTAC, GATTACA},
$Suff(w)$ = {ϵ, A, CA, ACA, TACA, TTACA, ATTACA, GATTACA},
$Fact(w)$ = {ϵ, A, C, G, T, GA, AT, TT, TA, AC, CA, GAT, ATT, TTA, TAC, ACA, GATT, ATTA, TTAC, TACA, GATTA, ATTAC, TTACA, GATTAC, ATTACA, GATTACA}.

Notons que pour les séquenceurs haut débit, quelques termes spécifiques sont couramment employés. Les *reads* ou *short reads* sont associés aux courtes lectures (de l'ordre de la centaine de pb) issues des séquenceurs. Les *long reads* sont quant à eux associés aux longues lectures

(de l'ordre du millier voire de la centaine de milliers de *pb*) issues des séquenceurs. Il est à noter que les notions de *short* et de *long reads* sont également étroitement liés aux technologies de séquençage utilisées. L'*assemblage* des *reads* est une méthode qui consiste à obtenir un ensemble de séquences approximant au mieux la séquence d'un génome. Pour cela, il faut créer des séquences génomiques continues et ordonnées appelées *contigs*.

Ces *contigs* (séquences sans intervalle obtenues par chevauchement des *reads*) forment par *échafaudage* des *scaffolds*. Un *scaffold* sera constitué par plusieurs *contigs* ordonnés. L'assemblage peut-être comparé à un puzzle. Les pièces du puzzle sont les *reads* issues du séquenceur, l'utilisateur les regroupe par ressemblance pour former des *contigs*. Puis il regroupe les morceaux déjà assemblés pour essayer de former l'ensemble du puzzle. Le *mapping* est la méthode qui consiste à aligner des *reads* sur une séquence génomique. Dans notre analogie avec le puzzle, elle correspondrait à positionner nos pièces (*reads*) sur le dessin (séquence de référence) pour voir quelles sont les pièces identiques et celles que ne le sont pas.

Dans cette thèse, nous considérons les mots issus à la fois de génomes assemblés et de données de séquençage haut-débit.

1.3 Séquençage

Frederick Sanger et ses collaborateurs ont réalisé le premier séquençage de l'ADN dès 1977 [Sanger et al., 1977]. Le $\phi X174$ devient le premier organisme (virus de $6Kpb$) dont la séquence génomique a été découverte. C'est 26 ans après le premier organisme séquencé que le premier génome humain de $3Gpb$ voit le jour [Collins et al., 2004].

Dans cette sous-partie nous aborderons les différents types de séquenceurs, les erreurs dûes aux nouvelles méthodes de séquençage, les difficultés qui ont vu le jour suite à ce flot de données et les applications biologiques qui sont nées pour répondre à ces nouveaux besoins.

Il existe une myriade de séquenceurs et leurs technologies évoluent rapidement. Ne seront introduites dans ce manuscrit que celles en adéquation avec les données auxquelles nous nous intéressons dans le cadre du projet *GenomeHarvest*.

Définition 1.8. Un *run* est un cycle d'utilisation d'une machine permettant l'extraction de séquences génomiques. Un *run* comporte plusieurs échantillons et un séquenceur peut effectuer plusieurs *runs* dans une même journée.

La société Illumina® (qui a racheté Solexa en 2007) proposait initialement la technologie *HiSeq*™ qui a une approche PCR et permet de produire 80 000 000 à 100 000 000 *reads* par *run*. À l'issue de ces *runs*, la longueur des *reads* varie de $35bp$ à $150bp$. Son débit est de $8Gbp$ à $10Gbp$ et il faut compter de 3 à 10 jours pour 1 *run* [Bentley et al., 2008]. Cette société propose actuellement la technologie *NextSeq*™ qui propose des *runs* de 12 à 30 heures pour une sortie de $120Gpb$, 400 000 000 *reads* par *run*, et des *reads* de longueur $2 \times 150bp$.

Les laboratoires Roche® utilisent une approche par pyroséquençage ; le pyroséquençage est basé sur le principe de « séquençage par synthèse ». À chaque nucléotide est rattaché un pyrophosphate qui va libérer une lumière qui pourra être détectée lors de la réaction. Cette technologie permet 400 000 à 1 000 000 *reads* par *run* ; le séquençage propose des *reads* de longueur de $250bp$ à $400bp$ avec un débit de $100Mpb$ à $400Mpb$ par *run* et des *runs* plus rapides, de 7h30 à 10h [Citation : Luo et al., 2012].

L'Oxford Nanopore® Technology MinION™ l'ONTM est le premier séquenceur disponible dans le commerce qui utilise des nanopores. Le séquençage par nanopores discrimine les

nucléotides individuels en mesurant le changement de conductivité électrique lorsque les molécules d'ADN traversent un pore biologique (dont le diamètre est de l'ordre du nanomètre). L'ONTM est un dispositif de séquençage portable générant des longueurs de *reads* maximales supérieures à 100 *Kpb* [Sevim et al., 2019].

1.3.1 Les différents séquenceurs

Avant le passage aux nouvelles technologies de séquençage (NGS), les séquenceurs Sanger ont largement dominé l'industrie du séquençage. L'arrivée des NGS dans le domaine de la génomique a considérablement modifié les approches scientifiques. Nous concentrerons notre attention sur les trois technologies les plus utilisés : Illumina (Solexa), Roche 454, SOLiD. Ces technologies relativement récentes permettent de séquencer l'ADN et l'ARN beaucoup plus rapidement que les méthodes précédentes comme le séquençage de Sanger, et comme telles ont révolutionné l'étude de la génomique et de la biologie moléculaire. Les technologies NGS présentent 3 étapes communes :

La préparation de *banques* : les banques sont créées en utilisant une fragmentation aléatoire de l'ADN suivie de la liaison avec des petites séquences spécifiques.

L'*amplification* : la banque est amplifiée grâce à des méthodes d'amplification clonale et d'amplification en chaîne par polymérase (PCR).

Le *séquençage* : l'ADN est séquencé en utilisant différentes approches en fonction de la technologie utilisée.

La technologie NGS la plus utilisée est la technologie Illumina. Cette technologie utilise l'amplification clonale et la séquençage par synthèse. Le processus permet d'identifier simultanément les bases d'ADN lorsqu'elles sont incorporées dans la chaîne d'acide nucléique. Chaque base émet un signal de fluorescence unique lorsqu'elle est ajoutée au brin en cours de synthèse ; ceci est utilisé pour déterminer la séquence d'ADN. La technologie NGS peut être utilisée pour séquencer l'ADN de n'importe quel organisme.

1.3.2 Les erreurs dûes au séquençage

Dans un projet complet de séquençage, la notion de *profondeur* est le rapport entre la longueur du génome et la longueur de l'ensemble des séquences mises bout à bout. La profondeur a un lien direct avec la notion de couverture, à savoir le nombre de *reads* qui vont chevaucher une région donnée.

Exemple 1.2 (Profondeur de séquençage)**.**

Soit le génome d'un organisme A noté G_A de 10 millions de bases (Mpb). Sa taille est notée $|G_A|$ et le résultat du séquençage de G_A est une séquence noté S_{G_A} de 50 Mpb (sa taille est notée $|S_{G_A}|$).
Nous avons une profondeur notée PX de 5 fois la taille du génome que nous notons $5X$.

$$PX = \frac{|S_{G_A}|}{|G_A|}X = \frac{50Mpb}{10Mpb}X = 5X$$

Plus PX est élevé plus la couverture du génome sera importante, et donc plus nombreux seront les *reads* chevauchants.

Ceci permet d'obtenir une séquence dite assemblée la plus complète possible, avec un nombre de « trous » minimum (régions non séquencées).

Toutefois, si l'augmentation de la profondeur du séquençage permet de diminuer ces lacunes de séquence, il arrive un seuil où il est plus économique de combler les quelques trous restant avec des stratégies intelligentes au lieu d'augmenter la profondeur. Augmenter la profondeur de séquençage est coûteux sur plusieurs points : (i) le stockage et (ii) le traitement des données générées. De plus, il peut y avoir des biais de représentation qui font que certaines régions sont moins couvertes, voire jamais couvertes.

Il est donc judicieux au préalable de calculer une bonne profondeur pour le séquençage pour obtenir une bonne couverture.

La représentation mathématique de ces trous peut-être définie comme suit : la longueur moyenne m des *reads* étant inférieure à la longueur L du génome, on peut considérer que la probabilité qu'une base de la séquence cible soit représentée dans x *reads* suit une loi de Poisson de paramètre PX donc $Prob(X = x) = \frac{P^x \times e^{-P}}{x!}$ et nous pouvons donc estimer la profondeur $PX = \frac{N}{L}X$ avec N le nombre de nucléotides total des *reads* et L la taille de l'ADN étudié. Nous pouvons également estimer le nombre de trous en fonction de la couverture et la taille moyenne des trous. Le nombre de trous est $\frac{N}{m} \times e^{-\frac{N}{L}}$ et la taille des trous est : $L \times \frac{m}{N}$ [Lander and Waterman, 1988].

Exemple 1.3 (Application aux données de riz du projet 3 000 génomes)**.**

$L = 5 \times 10^8$ *nucléotides*

$m = 100$ *nucléotides*

$N = 5 \times 10^9$ *nucléotides*

$PX = \frac{5.10^9}{5.10^8}X = 10X$

Nombre de trous : $\frac{5\times10^9}{100} \times e^{-10} = 2270$ *trous.*

Taille des trous : $5 \times 10^8 \times \frac{100}{5\times10^9} = 10$ *nucléotides.*

1.4 Algorithme de recherche de motifs

La recherche de motifs, de sous-chaînes de caractères, dans une chaîne de caractères plus longue est un composant important de nombreux algorithmes en bioinformatique. Les textes en bioinformatique ont une taille pour les plus grands génomes allant de 10^6 paires de bases (lettres) à 10^{11} paires de bases (*e.g*, *Paris japonica*) ; il faut donc chercher à réduire le nombre de comparaisons. Les stratégies les plus connues sont dites stratégies *online* ou séquentielles et ne requièrent pas une connaissance *a priori* du texte dans lequel on cherche.

1.4.1 Approche naïve

Recherchons un motif de longueur n dans un texte de longueur m alors dans le pire des cas, il nous faudra $O(n \times (m - n))$. L'algorithme naïf est l'algorithme le plus basique auquel il semble trivial de penser. Dans ce cas, il suffit de placer le motif à rechercher au début de la séquence et pour chaque lettre, regarder si elle correspond à la lettre du texte dans lequel la recherche de motif est effectuée. L'algorithme naïf répète cette opération pour chaque lettre jusqu'au texte complet moins la taille du motif. Dès qu'un caractère ne correspond plus alors il décale le motif d'une position et il le compare à nouveau à partir de la première position. Il existe deux grands moyens d'améliorer cette approche, d'une part prétraiter le motif, ce que nous décrivons ci-dessous, et d'autre part, exploiter la connaissance que l'on peut avoir du texte dans lequel nous souhaitons effectuer la recherche d'un motif.

1.4.2 Algorithme de Boyer-Moore

L'algorithme de Boyer-Moore commence la comparaison par l'extrémité droite du motif et le motif est prétraité de manière à ce que, tout au long de la recherche, nous puissions utiliser les informations sur les occurrences des lettres au sein du motif. Plus le motif est long, plus le gain est important. Si la dernière lettre du motif ne correspond pas à la $n^{\text{ème}}$ lettre du texte et que la $n^{\text{ème}}$ lettre du texte n'apparaît pas dans notre motif alors on peut avancer le motif de n lettres. Si la dernière lettre du motif correspond et que la $(n-1)^{\text{ème}}$ lettre ne correspond pas, si elle existe dans le motif alors on fait progresser le motif jusqu'à la prochaine occurrence de la lettre dans le motif (dans le sens des préfixes) à l'indice i dans notre motif, si la $n^{\text{ème}}$ lettre du motif correspond à la $(n+i)^{\text{ème}}$ lettre dans le texte ainsi de suite. Cet algorithme minimise très largement le nombre de comparaisons au prix d'un prétraitement des informations contenues dans le motif. La complexité de l'algorithme de Boyer-Moore est en $O(m+n)$ pour trouver la première occurrence du motif, la complexité pour trouver toutes les occurrences est en $O(mn)$ avec m la longueur du motif recherché, et n la longueur du texte où l'on recherche.

L'exploitation du motif ne suffit cependant pas en génomique pour un passage à l'échelle qui soit satisfaisant. Nous allons donc prétraiter le texte. Il existe différentes façons de prétraiter le texte, les plus efficaces étant de stocker des informations de façon structurée, dans des structures de données *ad hoc*. L'objectif de ce travail de thèse est de proposer une structure de données qui sait s'adapter aux besoins rencontrés en analyse pangénomique. Le prétraitement du texte sera présenté dans la section 2.4.

1.5 Le riz

Le riz est une céréale de la famille des Poacées. À lui seul, il représente 20% de l'alimentation mondiale et la principale source nutritive de plus de 3 milliards d'êtres humains. La culture du riz nécessite des quantités très importantes d'eau, plus de 1 500 tonnes d'eau par tonne de riz. La séquence génétique complète du riz a été publiée en 2005 dans le journal Nature [Matsumoto et al., 2005]. Son génome représente entre 400 et 500 millions de paires de bases d'ADN, comptant un peu moins de 38 000 gènes répartis sur douze chromosomes.

Dans cette publication ont été identifiés exactement 37 544 gènes du riz, et la position de chaque gène a été établi sur chacun des 12 chromosomes du riz. Une annonce a également été faite portant sur l'identification de quelques gènes particulièrement importants qui pourraient accroître le rendement et la productivité. Tout n'a pas encore été découvert : l'origine du riz ainsi que le rôle de certains de ses gènes ne sont pas, à ce jour, complètement connus.

Le projet international de séquençage du génome du riz (IRGSP), un consortium de laboratoires publics, fondé en 1997, a visé à obtenir une séquence cartographiée très précise du génome du riz en utilisant le *cultivar*[3] Nipponbare de *Oryza sativa japonica*. L'IRGSP, dirigé par le Japon compte neuf autres membres : les États-Unis, la Chine, Taiwan, la Corée, l'Inde, la Thaïlande, la France, le Brésil et le Royaume-Uni. Les études sur le riz sont nombreuses et ce manuscrit s'intéresse seulement aux analyses génomiques faites sur le riz [Matsumoto et al., 2005].

Le riz, sur le plan génomique, se rapproche du maïs, du blé, de l'orge, du seigle, du sorgho et de la canne à sucre, ce qui signifie que l'étudier permettra d'apporter des solutions sur le riz, mais aussi comprendre les autres cultures vivrières proches. Le décryptage du génome

3. Sont appelées *cultivar* les variétés obtenues en culture et sélectionnées pour leurs caractéristiques.

du riz permet de proposer une histoire évolutive cohérente, d'éventuellement sélectionner certains gènes pour permettre aux agriculteurs d'accroître la production de riz, de protéger les cultures contre les maladies et les ravageurs et de rendre le riz, de même que d'autres cultures céréalières, résistant à la sécheresse, ce qui contribue à la lutte contre la faim dans le monde [Zhao et al., 2018].

Le génome du riz est plus complexe que celui d'*Arabidopsis thaliana*, la plante modèle en biologie moléculaire des végétaux [Sijmons et al., 1991]. Connaître sa carte génétique a des retombées importantes pour l'agriculture et l'agronomie, pour la culture du riz lui-même et pour celle d'autres plantes comme le blé ou l'orge.

Le riz a une ampleur spectaculaire d'adaptation aux différentes latitudes, aux différentes altitudes et aux différents régimes hydrologiques. L'espèce prédominante de riz cultivé est *Oryza sativa*, domestiquée en Asie, et une autre espèce, *Oryza glaberrima*, a été domestiquée en Afrique.

Le riz asiatique a deux principaux types de variétés, souvent appelés *Indica* et *Japonica*. Il existe d'autres groupes, un groupe spécifique comprenant les variétés dites *Aus* du nord-est du sous-continent indien et un groupe comprenant les célèbres variétés *Basmati*.

Selon l'état de l'art en évolution moléculaire, la différenciation *Indica-Japonica* remonte à plus de 400 000 ans [Ma and Bennetzen, 2004], bien avant le début de la domestication. Les gènes contrôlant des caractères sous sélection humaine, des caractères tels que le parfum ou son comportement à la cuisson, sont soumis au même phénomène [Kovach et al., 2009].

D'autres découvertes suggèrent que les premières formes de *japonica* ont été introgressées [4] par un riz de type *indica*. Cette introgression a permis au *japonica* de devenir un riz cultivé de référence. Nous parlerons alors de *cultivars japonica* [Izawa et al., 2009].

Les discussions globales sur la domestication du riz en Asie portent sur la différenciation entre *Indica* et *Japonica* et négligent les types mineurs concentrés sur le sous-continent indien [Chong and Xu, 2014]. Des foyers spécifiques continuent de révéler de nouvelles grappes de variétés, telles que les riz aromatiques du Myanmar [Myint et al., 2012].

Le séquençage de 3 000 variétés de riz a été rendu public, *The 3 000 rice genomes project*, ainsi qu'un appel à une action coordonnée en termes d'analyse des données [Zhao et al., 2018].

Une initiative française, le projet IRIGIN (*International Sequencing Initiative of Major Genetic Stocks and Resources for Rice Knowledge*) piloté par l'une des équipes de *GenomeHarvest*, ajoute des données sur les matériaux complémentaires du riz africain et d'éventuels hybrides interspécifiques.

Afin d'établir de façon très détaillées, à plusieurs échelles, les interactions entre tous ces génomes, il est nécessaire de disposer d'un outil suffisamment souple, complet et efficace.

Ces considérations sous-jacentes ont guidé tout le travail de cette thèse. La souplesse est apportée par l'approche par k-mers, qui permet d'identifier des structures similaires avec une granularité fine, tandis que la recherche de performance s'appuie sur la conception de la structure de données.

Dans le chapitre suivant, nous allons détailler les différents outils manipulant les k-mers ; outils desquels nous nous sommes inspirés.

4. L'introgression des gènes est définie comme étant le transfert des gènes d'un individu vers un autre individu par hybridation. Le terme « introgressé » désigne le transfert de gènes d'une espèce vers une autre espèce génétiquement proche.

Chapitre 2

L'approche par k-mers, comptage et indexation

Ce chapitre présente un état de l'art succinct et non exhaustif des structures et méthodes exploitant les k-mers. Ont été sélectionnées celles qui interviennent dans les outils avec lesquels nous sommes susceptibles de nous comparer.
Les principales méthodes de comptage de k-mers sont ensuite présentées.
Dans une troisième section nous décrivons les notions gravitant autour des structures succinctes, avant de décrire les structures d'indexation de texte les plus courantes.

2.1 Notion et utilisation des k-mers

En bioinformatique, l'utilisation des k-mers devient de plus en plus fréquente. Le site omicX [1] dénombre – au moment de la rédaction de ce manuscrit – 685 outils utilisant une approche par k-mers.
Les utilisations sont variées ; les principales catégories sont l'assemblage *de novo* (`ABySS` [Simpson et al., 2009], `Meraculous` [Chapman et al., 2011], ...), l'alignement de *reads* sur

1. *OmicTools* (`https://omictools.com/`) est une base de données regroupant les outils adaptés aux analyses des données omiques.

une séquence de référence (`SOAP` [Li et al., 2008], `HiLive` [Lindner et al., 2016], ...), la correction d'erreurs (`Bloocoo` [Benoit et al., 2014], `Quake` [Kelley et al., 2010], ...), la prédiction de répétitions en tandem (`Kmer-SSR` [Pickett et al.,], `kissplice` [Sacomoto et al., 2012], ...), le *clustering* de données d'expression des gènes (`ClusterEnG` [Manjunath et al., 2017], `KABOOM` [Hazelhurst and Lipták, 2011], ...) ou tout simplement, les outils de comptage de k-mers dans les séquences (*Jellyfish* [Marcais and Kingsford, 2011], `KMC` [Deorowicz et al., 2013b], ...).

Il existe différentes manières de représenter en mémoire des ensembles de k-mers. Une solution *a priori* évidente est l'utilisation de tables de hachage. Cela offre l'avantage d'être relativement simple à mettre en œuvre, d'accéder rapidement aux éléments indexés, mais cela présente le désavantage majeur d'être consommateur de mémoire. D'autres solutions vont par exemple exploiter les filtres de Bloom. Ces filtres sont des structures probabilistes, ce qui signifie qu'il peut y avoir des faux positifs, mais elles ont l'avantage de maîtriser la mémoire utilisée. Moins il y a de mémoire, plus il est probable que les k-mers soient représentés alors qu'ils ne sont pas réellement présents dans le jeux de données de départ.

Dans le contexte de l'assemblage *de novo*, il s'agit donc d'assembler des millions de *reads* ensemble, dont on a aucune information sur le positionnement (ni le début ni la fin) et qui contiennent potentiellement des erreurs de séquençage.

L'approche est de construire le génome pas à pas en ajoutant les *reads* les uns après les autres. Pour ce faire il faut rechercher les chevauchements. Une des difficultés est que plusieurs chevauchements sont possibles. De nouveaux outils ont été conçus pour fournir un haut niveau de précision, comme HISEA [Khiste and Ilie, 2017] ; le niveau de précision étant le score de ressemblance entre deux séquences. Leurs temps de calcul pour la comparaison de nombreuses et longues séquences sont cependant très longs. Il faut également faire face aux séquences contenant des insertions, des suppressions, des erreurs inhérentes au séquençage (produites lors de la phase d'amplification ou dues tout simplement à une mauvaise lecture par la machine), erreurs qui sont de l'ordre de $0,1\%$ à 1% de l'ensemble des bases pour Illumina [Metzker, 2010].

L'inconvénient de l'approche par k-mers est que le volume des données est multiplié. Pour un mot de longueur m, il y aura $(m-k+1)$ k-mers. Il n'est donc pas envisageable de les stocker tels quels. Naïvement, en utilisant une table contenant tous les $20-$mers possibles avec un compteur associé, il y aurait 4^{20} entrées de 20 lettres codées chacune sur 1 octet soit environ 20 TiB.

C'est pour cela qu'il est nécessaire de structurer l'information afin de trouver un compromis acceptable entre le temps de traitement et la mémoire utilisée.

2.1.1 Espace et mémoire

En faisant abstraction de l'utilisation d'un alphabet dégénéré pour dénoter les différentes combinaisons de nucléotides (standard IUPAC), dans sa version la plus simple, un k-mer est un mot composé de quatre lettres différentes. Il n'est pas nécessaire d'utiliser un octet complet (256 valeurs distinctes) pour encoder une lettre. Elles peuvent donc être codées en binaire, sur seulement 2 bits et permettre ainsi la réduction de la mémoire nécessaire par 4.

Quand bien même, on souhaiterait permettre l'usage d'un alphabet dégénéré (soit 16 symboles pour les $\sum_{i=0}^{4} \binom{i}{4} = 2^4$ combinaisons), il est possible de représenter deux symboles par octet, ce qui permettrait tout de même de diminuer la mémoire de moitié. Cependant, les données issues de séquençages (et par conséquent la majorité des séquences de génomes ou de transcriptomes produites à partir de ces données) sont constituées des cinq lettres

$\{A, C, G, T, N\}$, le N étant utilisé non pas pour signifier que les quatre nucléotides ont été trouvés à cette position de la séquence (ce qui est l'objectif premier de la nomenclature IUPAC [IUPAC IUBMB JCMB, 1983]), mais pour signifier qu'il existe un nucléotide qui n'a pas pu être déterminé. Ainsi, pour représenter ces cinq possibilités, il faudrait utiliser 3 bits par lettre au lieu de 2, ce qui représente un surplus de 50% de mémoire, pour une information somme toute partielle.

Une autre optimisation consiste a utiliser une forme canonique des k-mers ; c'est-à-dire en choisissant une représentation des k-mers telle qu'un k-mer et sa séquence complémentaire inversée aient une représentation identique. Cela permet de diminuer le nombre de k-mers représentés en mémoire. Une représentation canonique que l'on retrouve dans la quasi-totalité des outils est, pour un k-mer et son complémentaire inversé, de choisir le plus petit selon une relation d'ordre sur les séquences (très souvent l'ordre lexicographique, mais ce n'est pas systématique). Par exemple, DSK [Rizk et al., 2013] utilise la relation d'ordre $A < C < T < G$, et Turtle [Roy et al., 2014] utilise la première occurrence rencontrée). Cette optimisation permet de diviser l'empreinte mémoire jusqu'à un facteur deux.

2.1.2 Table de hachage

Une table de hachage, ou *hash table* en anglais, est une structure de données qui utilise une fonction de hachage pour calculer à partir d'une clé, à quel indice de la table, le couple clé-valeur doit être stocké. Cela permet de stocker des couples clé-valeur et de retrouver plus ou moins efficacement (selon la fonction de hachage utilisée et la dimension de la table) la valeur associée à une clé donnée. Une fonction de hachage est une fonction mathématique qui, à partir d'une donnée en entrée calcule une signature ; cette signature permet d'identifier rapidement un élément dans la table. Cette signature, qui peut-être vue comme un score, a un impact direct sur le nombre de données représentables sans qu'il n'y ait de collisions (c'est-à-dire que deux données en entrée peuvent fournir la même signature).

C'est pourquoi la taille du score doit être optimisée (chercher le plus petit score, la plus petite signature) pour minimiser l'impact que peuvent avoir ces collisions, ou garantir l'absence de collision, et maximiser le nombre de données représentables [Organick, 1975].

2.1.3 Filtre de Bloom

Un filtre de Bloom est un algorithme qui permet de tester l'appartenance d'un élément à un ensemble de données. Les filtres de Bloom utilisent un ensemble de fonctions de hachage.
Les fonctions de hachage renvoient un entier unique compris entre 0 et n. Il est possible d'estimer le nombre de collisions. Il est également possible de réduire le nombre de collisions et intrinsèquement le nombre de faux positifs en augmentant la taille de n [Holley et al., 2016].
Il est possible d'estimer le nombre de faux positifs (noté P_{fp}). L'estimation s'effectue en fonction de n la taille du tableau booléen [2], m le nombre d'éléments et f_h le nombre de fonctions de hachage utilisées.
Supposons que la probabilité qu'une fonction de hachage change un bit donné du tableau booléen est homogène et donc égale à $\frac{1}{n}$. La probabilité que ce bit ne change pas avec cette fonction est donc de $1 - \frac{1}{n}$. La probabilité qu'un bit ne change pas avec l'ensemble des fonctions est donc de $\left(1 - \frac{1}{n}\right)^{f_h}$. Par conséquent, sur l'ensemble des m éléments, la probabilité

2. Un booléen en informatique est un type de variable à deux états ici 1 et 0

qu'un bit du vecteur ne soit jamais à 1 (toujours à 0) est de $\left(1-\frac{1}{n}\right)^{f_h\, m}$. On a :

$$\left(1-\frac{1}{n}\right)^{f_h\, m} = e^{f_h\, m\, \log\left(1-\frac{1}{n}\right)} \quad .$$

En utilisant le développement en série de Maclaurin, on obtient :

$$\log(1+x) = \sum_{i=0}^{+\infty} \frac{x^i}{i} \quad (\text{si } -1 < x < 1)$$
$$\Leftrightarrow \log(1-x) = -\sum_{i=0}^{+\infty} \frac{x^i}{i} \quad (\text{si } -1 < x < 1) \quad .$$

Par conséquent :

$$f_h\, m\, \log\left(1-\frac{1}{n}\right) = -f_h\, m \sum_{i=1}^{+\infty} \frac{\left(\frac{1}{n}\right)^i}{i}$$
$$= -f_h\, m\, \left(\frac{1}{n}\right) + o\left(\frac{f_h\, m}{n}\right)$$
$$\left(1-\frac{1}{n}\right)^{f_h\, m} \simeq e^{\left(-\frac{f_h\, m}{n}\right)} \quad .$$

On en déduit que la probabilité qu'un bit du vecteur soit à 1 est $1-e^{\frac{f_h\, m}{n}}$. Il en résulte que si les fonctions sont considérées comme indépendantes on obtient $P_{fp} = \left(1-e^{\frac{-f_h\, m}{n}}\right)^{f_h}$

Nombreux sont les outils qui utilisent des filtres de Bloom ou des tables de hachage afin de compter les k-mers présents dans des séquences biologiques (typiquement provenant de NGS). Les outils étudiés dans cette section sont *Jellyfish* [Marcais and Kingsford, 2011], `KMC` [Deorowicz et al., 2013b], `khmer` [Zhang et al., 2014]. D'autres outils existent, mais nous avons sélectionné les plus couramment utilisés.

2.2 Les compteurs de k-mers

Dans cette section, nous présenterons les compteurs de k-mers. En bioinformatique, le fait de découper les séquences et les *reads* en k-mers permet d'avoir une fréquence d'ensemble de k-mers. Cette fréquence peut être utilisée en tant que *signature* de sous-séquence. La comparaison de ces fréquences est mathématiquement plus facile que l'alignement de la séquence entière. Les compteurs de k-mers peuvent donc être utilisés comme une première étape d'analyse avant un alignement.

2.2.1 *Jellyfish*

Jellyfish [Marcais and Kingsford, 2011] est un logiciel dont l'algorithme principal est un algorithme de comptage exact implémenté en C++. Ses opérations sont principalement faites en mémoire RAM, et lorsqu'il y a un dépassement de capacité de mémoire disponible, *Jellyfish* a fait le choix d'utiliser l'espace disque.
Jellyfish applique une fonction de hachage sur les séquences de k-mers, le résultat de la

fonction donne les clés représentant la séquence. Ces clés sont associées à une valeur qui correspond au nombre d'occurrences du k-mer.

L'emplacement mémoire d'un k-mer est déterminé par la valeur de sa clé, ainsi que par une seconde fonction de hachage (appelée hachage secondaire). Dans le cas où deux k-mers différents auraient la même clé (fonction de hachage principale), l'algorithme utilise la fonction $reprobe(i) = \frac{i(i+1)}{2}$ afin de trouver un emplacement disponible dans la table.
Une des particularités de cette implémentation est le recours à l'instruction d'assembleur CAS (*Compare And Swap*) pour incrémenter les compteurs associés aux k-mers qui sont traités en parallèle (parallélisme léger – *multithreading* en anglais). Cette instruction permet de modifier la valeur d'une variable sans blocage des accès concurrents dans un contexte d'exécution parallèle et a pour effet d'améliorer considérablement le temps d'exécution.
L'opération d'incrémentation de la table est gérée par la fonction de hachage, la localisation est déterminée par les hachages principaux et secondaires. Soit la clé est déjà présente soit l'emplacement est vide auquel cas la clé est écrite ou réécrite et sa valeur associée est incrémentée.

Si la localisation est utilisée par un autre k-mer, cas de collision, l'indice i de la fonction *reprobe* est incrémentée jusqu'à ce que la localisation soit vide ou contienne le même k-mer (en réalité, seule une partie du k-mer est stocké car le k-mer complet peut-être recalculé à partir de la clé associée). Pour diminuer les besoins en mémoire, les entrées de la table sont compactées. Les valeurs associées sont codées sur peu de bits car de nombreux k-mers ne se retrouvent qu'une seule fois, soit parce qu'ils contiennent des erreurs de séquençage, soit parce que dans le génome considéré, les répétitions sont rares. La plupart des k-mers se retrouvent approximativement c fois, c représente la couverture du séquençage. Toutefois certains k-mers, se retrouvent de nombreuses fois, dans ce cas une deuxième entrée sera créée avec la même clé key, nous aurons donc deux entrées, $\langle key, v_1 \rangle$ et $\langle key, v_2 \rangle$.

Une fois les tables de hachages remplies, elles sont écrites sur le disque puis triées suivant les plus petits bits de poids faible de la valeur de hachage de m qui est $pos(m, 0)$. Ce tri permet de rassembler ensuite toutes les données dans le cas où un comptage de k-mer serait stocké sur plusieurs entrées.
Le fichier de sortie est un fichier binaire, mais une fonction (`dump`) permet d'obtenir un fichier sous forme de deux colonnes séparées par une tabulation avec la première colonne contenant le k-mer et la seconde, son compte associé (il existe également une possibilité de produire une sortie suivant le format FASTA où le descriptif – *header* – associé à chaque k-mer est le compte de celui-ci). Ce fichier n'est pas trié par ordre lexicographique des k-mers, et la documentation précise qu'il y a bien une relation d'ordre entre les k-mers dans les sorties, mais que cette relation dépend des données en entrée ainsi que du paramètrage utilisé.

2.2.2 KMC

Cet algorithme est déterministe et exact, il est implémenté en C++. Il s'appuie principalement sur l'écriture sur disque plutôt que de stocker les informations en mémoire, en créant une structure de dictionnaire compactée.
La première étape de cette méthode consiste à lire les fichiers provenant du séquençage par portion de quelques mégaoctets.
Les k-mers des *reads* sont extraits et convertis dans une forme canonique (ordre lexicographique) et écrits dans des fichiers suivant un premier tri par préfixe p_1 (n premières lettres de la séquence). Le préfixe p_1 est retiré du k-mer puis distribué suivant un deuxième préfixe p_2 ordonné. Le facteur p_2 qui suit le préfixe p_1 devient le préfixe du mot. Ce préfixe est

également retiré des k-mers. p_1 et p_2 sont choisis pour que $k - |p_1| - |p_2|$ soit un multiple de 4. Il faut donc p_1 et p_2 pour accéder au suffixe correspondant.

Pour effectuer le tri des suffixes, chaque paquet est à nouveau ouvert et les suffixes sont triés par une méthode de tri par base. Cette méthode consiste à prendre le groupe de bits le moins significatif de chaque clef, à trier les éléments suivant l'ordre de ces bits tout en conservant l'ordre des éléments ayant la même valeur. Puis on répète le tri avec les bits plus significatifs. Les suffixes identiques se retrouvent donc adjacents ; on compte leur fréquence et on élimine ceux qui sont en-dessous d'une fréquence choisie par défaut ou spécifiquement par l'utilisateur.

Les paquets sont ensuite triés entre eux par un tri par sélection (parcours de la liste, recherche du minimum et ainsi de suite). Cette méthode de tri est la plus simple ; elle nécessite beaucoup d'opérations, mais à la vue du nombre de paquets relativement restreint, elle est suffisante.

La sortie de cet outil est sous la forme de deux fichiers binaires mais une autre fonction (dump) permet d'obtenir un fichier sous forme de deux colonnes séparées par une tabulation, avec la première colonne contenant le k-mer, et la seconde son compte associé. Ce fichier n'est pas non plus trié par ordre lexicographique des k-mers.

2.2.3 Khmer

Khmer est implémenté en C++, basé sur le principe du filtre de Bloom précédemment décrit, mais au lieu d'utiliser plusieurs fonctions de hachage sur une même table, Khmer utilise une unique fonction de hachage sur plusieurs tables de hachage de tailles différentes (structure probabiliste). Seul le compteur est stocké, sur 8 bits. Cette fonction de hachage est décrite comme suffisamment complexe pour ne pas créer de collision. S'il y a collision dans l'une de ces tables, il est peu probable qu'elle ne se reproduise dans toutes les autres tables.
Toutefois *Khmer* ne travaille pas sur une structure exacte et peut présenter des erreurs.

2.2.4 Gk-Arrays

Les *Gk-Arrays* sont une structure de données, implémenté en C++, pour indexer les k-mers dans une collection de lectures [Philippe et al., 2011]. Étant donné un paramètre k, les tableaux *Gk* indexent toutes les occurrences des k-mers dans la collection d'entrée de lectures. Une fois que l'index est construit et conservé en mémoire, le programme peut demander des requêtes à l'index. Cette structure est composée de trois tables, appelées respectivement *GkSA*, *GkIFA* et *GkCFPS*.

Le *GkSA* (Generalized k Suffix Array) est un tableau de suffixes amélioré qui contient la position et le suffixe de longueur au moins k. Si nous notons q le nombre total de k-mers y compris les k-mers redondants, la taille de ce tableau est $q \times \log_2(q)$ bits. Si les identifiants sont stockés sur des entiers, cela demande d'avantage de ressources mais cela permet de gagner du temps pour accéder aux données.

Le *GkIFA* (Generalized k Inverse Factor Array) est la structure inverse de *GkSA* permettant de récupérer le premier identifiant du suffixe (dans le *GkSA*) correspondant à une position donnée. La taille de ce tableau est la même que la taille de GkSA.

Le *GkCFPS* (Generalized k Counting Factor Array). C'est le tableau de facteurs de comptage k généralisés. Il stocke la somme préfixe du compte de chaque k-mer distinct. Chaque k-mer est représenté par l'index du premier identifiant de chaque suffixe dans le *GkSA* ayant ce k-mer comme préfixe. La somme préfixe permet de récupérer le nombre en soustrayant deux valeurs consécutives dans le tableau. Si nous notons d le nombre de k-mers distincts, la taille

de ce tableau est $d \times \log_2(q)$ bits. Si les sommes sont stockées sur des entiers, cela consomme plus d'espace mémoire, mais permet un accès aux données plus rapide.

Pour récupérer le nombre d'occurrences d'un k-mer donné, il suffit de faire une recherche dichotomique dans le *GkCFPS*. Afin de récupérer également toutes les positions associées, le *GkCFPS* renvoie l'index de début et de fin dans GkSA du sous-tableau contenant tous (et seulement) les positions correspondantes.

Le tableau *GkIFA* est une fonctionnalité supplémentaire des *Gk-Arrays* pour améliorer les requêtes (temps constant au lieu d'un temps logarithmique sur la taille de *GkCFPS*). En effet, lorsqu'une requête porte sur un k-mer extrait d'un *read*, alors la table *GkIFA* permet de consulter les tables *GkCFPS* et *GkCFA* directement aux bons indices. Ainsi, l'empreinte mémoire globale des *Gk-Arrays* est de $(2q + d) \times \log(q)$ bits.

Avec cette structure de données, une requête de comptage pour certains k-mer est réalisée en temps logarithmique sur le nombre de k-mers distincts. Ce temps devient constant si l'identifiant k-mer est connu. La récupération de tous les identifiants d'une requête k-mer prend le même temps que le comptage plus un temps linéaire sur le nombre d'occurrences en raison de l'énumération des identifiants.

L'inconvénient majeur de cette structure est son empreinte mémoire lorsque le nombre de k-mers augmente. Et pour une grande quantité de séquences biologiques, l'utilisation de *Gk-Arrays* dans la pratique peut nécessiter des super-ordinateurs ayant des centaines de Go de RAM.

Illustration des *Gk-Arrays*

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
$C_R[i]$	A	G	A	T	G	A	T	A	A	A	C	G	A	T	A	A	T	A	A	A	T
$g(i)$	0	1	2	3	4			5	6	7	8	9			10	11	12	13	14		

(a) Identification des k-mers dans l'ensemble des séquences. On considère ici le texte C_R construit par la concaténation des *reads* $\{AGATGAT, AAACGAT, AATAAAT\}$. La fonction $g(i)$ associe à la position i de C_R, un identifiant unique de 3-mers.

j	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
$GkIFA[j]$	4	8	6	10	8	0	1	3	7	8	2	5	9	0	2
$GkSA[j]$	5	13	6	10	14	7	0	11	2	8	1	4	9	12	3
$Rank$	0		1	2		3	4	5	6	7	8			9	10

(b) Table inverse des facteurs et table des suffixes associées aux k-mers. La table *GkSA* correspond aux 3-mers triés dans l'ordre lexicographique puis d'apparition dans C_R en cas d'égalité ; chaque 3-mer étant représenté par son identifiant dans C_R (fonction g). La table *GkIFA* associe à chaque 3-mer son rang lexicographique. La table *Rank* est fournie à titre indicatif, elle n'est pas construite.

j	-1	0	1	2	3	4	5	6	7	8	9	10
$GkCFA[j]$		2	1	2	1	1	1	1	1	3	1	1
$GkCFPS[j]$	0	2	3	5	6	7	8	9	10	13	14	15

(c) La table des *GkCFA* associe à un k-mer son nombre d'occurrences dans C_R. La table *GkCFPS* regroupe la somme cumulative des *GkCFA*.

Table 2.1: Exemple de structure des *Gk-Arrays* pour une collection de 3 *reads* avec $k = 3$. Les *Gk-Arrays* sont construits pour la collection $R = \{AGATGAT, AAACGAT, AATAAAT\}$. La table (a) montre le texte C_R qui est la concaténation de l'ensemble des *reads* de R. La fonction g permet la numérotation des 3-mers de C_R. La table $GkIFA$ (table (b)) permet de retourner la position lexicographique de chaque 3-mer.

La table 2.1 regroupe trois sous tables qui permettent d'illustrer la création de la table des *Gk-Arrays*. La table (a) enregistre les indices correspondant au point de départ des k-mers dans le texte C_R, lui même construit par la concaténation de tous les *reads*. Dans la table (a) la première ligne i correspond à l'ensemble des indices allant de 0 à 20. La seconde ligne $C_R[i]$ est l'ensemble des lettres correspondant à leur indice i dans le texte. Finalement la dernière ligne $g(i)$ utilise la fonction g pour renuméroter les positions des lettres de C_R pour les faire correspondre a des 3-mers. Par exemple, pour $i = 5$, $g(5)$ n'a pas de valeur dans le tableau, $C_R[5] = ATA$, le 3-mer ATA n'existe pas à cette position dans le texte C_R. Un autre exemple, $i = 8$, $g(8) = 6$, $C_R[8] = AAC$, le 3-mer AAC est le $6^{\text{ème}}$ 3-mer dans le texte C_R.

La table (b) présente deux tables, celle des (Generalized k Inverse Factor Array) *GkIFA*, une table inverse de suffixes modifiée qui enregistre pour chaque position, l'ordre d'apparition des k-mers dans le texte, le rang lexicographique des k-mers. Et celle des (Generalized k Suffix Array) *GkSA*, table des suffixes modifiée qui range les k-mers par ordre lexicographique.

La table (c) présente deux tables, celle des (Generalized k Counting Factor Array) *GkCFA*, une table qui associe à un k-mer (plus précisément à son rang) son nombre d'occurrences. Et celle des (Generalized k Counting Factor Prefix Sum) *GkCFPS*, qui est la somme cumulative des *GkCFA*.

Par exemple, le 3-mer TAA possède 3 occurrences dans $C - R$ aux positions 6, 13 et 16. Parmi ces positions, seulement 16 est une position valide. Le rang lexicographique de TAA est donné par $GkIFA[g(16)] = GkIFA[12] = 9$. Le nombre d'occurrence est donné par la table *GkCFA*, $GkCFA[9] = 1$, il y a donc une seule occurrence du 3-mer TAA dans les *reads* de cet exemple.

2.2.5 Synthèse sur les compteurs de k-mers

Nous excluons d'ores et déjà que la structure que nous souhaitons mettre en place puisse s'appuyer sur une gestion approximative des k-mers.

Le comptage des k-mers est une étape indispensable pour de nombreuses tâches en rapport avec l'analyse de génomes. On peut ainsi détecter notamment la présence ou l'absence de certains k-mers. En revanche, compter les k-mers ne suffit pas à réaliser des analyses plus poussées nécessitant l'indexation, c'est-à-dire l'information de positionnement dans les génomes étudié.
De plus, comme nous l'avons vu, les sorties des outils de comptage de k-mers ne sont pas pratiques pour effectuer des requêtes sur ces k-mers. Le volume pris par ces sorties, et la nécessité de prendre en compte un nombre croissant de séquences, nous a poussés à nous intéresser aux structures succinctes, qui elles sont conçues justement pour faciliter le requêtage.

À notre connaissance, la seule structure de données existante qui fournit une indexation k-mers sont les *Gk-Arrays* [Philippe et al., 2011]. Toutefois, cette structure présente comme inconvénient majeur une occupation mémoire conséquente qui rend assez rapidement inappropriée cette structure pour traiter de grands jeux de données. Il existe également la structure *PgSA* [Kowalski et al., 2015] qui est basée sur les travaux des *Gk-Arrays*. La différence est basée sur la construction d'un pseudogénome à partir des chevauchements des *reads* dans la collection et en utilisant le tableau des suffixes épars pour effectuer les requêtes.

2.3 Structures succinctes

Lorsque les données sont compressibles, comme par exemple dans des textes en langue naturelle. La structure de données compressée tend à occuper une place proche du minimum théorique.

Une structure de données succincte est une structure de données qui, pour encoder une information, utilise une quantité d'espace « proche » de la limite inférieure de l'espace nécessaire pour représenter des données.
Cette limite inférieure peut être estimé par la théorie de l'information développée par Shannon [Shannon, 1948]. Chaque symbole x_i constituant une information I a une probabilité p_i d'apparaître, l'entropie de Shannon H est définie par :

$$H(I) = -\sum_{x_i \in I} p_i \log_2 p_i \tag{2.1}$$

Plus formellement, une structure succincte représentant une information I est une structure de donnée qui utilise $Z + o(Z)$ bits en espace pour représenter I où Z est le nombre optimal de bits nécessaires pour représenter l'information I (selon la théorie de l'information de Shannon).
Cependant, contrairement aux autres représentations compressées, la structure de données succincte permet des opérations de requête efficaces. Parmi les structures succinctes, nous nous sommes intéressés à la représentation des vecteurs binaires.
Les opérations `Rank()` et `Select()` sont décrites ci-dessous, elles sont le fondement du requêtage sur ces structures.

Contrairement aux algorithmes généraux de compression de données sans pertes tels que l'arbre d'Huffman [Huffman, 1952] ou le codage de Lempel-Ziv [Ziv and Lempel, 1978], les structures succinctes permettent en plus un accès aux informations sans avoir à décompresser au préalable le vecteur binaire en utilisant les opérations, `Rank()` et `Select()` [Gog et al., 2014].

2.3.1 Opérations `Rank()` et `Select()`

Les opérations `Rank()` et `Select()` sont des opérations s'appliquant sur des vecteurs binaires. Elles sont cependant généralisables à n'importe quel vecteur dont les éléments sont représentables en binaire.

L'opération $Rank_1(x)$ permet d'obtenir le nombre de bits à 1 jusqu'à une position x (exclue) donnée dans le vecteur. À l'inverse, l'opération $Rank_0(x)$ donne le nombre de bits à 0 jusqu'à une position x.

L'opération `Select()` est l'opération contraire de `Rank()`. Elle indique à quelle position se situe le $i^{\text{ème}}$ bit à 1 (inclus). On la note $Select_1(i)$ où i représente le nombre de bits à 1 souhaité. De même, l'opération notée $Select_0(i)$ renvoie la position du $i^{\text{ème}}$ bit à 0.

Il existe plusieurs types de structure distinctes, chacune avec leurs particularités. En C++, la *Succinct Data Structures Library* (SDSL) en implémente plusieurs [Gog et al., 2014], à savoir :

- la structure succincte entrelacée (*Interleaved bitvector*),
- la structure succincte éparse/creuse (*Sparse bitvector*),
- la structure succincte RRR (*RRR bitvector*).

À noter que même si dans la SDSL ces structures sont nommées « bitvector », il s'agit bien de structures succinctes qui s'appliquent sur un vecteur binaire (tableau de 0 ou de 1, tableau de bits).

Dans la table 2.2, les différentes complexités des structures succinctes implémentées dans la SDSL sont présentées.

	Espace mémoire	Temps pour les opérations `Rank()`	`Select()`
Éparse	$m\ \left(2 + \log \frac{n}{m}\right)$	$O\left(\log \frac{n}{m}\right)$	$O(1)$
RRR	$\left\lceil \log \binom{m}{n} \right\rceil$	$O(k)$	$O(\log n)$
Entrelacée	$n\ \left(1 + \frac{64}{K}\right)$	$O(1)$	$O(\log n)$

Table 2.2: Complexité des différentes structures de compression pour un vecteur de bits de longueur n avec m bits définis. Le facteur K est le facteur d'entrelacement ce qui signifie que la structure va garder $\frac{n}{K}$ compteurs du nombre de bit à 1 et k est le ratio d'échantillonage du vecteur binaire. Le logarithme est en base 2.

Dans ce tableau, n représente la taille totale du vecteur binaire et m représente le poids du vecteur binaire. Le poids d'un vecteur binaire est le nombre de bits à 1 qu'il contient.
L'absence d'informations sur la structure *hybride* est liée au fait que la structure n'effectuera pas toujours le même type de découpage sur le vecteur binaire. Sa complexité dépendra des données (*cf.* section 2.3.2).

2.3.2 Fonctionnement des différents types de structures succinctes

De manière générale, toutes les structures succinctes commencent par diviser le vecteur binaire en x parties nommées blocs. Ces blocs sont ensuite compressés de façon différente, selon la structure.

Structure succincte hybride

La structure succincte hybride a été pensée pour compresser des données qui, sous forme de vecteurs binaires, possèdent des parties très distinctes. C'est-à-dire que l'on peut diviser le vecteur binaire en bloc de taille variable en fonction de propriétés bien distinctes, comme la répétition d'un bit ou l'alternance régulière de bits. Une fois que les blocs ont été divisés, ils peuvent être encodés différemment. Cela permet d'optimiser la compression d'un bloc par rapport à son contenu [Gog et al., 2014].

Par exemple, si on a x bits à 0 qui se suivent on peut mettre ces x bits dans un bloc. Les n autres bits suivant les x bits sont uniquement des 1, on peut donc les regrouper dans un autre bloc. Si on a une succession de $(01)^{\{p\}}$ (p = nombre de fois où la combinaison 01 est répétée), on peut les regrouper aussi dans un bloc. Une fois les blocs découpés, chacun de ces blocs va être compressé.
Pour cela, chaque bloc est représenté par deux valeurs, un « head », noté h, et un « body », noté b. Le « head » indique le nombre de bits à 1 dans le bloc et le « body » indique la taille du bloc.

Avec la structure hybride, nous avons une division du vecteur binaire en plusieurs blocs. Ces blocs peuvent avoir des tailles différentes. Une fois les blocs déterminés, l'encodage de chaque bloc est choisi indépendemment des autres blocs. La structure hybride a comme avantage d'encoder différemment les blocs et ainsi d'être la plus adaptée à la composition d'un bloc. En revanche, le découpage des blocs et le choix de l'encodage représente un surcoût en temps d'exécution et en mémoire. Il peut être limité en augmentant la taille des blocs.

Structure succincte éparse/creuse

La structure succincte éparse est une structure adaptée pour les vecteurs binaires ayant un faible poids.
Soit un poids maximum de $m \leq \frac{n}{2}$. La valeur m représente le nombre de bits à 1 et n représente la taille totale du vecteur binaire. Ainsi le rapport $\frac{m}{n}$ représente la densité du vecteur binaire et dans ce cas on a bien une densité $\leq \frac{1}{2}$.
La structure divise le vecteur binaire en z blocs de taille b (sauf le dernier bloc qui peut être de taille inférieure), avec $b = 2^{\lceil \log \frac{n}{m} \rceil}$ (et donc $z = \lfloor m \rfloor$).
Pour chaque bloc, seule la position des bits à 1 est conservée. Il y aura, au final, une succession de $\langle x_i, p_i \rangle$ où x_i représente le numéro de bloc ($1 \leq x_i \leq z$) et où p_i représente la position du bit à 1 dans le bloc x_i ($1 \leq p_i \leq b$) [Gog et al., 2014].
L'encodage du bloc x_0 est 1 et l'encodage du bloc x_{i+1} est fait relativement à x_i en utilisant $x_{i+1} - x_i$ fois le symbole 0 suivi du symbole 1. Le nombre de symboles 0 pour coder l'ensemble des blocs x_i est donc borné par z (donc m) et le nombre de symboles 1 est exactement m. Soit au plus $2\,m$ symboles pour coder l'intégralité des x_i. À cela s'ajoute le codage des positions p_i qui requiert $\log \lceil b \rceil = \log \lceil \frac{n}{m} \rceil$ bits par position.

Structure entrelacée

La structure entrelacée est composée d'un vecteur de bits d'origine auquel sont ajoutées des informations pour le classement des éléments. C'est un vecteur de bits qui entrelace le vecteur de bits avec des informations annexes. Cette classe est une représentation vectorielle de bits non compressée. Il copie le vecteur de bits original et entrelace les données en ajoutant une somme cumulative des bits définis avant la position actuelle. Chaque somme cumulée est stockée en un mot de 64 bits. Il faut pour cela connaître la taille du bloc en bits qui doit être une puissance de 2 et supérieure à 64 bits.

Structure succincte RRR

La structure succincte RRR divise le vecteur binaire en blocs de même taille z. Chaque bloc est représenté par deux valeurs. Une valeur c nommée valeur de classe et une valeur o nommée valeur de décalage.

La valeur de classe représente le nombre de bits à 1 dans le bloc et la valeur de décalage indique comment ces bits sont agencés dans le bloc.
Pour utiliser ces deux valeurs, la RRR a donc des tableaux stockés en brut. Le tableau de classe C est composé de $z + 1$ cases, où la case c du tableau de classe renvoie sur un tableau de décalage O_c. Ce tableau de décalage possède $\binom{z}{c}$ cases (pour rappel, c correspond au nombre de bits à 1 dans un bloc).

Grâce à ces tableaux, nous pouvons réduire le bloc i au couple $\langle c_i, o_i \rangle$.
Comme ces tableaux sont les mêmes pour tous les blocs, ils ne sont stockés qu'une seule fois. Cela n'engendre donc pas de surcoût. Cela permet donc, même si les tableaux sont stockés, de compresser le vecteur binaire (le gain de codage porte sur o_i).

2.3.3 Bilan sur les structures succinctes

Le problème de toutes ces structures est qu'elles ne sont pas modifiables : on ne peut pas changer la valeur d'un bit après compression.
Pour modifier une valeur dans le vecteur binaire compressé, il faudrait le décompresser,

modifier le bit en question puis le re-compresser. Cela serait très coûteux en temps. C'est pour résoudre ce problème que RUBIKS a été pensé (détaillé en section 4.3).

2.4 Structures de données d'indexation de textes

Pour rechercher une information précise au sein de données textuelles, une solution naïve consiste à les parcourir en totalité.
Lorsque un grand nombre de requêtes est réalisé, chacune d'entre elle va nécessiter de re-parcourir la totalité des données. Cela devient rapidement coûteux en temps et en ressources computationnelles.

Un *index* est une structure de données permettant de répondre à de telles requêtes, sans avoir à parcourir les données dans leur intégralité. Pour un texte, un index permet généralement deux types de requêtes : compter le nombre d'occurrences d'une suite quelconque de caractères, et localiser chacune de ces occurrences. Un index permet de répondre rapidement à ces requêtes sans avoir à re-parcourir l'intégralité du texte.
Cependant, construire un index a un coût en temps. Sachant que l'index porte sur le texte entier, sa construction nécessite de parcourir le texte intégralement au moins une fois.

L'indexation va donc être utile dès que le texte est connu à l'avance et qu'un nombre suffisamment important de requêtes vont devoir être faites. L'index permet d'avoir des requêtes dont le temps est généralement indépendant, ou dépendant logarithmiquement, de la longueur du texte alors qu'en absence d'index ces temps sont linéaires dans le pire des cas (et souvent dans le cas moyen). Cependant, pour garantir son efficacité, un index est généralement stocké en mémoire centrale et il a un coût non négligeable en espace.

Dans cette section nous nous intéresserons uniquement aux index dits « plein texte ». À l'inverse des listes inversées ou des tables de hachage, par exemple, ces index permettent de chercher un facteur quelconque d'un texte. Nous commençons par présenter la terminologie de cette famille d'index.

Définition 2.1. Un *index succinct* est un index dont la taille est proportionnelle à l'espace théoriquement minimal pour stocker tout texte de même taille et de même alphabet que celui indexé.

Définition 2.2. Un *index compressé*, tire parti de la régularité du texte indexé, il prend un espace mémoire proportionnel à l'entropie empirique du texte.

2.4.1 L'arbre des suffixes

L'arbre des suffixes est une structure d'indexation encore très utilisée, notamment pour la recherche de répétitions dans une séquence génomique mais aussi pour de nombreuses autres applications en algorithmique du texte. En effet, il permet, de rechercher n'importe quel motif, de longueur m, dans un texte de longueur n en temps $O(m)$ [Gusfield, 1997]. Ainsi, la recherche sur un alphabet de taille constante prend un temps qui dépend uniquement de la longueur des motifs.

Définition 2.3. Soit T le texte à indexer, alors l'arbre des suffixes de T doit remplir les deux conditions suivantes :

1. Les feuilles de l'arbre contiennent un numéro qui correspond à la position de début du suffixe dans T.
2. Les branches peuvent être étiquetées de différente manière :

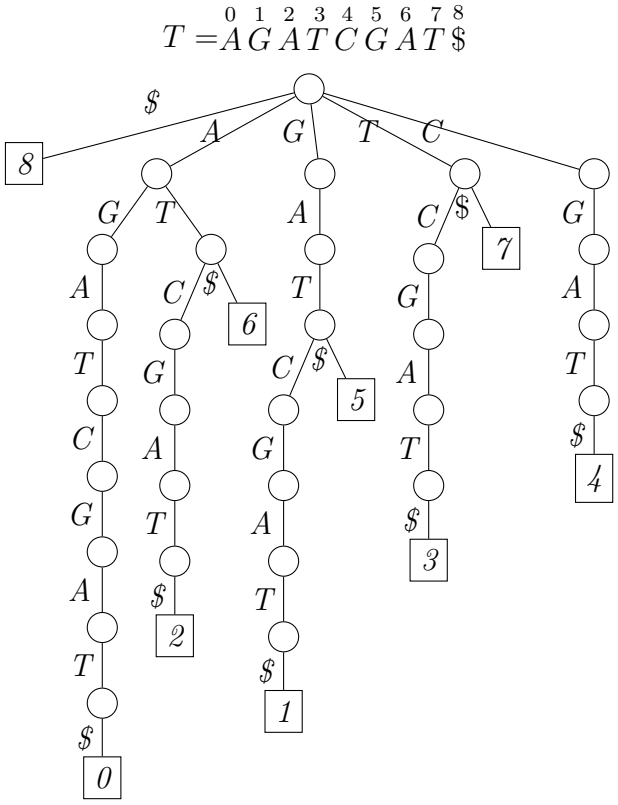
— par une chaîne de caractères de longueur supérieure ou égale à 1.
— par un couple $\langle p, l \rangle$ qui correspond à la sous-chaîne commençant à la position p de longueur l, dans T.

3. La première lettre des étiquettes de chaque branche est différente en sortie d'un nœud de l'arbre.

En général, on ajoute un caractère à la fin du texte pour éviter que certains suffixes ne se terminent sur des nœuds de l'arbre, c'est-à-dire un caractère spécial qui n'est pas présent dans le reste du texte. Dans notre exemple nous utilisons le caractère \$ pour terminer notre texte.

Exemple 2.1 (Arbre des suffixes du mot $T = AGATCGAT$)**.**

Tout chemin partant de la racine (en haut) vers une feuille (représentée par un rectangle) est un suffixe du mot T. La valeur dans une feuille est l'indice de début du suffixe dans le texte.



L'arbre compact des suffixes

L'arbre des suffixes peut être un ensemble de longs chemins unaires le cas échéant tous les nœuds intermédiaires sur ce chemin ont exactement un seul enfant. Cela conduit à un gaspillage de l'espace mémoire à cause du trop grand nombre de branches et de nœuds dans l'arbre.

Dans le pire des cas le nombre de nœuds (incluant les feuilles) peut être quadratique (dans le cas où le texte est composé de caractères tous distincts par exemple). Une telle complexité en espace est incompatible avec l'indexation de grands textes comme une large collection de génome. L'arbre compact des suffixes permet de pallier partiellement à ce problème. Il a été introduit par Weiner [Weiner, 1973].

Définition 2.4. Un arbre compact des suffixes doit respecter les propriétés suivantes :

1. Chaque nœud interne possède au moins deux fils.

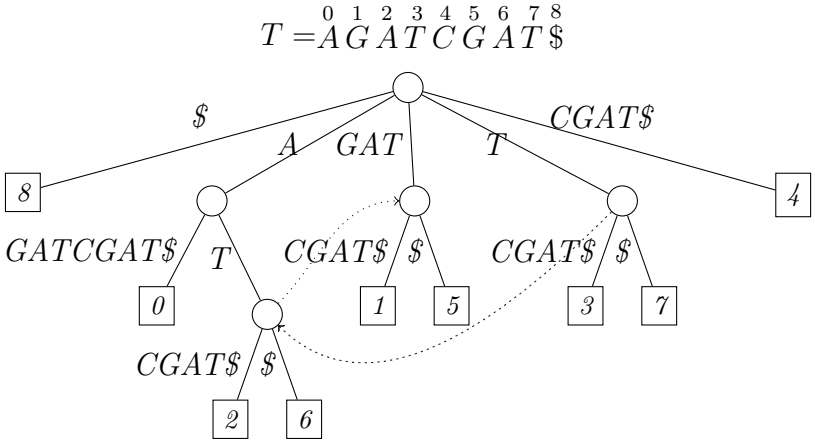
2. Toutes les étiquettes des branches sortantes d'un même nœud commencent par une lettre différente.

Par la suite, l'algorithme de construction des arbres compacts des suffixes a été amélioré avec l'utilisation de liens suffixes.
Ce sont des liens qui vont du nœud représentant le facteur au, avec $a \in \Sigma$ et $u \in \Sigma^*$, au nœud représentant le facteur u, s'il existe.

Dans l'exemple 2.2, les liens suffixes sont représentés par des flèches en pointillés (les flèches dont le nœud destination est la racine ne sont par convention pas représentées).

Nous pouvons observer dans l'exemple 2.2 que les liens suffixes permettent de cerner les parties communes de T (trois sous-arbres possédant des branches avec les mêmes étiquettes). En conséquence, les liens suffixes permettent à la fois de faciliter la construction de l'arbre compact des suffixes et de réduire l'espace mémoire utilisé par la structure.

Exemple 2.2 (Arbre Compact des Suffixes du mot $T = AGATCGAT$)**.**



Les structures permettant de représenter un tel arbre sont linéaires en espace et en temps (sous l'hypothèse raisonnable qu'un mot machine peut être stocké et manipulé en $O(1)$).

L'arbre compact des suffixes permet de compter les occurrences d'un motif en temps optimal $O(m)$ en suivant un principe semblable à celui de l'arbre des suffixes : il suffit de parcourir le chemin partant de la racine et étiqueté par le motif. Retrouver les positions des occurrences du motif se fait également en temps optimal $O(m + occ)$ où occ est le nombre d'occurrences du motif.

Les complexités en temps des requêtes sur l'arbre des suffixes sont optimales, signifiant qu'il n'est pas possible de trouver d'algorithmes asymptotiquement plus rapides en théorie. L'inconvénient de l'arbre des suffixes réside dans l'espace mémoire qui lui est nécessaire : $10n$ octets en moyenne.
Pour remédier à cet inconvénient de l'arbre compact des suffixes, une structure plus économe a été introduite, la table des suffixes.

2.4.2 La table des suffixes

La table des suffixes est une structure de données. Pour un mot donné, le tableau contient une liste d'entiers qui correspondent aux positions de début des suffixes du mot, lorsqu'ils sont triés selon l'ordre lexicographique [Manber and Myers, 1993a]. La table des suffixes d'un mot est donc de même taille que la longueur du mot représenté. Le principal avantage des tables des suffixes par rapport aux arbres de suffixes est qu'en pratique, ils utilisent trois à

cinq fois moins d'espace. Au niveau de la complexité, les tables des suffixes permettent des recherches du type « est-ce que m est une sous-chaîne du mot w ? ». La complexité en temps d'une telle recherche est $O(P + \log N)$, où P est la longueur de m et N est la longueur de w, qui est compétitive avec les arbres des suffixes. Le seul inconvénient est que dans les cas où l'alphabet sous-jacent est fini et petit, les arbres de suffixes peuvent être construits en temps $O(N)$ dans le pire des cas, par rapport au temps $O(N \log N)$ pour les tableaux de suffixes.

Exemple 2.3 (Table des Suffixes du mot $T = AGATCGAT$)**.**

Les suffixes correspondants sont affichés verticalement et sont triés par ordre lexicographique. Seules les valeurs numériques sont réellement stockées, les suffixes ne sont affichés qu'à titre indicatif. SA[3] = 6 ce qui signifie que le 3ème suffixe dans l'ordre alphabétique ($<A<C<G<T) est celui en position 6 dans le texte, c'est-à-dire AT$.

$$T = \overset{0}{A}\overset{1}{G}\overset{2}{A}\overset{3}{T}\overset{4}{C}\overset{5}{G}\overset{6}{A}\overset{7}{T}\overset{8}{\$}$$

8	*0*	*6*	*2*	*4*	*5*	*1*	*7*	*3*
$	*A*	*A*	*A*	*C*	*G*	*G*	*T*	*T*
	G	*T*	*T*	*G*	*A*	*A*	*$*	*C*
	A	*$*	*C*	*A*	*T*	*T*		*G*
	T		*G*	*T*	*$*	*C*		*A*
	C		*A*	*$*		*G*		*T*
	G		*T*			*A*		*$*
	A		*$*			*T*		
	T					*$*		
	$							

2.4.3 Les structures compressées

Il semblerait que les structures construites sur le principe d'enregistrer l'ensemble des suffixes d'un texte ne soient pas la solution la plus appropriée pour enregistrer une grande quantité de données. Récemment des structures d'indexation compressées ont été élaborées afin de pouvoir organiser massivement des informations.

Ce type de structure est essentiellement utilisé dans le haut débit pour l'indexation complète des génomes. Les structures d'indexation compressées qui existent à ce jour sont divisées en deux familles : les index utilisant la compression LZ78 [Ziv and Lempel, 1978] comme par exemple *CHICO* ou *r-index* [Valenzuela, 2016, Gagie et al., 2017] et les index utilisant la table des suffixes tels que *CSA* et *PgSA* [Sirén, 2009, Kowalski et al., 2015].
Dans ces deux cas, cela permet de diminuer considérablement l'espace mémoire requis pour indexer un texte, tout en conservant la possibilité de lire son contenu sans le décompresser.

Le FM-Index [Ferragina and Manzini, 2001] utilise une méthode d'échantillonnage. Le FM-Index est basé sur la transformée de Burrows-Wheeler. Cette méthode permet le chargement et la lecture d'un génome humain en mémoire avec moins de 2 GiB. Cependant, ce gain en mémoire se perd essentiellement en temps avec les requêtes qui prennent plus de temps car certaines données doivent être recalculées.

Une autre piste de recherche dans le domaine des structures d'indexation compressées a consisté à explorer un concept très proche de la table des suffixes, mais qui était uniquement utilisé en compression de données : l'utilisation de la transformée de Burrows-Wheeler [Rosone and Sciortino, 2013] .

2.4.4 La transformée de Burrows-Wheeler

La transformée de Burrows-Wheeler d'un texte consiste en une réorganisation complète de ce texte afin de le compresser. Pour présenter cette transformée, nous commençons par introduire le concept de rotation.

Définition 2.5. Une rotation d'un texte T de longueur n est une chaîne de caractères de même longueur composée d'un suffixe de T concaténé à un préfixe de T. Nous notons $T^{[k]}$ la $k^{\text{ème}}$ rotation du texte T , c'est-à-dire celle dont le préfixe de T est de longueur k.

Exemple 2.4 (Transformée de Burrows-Wheeler du mot $T = AGATCGAT$)**.**

Construction de la transformée de Burrows-Wheeler pour le mot T. Dans un premier temps il faut effectuer toutes les rotations (tableau de gauche). Puis les rotations du mot sont triées (tableau de droite), la transformée de Burrows-Wheeler correspond à la dernière colonne de ce tableau (marquée L).

Rotations

$T^{[0]}$	A	G	A	T	C	G	A	T	$\$$
$T^{[8]}$	$\$$	A	G	A	T	C	G	A	T
$T^{[7]}$	T	$\$$	A	G	A	T	C	G	A
$T^{[6]}$	A	T	$\$$	A	G	A	T	C	G
$T^{[5]}$	G	A	T	$\$$	A	G	A	T	C
$T^{[4]}$	C	G	A	T	$\$$	A	G	A	T
$T^{[3]}$	T	C	G	A	T	$\$$	A	G	A
$T^{[2]}$	A	T	C	G	A	T	$\$$	A	G
$T^{[1]}$	G	A	T	C	G	A	T	$\$$	A

Rotations triées

	F							L
$T^{[8]}$	$\$$	A	G	A	T	C	G	A	T
$T^{[0]}$	A	G	A	T	C	G	A	T	$\$$
$T^{[6]}$	A	T	$\$$	A	G	A	T	C	G
$T^{[2]}$	A	T	C	G	A	T	$\$$	A	G
$T^{[4]}$	C	G	A	T	$\$$	A	G	A	T
$T^{[5]}$	G	A	T	$\$$	A	G	A	T	C
$T^{[1]}$	G	A	T	C	G	A	T	$\$$	A
$T^{[7]}$	T	$\$$	A	G	A	T	C	G	A
$T^{[3]}$	T	C	G	A	T	$\$$	A	G	A

Pour créer la transformée de Burrows-Wheeler d'un texte T, il faut extraire toutes les rotations de T et les trier par ordre lexicographique (*cf.* exemple 2.4). Pour bien visualiser la rotation la case contenant la première lettre du mot T est colorée en rouge.

Pour effectuer une requête $Q = q_1 q_2 \dots q_n$ sur la structure obtenue par la transformée de Burrows-Wheeler d'un texte T, on procède de la façon suivante : en partant de q_n et pour chaque caractère, regarder si il est présent dans la première colonne de la table notée F et que le caractère qui le précède est bien présent dans la dernière colonne de la table notée L.

Exemple 2.5 (Illustration d'une requête $Q = TCGAT$ dans un texte $T = AGATCGAT$)**.**

Exemple d'une requête $Q = TCGAT$ dans un texte $T = AGATCGAT$. Pour une meilleure lisibilité, nous avons numérotée les lettres avec leur position dans le texte ainsi la première lettre porte l'indice 0*, la deuxième* 1 *et ainsi dessuite.*

Texte d'origine	A_0	G_1	A_2	T_3	C_4	G_5	A_6	T_7	$\$_8$
F	$\$_8$	A_0	A_6	A_2	C_4	G_5	G_1	T_7	T_3
L	T_7	$\$_8$	G_5	G_1	T_3	C_4	A_0	A_6	A_2

Dans l'exemple 2.4, si nous souhaitons rechercher le mot $Q = TCGAT$, il faut premièrement regarder la dernière lettre du mot soit la lettre T dans F. Parmi ces lettres lesquelles ont la lettre précédente, le cas échéant la lettre A, dans le mot qui soit présente dans L comme illustré dans l'exemple 2.5. Dans cet exemple, on regarde les lettres T dans F qui ont comme lettre correspondante dans L un A. F donne pour T_3 et T_7 deux rotations soit les lettres A_2 et A_6. On cherche pour ces deux permutations dans F lesquelles donnent un G dans L. On obtient G_1 et G_5, on réitère le processus et la BWT donne A_0 et C_4 pour ces deux

permutations. On recommence avec la permutation C_4 en colonne F. Il reste à vérifier si la rotation associée à ce C_4 en F correspond à la lettre T dans L. C'est le cas et l'on obtient la rotation T_3. La requête à bien été intégralement parcourue, et nous a mené à l'unique rotation T_3. On en conclue que $TCGAT$ a une seule occurrence dans le texte et elle se trouve à la position 3 du texte T.

Nous avons vu différentes structures avec des caractéristiques permettant ou non la compression de données, l'interrogation rapide ou encore la mise à jour de la structure.
Aucune structure réunissant tous ces aspects à la fois n'émerge du lot.
Dans le chapitre suivant, nous allons détailler le vocabulaire et les motivations spécifiques à l'étude des collections de génomes.

Chapitre 3

Du génome de référence au pan-génome

Dans ce chapitre, nous verrons comment l'obtention de plusieurs nouveaux génomes pour différentes lignées d'une même espèce a mis en lumière le fait qu'une unique référence génomique n'était pas suffisante pour capturer l'ensemble des variations génomiques existantes. L'approche intégrative qu'est la pan-génomique a pour but de donner accès à l'ensemble des variations génomiques possibles au sein d'un clade. Il est donc important de présenter les nombreuses avancées concernant les méthodes, les formats de données, et les outils de visualisation pour la pan-génomique.
Actuellement, deux types d'approches sont explorés. La première est une approche en graphes, où chaque génome constitutif est découpé en séquences communes. La seconde est une approche linéaire, dérivée des *Genome Browsers* existants. Une référence pan-génomique est construite, et l'information de présence/absence des séquences pour chaque génome est affichée. L'avantage de cette représentation est qu'elle est plus lisible mais à l'inconvénient d'être moins modulable. Elle nécessite en outre d'avoir une pan-référence disponible.
Ce manque de ressources est en partie lié à une notion assez large de ce qu'est un pan-génome ; pour pallier ce manque, nous présenterons les différentes notions de *core*, *dispensable*, *cloud* et de *shell* au sens large. Nous redéfinirons ces termes pour une utilisation plus en adéquation avec notre problématique et l'approche par k-mers, et aborderons les définitions d'un pan-génome pour se focaliser sur l'ensemble des gènes possibles au sein d'un groupe, alors que d'autres vont aussi considérer les blocs de séquences génomiques partagés ou non entre ces mêmes espèces. L'abstraction du concept influence également les outils de visualisation à disposition. Des *Genome Browsers* classiques prenant une espèce comme référence se montrent insuffisants pour une application aux pan-génomes, et d'autres outils dédiés sont donc nécessaires.
L'accent sera mis sur les outils existants pour la pan-génomique bactérienne, qui se restreignent à une définition génique des pan-génomes et ne permettent pas l'exploration structurale des pan-génomes de plantes.
Nous abordons donc en premier lieu la pan-génomique qui comprend l'étude complète d'un génome pour une espèce souhaitée. Nous redéfinirons ces termes pour une utilisation plus en adéquation avec notre problématique et l'approche par k-mers. Nous aborderons également

comment, au travers d'une structuration et d'une représentation des pan-génomes, il est possible de répondre aux exigences suivantes : comment assurer la construction et la maintenance, l'ajout et la récupération d'informations, la comparaison avec d'autres ensembles de données génomiques et comment permettre la visualisation et le stockage des données.
Nous verrons des méthodes basées sur le stockage d'une séquence de référence et des informations sur chaque variation du génome par rapport à la séquence de référence.

3.1 Pan-génomique

Grâce aux progrès continus des technologies de séquençage, notamment avec l'arrivée des NGS, le nombre de génomes entièrement séquencés ne cesse d'augmenter [Stein et al., 2018, Sherman et al., 2018, Peter et al., 2018]. par voie de conséquence, les séquences complètes de génomes de nombreuses espèces et clades phylogénétiques sont devenues disponibles [O'Leary et al., 2016, Sichtig et al., 2019]. En parallèle, de plus en plus de projets de recherche s'emploient à séquencer un grand nombre de variétés pour une même espèce, souvent dans le cadre d'accords de coopération internationale [The 1000 Genomes Project Consortium, 2015, The 1001 Genomes Consortium, 2016]. L'exploitation des données de séquence dans ces projets à grande échelle nécessite une approche pan-génomique.

En effet, lorsque les génomes de différents individus sont comparés, les données de reséquençage sont généralement alignées sur un génome de référence. Cependant, un seul génome de référence est insuffisant pour représenter toute la richesse génétique des différents individus d'une même espèce, et ignore souvent certains gènes importants, ce qui conduit à une sous-estimation de la diversité génétique [Schatz et al., 2014]. Pour obtenir une carte complète de la diversité génomique, il est crucial de construire un pan-génome incluant tous les variations structurales génomiques, spécifiques de chaque individu [Giordano et al., 2018].

3.1.1 Différentes définitions du pan-génome

Le terme de pan-génome est introduit pour la première fois en 2005 dans une publication de Tettelin *et al.* [Tettelin et al., 2005]. Dans cette publication, la variabilité génétique au sein d'une espèce bactérienne (*S. agalactiae*) est étudiée. Le pan-génome est alors défini comme l'ensemble des gènes présents dans toutes les souches (*core*-genome) et des gènes absents d'une ou de plusieurs souches et des gènes uniques à chaque souche (*dispensable*-genome, *cf.* figure 3.1).

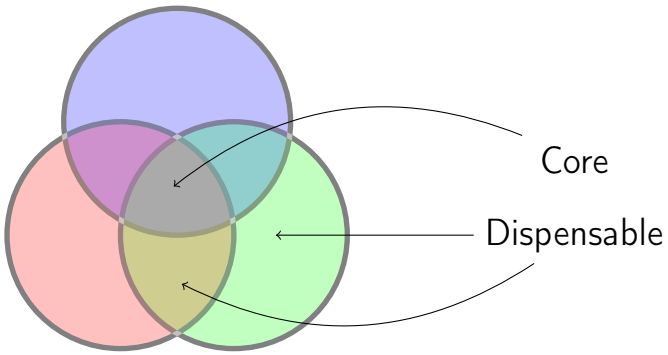


Figure 3.1: La pan-génomique est l'étude de la gamme complète des gènes dans leur diversité au sein d'une même espèce.

Par la suite, d'autres équipes ont nuancé ces catégories, en définissant le *core*-genome (gènes conservés dans presque tous les génomes), le *shell*-genome (gènes de fréquence intermédiaire

potentiellement associés à l'adaptation environnementale), et le *cloud*-genome (gènes à très faible fréquence, *cf.* figure 3.2) [Koonin and Wolf, 2008, Snipen and Ussery, 2010].

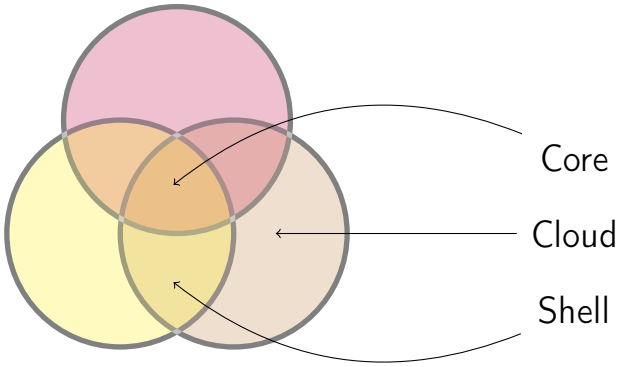


Figure 3.2: La pan-génomique est l'étude de la gamme complète des gènes dans leur diversité au sein d'une même espèce. Le *core* (noyau : gènes toujours présents dans n'importe quel génome du pan-génome), le *shell* (la coque : gènes fréquents) et le *cloud* (nuage : gènes rares).

Plus récemment, la définition étendue considère qu'un pan-génome est un ensemble de séquences génomiques devant être analysées ou servir de référence conjointement, et qui intègre l'information sur les variations structurales génomiques entre les différentes variétés, groupe d'individus ou individus d'une même espèce [Marschall et al., 2018]. L'analyse des pan-génomes de plantes est un domaine d'étude en pleine expansion [Golicz et al., 2016, Yu et al., 2019].

La mise en place d'une structuration et une représentation des données qui prennent en compte l'approche pan-génomique a de multiples avantages par rapport à une simple séquence génomique de référence linéaire dans le cas des projets d'amélioration végétale.
Avoir un pan-génome disponible pour une plante cultivée donnée, incluant des parents sauvages, permet d'avoir un système de coordonnées unique pour ancrer toutes les informations connues sur les variations génomiques (SNP, SV, etc.) et les phénotypes. Ce qui permet l'identification de nouveaux gènes, présents dans les ressources génétiques, et absents dans le ou les génomes de référence.
De plus, le pan-génome révèle des ré-arrangements chromosomiques entre les génotypes, qui peuvent entraver l'*introgression* des gènes désirés. Il fournit également une représentation compacte des génomes *polyploïdes*.

Nous avons fait le choix de préciser que pour nos études et nos représentations, le *core* représente l'ensemble des k-mers partagés par tous les génomes. Le *cloud* représente les k-mers appartenant à un seul génome. Le *shell* est l'ensemble des k-mers qui ne sont ni propres à un seul génome ni partagés par l'ensemble de la collection étudiée.

3.1.2 Structuration et représentation des pan-génomes

Réunies dans un domaine en pleine évolution appelé *pan-génomique computationnelle*, plusieurs méthodes de génération de structures de données pour représenter les pan-génomes ont été développées.

Ces structures de données visent généralement à répondre aux exigences suivantes : (i) construction et maintenance simples, (ii) ajout et récupération d'informations, (iii) comparaison avec d'autres ensembles de génomes ou séquences d'individus, (iv) visualisation aisée, et (v) stockage de données avancé [Herbig et al., 2012].

L'approche la plus fréquente pour traiter la diversité génomique au sein d'une même population consiste à définir un génome de référence unique accompagné d'un catalogue complet de variants et d'autres éléments génomiques variables présents dans cette population [Jandrasits et al., 2018a].
Cette représentation est limitée, car des différences génétiques complexes telles que de grandes suppressions, des insertions ou des ré-arrangements ne peuvent pas être facilement exprimées par rapport à un seul génome de référence [Herbig et al., 2012].

Les structures de données utilisées pour représenter des pan-génomes suivent généralement la même approche, qui consiste à stocker une séquence de référence et des informations sur chaque variation du génome par rapport à la référence. Cela implique que de tels outils nécessitent une information préalable sur les variations génomiques.
Cette information doit avoir été préalablement calculée, par exemple, en effectuant un alignement de plusieurs génomes.

Cela présente un inconvénient important, car une représentation combinée de tout le contenu génomique d'une espèce ou d'une population, qui capture toutes les informations sur la similarité et la variation entre les génomes individuels est essentielle [Marschall et al., 2018].

Par conséquent, le concept plus polyvalent consistant à utiliser plusieurs génomes de référence plutôt qu'un seul pour des analyses communes de données NGS attire de plus en plus l'attention.

3.2 Utilisation des pan-génomes

Certains outils, comme par exemple `bwbble` [Huang et al., 2013] ou bien encore `GenomeMapper` [Schneeberger et al., 2009], se concentrent sur une analyse ultérieure, c'est-à-dire sur les lectures de graphes sur le pan-génome, tandis que d'autres comme `pan genomes indexing` [Dilthey et al., 2015] améliorent la détection de variants en utilisant un ensemble de séquences de références au lieu d'un seul. La dernière catégorie est constituée d'outils qui introduisent une structure de données complète et fournissent des méthodes pour la construction, le stockage, le traitement et la visualisation du pan-génome comme `PanCake` [Ernst and Rahmann, 2013] ou `GenomeRing` [Herbig et al., 2012].

3.2.1 La visualisation des pan-génomes

La visualisation de pan-génomes est une demande importante de la communauté biologique et bioinformatique et de telles visualisations sont généralement réalisées soit de manière linéaire, comme dans les navigateurs génomiques, soit en utilisant une approche circulaire, où les relations entre les régions génomiques sont indiquées par des arcs. Les deux méthodes permettent l'intégration d'informations supplémentaires telles que des données expérimentales ou des annotations.

Dans la figure 3.3 les panneaux supérieurs fournissent une caractérisation statistique du pan-génome et permettent le filtrage des clusters de gènes sur la longueur de gène. Le tableau des groupes de gènes est consultable et triable et permet à l'utilisateur de sélectionner des groupes de gènes individuels pour une inspection plus approfondie. Lors de la sélection dans le tableau, l'alignement du cluster de gènes est chargé dans la visionneuse au centre à droite, l'arbre de gènes est chargé dans la visionneuse d'arbres en bas à droite, et les modèles de présence/absence de ce cluster de gènes sont alignés sur le génome de base et représenté par un arbre en bas à gauche de la figure 3.3.

Figure 3.3: Un exemple de visualisation intéressante généré avec panX [Ding et al., 2017]. Nous montrons ici un exemple de visualisation des données de l'épidémiologie post-vaccinale de 616 souches de *S. pneumoniae* [Croucher et al., 2015]. Malheureusement, cet outil ne fonctionne pas avec nos données (*ran out of memory*) sur un cluster de calcul de type *SGE*.

Un tableau interactif contient un ensemble récapitulatif des statistiques et des annotations pour tous les clusters de gènes. Sous le tableau se trouve un arbre des SNP du génome ainsi qu'un arbre phylogénétique du groupe de gènes actuellement sélectionné. Un second tableau consultable sous les arbres permet un accès rapide aux méta-informations disponibles pour différentes souches et peut être utilisé pour sélectionner des souches dans l'arbre. Une visionneuse d'alignement montre l'alignement des nucléotides ou des acides aminés du groupe de gènes sélectionné dans le tableau. La disposition des différents composants peut être facilement réorganisée.

En bas à gauche se trouve un diagramme circulaire montrant la proportion du génome principal (*core*) et la proportion du génome accessoire (regroupant les gènes fréquents et les gènes rares) (*dispensable*), chacun pouvant être sélectionné en cliquant sur les secteurs du graphique.

Ces fonctionnalités résument les besoins des utilisateurs en matière d'exploration pan-génomiques.

3.2.2 Les outils existants

L'une des structures de données actuellement la plus couramment utilisée pour l'indexation du génome est le FM-index [Ferragina and Manzini, 2001] (*cf.* sous-section 2.4.3). Cette structure d'indexation compressée exploite les propriétés de la transformée de Burrows-Wheeler [Rosone and Sciortino, 2013] et son lien avec la structure de données de table des suffixes [Manber and Myers, 1993b] (*cf.* sous-section 2.4.2), ce qui permet de construire un index génomique linéaire en la taille du génome en temps comme en espace. Pour indexer une collection de génomes similaires, Sirén a proposé de créer un index en utilisant la transformée

de Burrows-Wheeler pour chaque génome de la collection et de fusionner chaque index [Sirén, 2016]. Toutefois, dans cette approche, la mise à jour de l'index semble constituer un obstacle majeur, car elle nécessite une fusion pour chaque génome.

Deorowicz *et al.* ont proposé une méthode efficace pour stocker de grandes collections de génomes [Deorowicz et al., 2013a]. Leur méthode utilise une séquence de référence et un tableau contenant les variations de chaque génome par rapport à un génome de référence.

Ils supposent que les nombreuses variations sont partagées sur l'ensemble des génomes. La structure de données est ensuite compressée, ce qui permet de stocker efficacement un ensemble de séquences génomiques très proches. Leur structure ne peut pas être interrogée et l'extraction d'un génome consiste à décompresser les données et à appliquer les variations indexées à la séquence de référence.

De nouvelles méthodes ont vu le jour à la fois pour le stockage et l'analyse des pan-génomes. Ces méthodes utilisent généralement la même approche, qui consiste à stocker une séquence de référence et des informations sur chaque variation du génome traité par rapport à la référence.

Cela suppose d'avoir les informations préalables sur les variations génomiques. Ces informations doivent avoir déjà été calculées, par exemple, en effectuant un alignement sur plusieurs génomes. Certaines de ces méthodes, telles que `SplitMEM` [Marcus et al., 2014] et `TwoPaCo` [Minkin et al., 2016], utilisent des graphes ou combinent des graphes avec un tableau de suffixes compressés généralisés (GCSA) [Sirén, 2009, Beller and Ohlebusch, 2016, Baier et al., 2016]. D'autres méthodes utilisent une structure de données basée sur les méthodes d'alignement de séquence [Ernst and Rahmann, 2013, Jandrasits et al., 2018b].

L'outil `MuGI` stocke la référence sous forme compacte (4 bits pour coder un seul caractère), une base de données de variants (un vecteur de bits pour chaque variant) et un tableau pour conserver les informations sur chaque k-mer. Cette méthode a été conçue pour indexer un ensemble de génomes humains. Une collection de 1092 génomes humains diploïdes (6,7TiB de séquences brutes) peut être stockée avec 32GiB de RAM [Danek et al., 2014].

Peu de méthodes visent à stocker un pan-génome sans connaissances préalables, et encore moins de méthodes permettent une interrogation directe de ce pan-génome.
Le programme `CHICO` [Valenzuela, 2016] utilise un index hybride qui combine les techniques de compression de Lempel-Ziv avec la transformation de Burrows-Wheeler mais ne passe pas à l'échelle de nos données de riz. Nous avons évalué `CHICO` sur un ensemble de 10 génomes (53GiB) sous forme de fichiers FASTQ compressés extraits du projet 3000 génomes de riz [Li et al., 2014] ; le résultat est un dépassement de la mémoire attribuée soit 326GiB.

`BFT` est utilisé pour indexer et interroger différents ensembles de données de pan-génomes. C'est une nouvelle structure de données succincte appelée Bloom Filter Trie pour indexer un pan-génome sous la forme d'un graphe de de Bruijn coloré.
La méthode `BFT` est basée sur une structure de données qui se construit sans alignement, sans référence et qui permet de stocker un pan-génome en tant que C-DBG [Holley et al., 2016]. La structure de données permet de stocker et de compresser un ensemble génomes en utilisant une approche par k-mers dit colorés, et également de parcourir efficacement le graphe.
Les k-mers sont dit colorés parce que l'arbre des suffixe (trie) stocke les k-mers et ainsi que leur provenance (leurs couleurs). `BFT` associe à chaque k-mer un vecteur binaire coloré. Les couleurs représentent les génomes dont est issu le k-mer. Le fait de stocker les k-mers et leur provenance dans les sommets des graphes, cela compresse et indexe les sous-chaînes communes.

Les sommets utilisent des tableaux pour le stockage de ces sous-chaînes ainsi que des filtres de Bloom sous-section 2.1.3 pour permettre des requêtes efficaces.

De nombreux outils existent pour indexer une collection de génomes, pour les requêter, pour mettre en évidence leurs variations, mais aucun de ces outils ne rassemblent tous les aspects nécessaires dans l'analyse de grandes collections de génomes à savoir, être capable d'indexer des volumes des données conséquents, être capable de requêter efficacement, être capable de combiner de l'information de nombreux génomes.

Deuxième partie

Résultats

Dans cette partie seront présentées nos différentes contributions. Dans un premier chapitre les travaux préliminaires sont présentés : l'indexation au sens large, l'ensemble des travaux qui nous mené à la conception de notre outil d'indexation. Dans un second chapitre l'indexation d'un génome avec *gkampi*. Dans un troisième chapitre l'indexation d'une collection de génomes avec *RedOak*.
Finalement dans une dernière partie les résultats biologiques obtenus permettant une validation.

Chapitre 4

Travaux préliminaires

Dans ce chapitre est introduit le cheminement qui nous a mené à la structure d'indexation que nous avons implémentée. Dans un premier temps, nous rappellerons les besoins qui nous ont poussé à chercher dans cette direction, puis les solutions que nous avons envisagées pour pallier les problèmes que nous avons rencontrés, et enfin les solutions déjà existantes.

L'indexation de manière générale se divise en 2 étapes. L'analyse et la sélection ; l'analyse pour apprendre à connaître le texte ou les textes à indexer et la sélection pour ne garder que les éléments qui vont nous permettre d'indexer nos textes.

Pour ce faire nous illustrerons dans un premier temps une étude qui a permis l'analyse combinatoire des k-mers issus de séquences d'une collection de génomes de riz, les 8 premiers génomes de riz assemblés et disponibles en 2016.
Cette étude préliminaire nous mène sur la deuxième partie de ce chapitre et qui porte sur notre premier outil *LEGO*, *LEGO* pour whoLE Genomes cOmparaison. Cet outil est basé sur les fichiers de k-mers obtenus à partir de *Jellyfish* et permet de créer des diagrammes de Venn qui sont des diagrammes illustrant les intersections entre ensembles. Ces ensembles sont le cas échéant un ensemble de k-mers et nous avons montré que les génomes de riz avaient un *core* k-mers important et donc un fort taux de compression potentiel, parce que les k-mers qui reviennent plusieurs fois n'auront à être stockés qu'une fois.
Nous présenterons ensuite *PICTURES* un outil implémenté en C qui permet, étant donnée une matrice de k-mers produite à partir de *Jellyfish*, soit de créer une image, soit d'utiliser une librairie R nommée UpSetR et de créer des diagrammes de Venn. PICTURE est disponible sur le GitLab de *South Green*, plateforme bioinformatique des partenaires du projet *GenomeHarvest* :

`https://gitlab.southgreen.fr/ALICIA/pictures.git`

Les images produite avec *PICTURES* nous ont illustré à quel point nos données étaient éparses. Nous verrons finalement comment nous avons imaginé un algorithme permettant de compresser des données très éparses : *RUBIKS* pour *RRR Update for Bit Indexing k-mer Structure*.

Pour mener à bien les recherches d'une telle structure, nous avons suivi une démarche exploratoire. La première étape a consisté en une phase d'analyse combinatoire de nos données.
Cette étape a mis en évidence la manière dont les données devaient être structurées. Après une analyse de la combinatoire de nos données et du facteur de compression qu'il était permis d'envisager, nous avons envisagé l'adaptation et l'optimisation de librairies comme la SDSL [Gog et al., 2014] permettant la compression de données.

4.1 Analyse combinatoire des k-mers d'une collection de génomes de riz

Dans le cadre de cette analyse préalable, l'ensemble des données générées n'est pas manipulable dans sa globalité. Il est possible de prendre les génomes les uns après les autres ou d'en prendre un échantillon représentatif.

4.1.1 Présentation des données

Dans une récente analyse [Zhao et al., 2018], les auteurs ont sélectionné 66 accessions de riz du genre *Oriza* (principalement *O. sativa*) correspondant à des données de séquençage non assemblées. Les auteurs les ont assemblées puis comparées avec un des génomes de référence pour *O. sativa*, le *cultivar* Nipponbare (*cf.* section 1.5).
Ce travail a permis l'identification de 16 563 789 SNP (Single Nucleotide Polymorphism), 5 549 290 petites insertions et suppressions (*indels*) inferieures à 20pb et 933 489 variants structuraux qui font référence à des *indels* importants allant de 20pb à 12Kpb.

L'ensemble des travaux, études et résultats présentés dans ce chapitre ont été réalisés à partir des génomes présentés dans la table 4.1.

Génome			Séquences		Taille du
N°	cultivar	*ssp.*	nombre	taille totale	fichier
1.	93-11	*indica*	12 chromosomes + 12 718 scaffolds	423 026 874pb	412MiB
2.	BABO01	*japonica*	12 chromosomes + 654 543 contigs	306 177 972pb	390MiB
3.	DJ 123	*aus*	2 819 scaffolds	345 981 746pb	335MiB
4.	IR 64	*indica*	2 919 scaffolds	345 209 449pb	334MiB
5.	Nipponbare (IRGSP-1.0)	*japonica*	12 chromosomes	373 245 519pb	362MiB
6.	Kasalath	*aus*	12 chromosomes + 1 scaffold (14 822 contigs concaténés et séparés par 1 000 `N`)	401 141 708pb	421MiB
7.	LNNJ01.1	*indica*	237 scaffolds	346 854 256pb	336MiB
8.	LNNK01	*japonica*	181 scaffolds	359 918 891pb	349MiB

Table 4.1: Liste des génomes utilisés pour l'étude préliminaire.

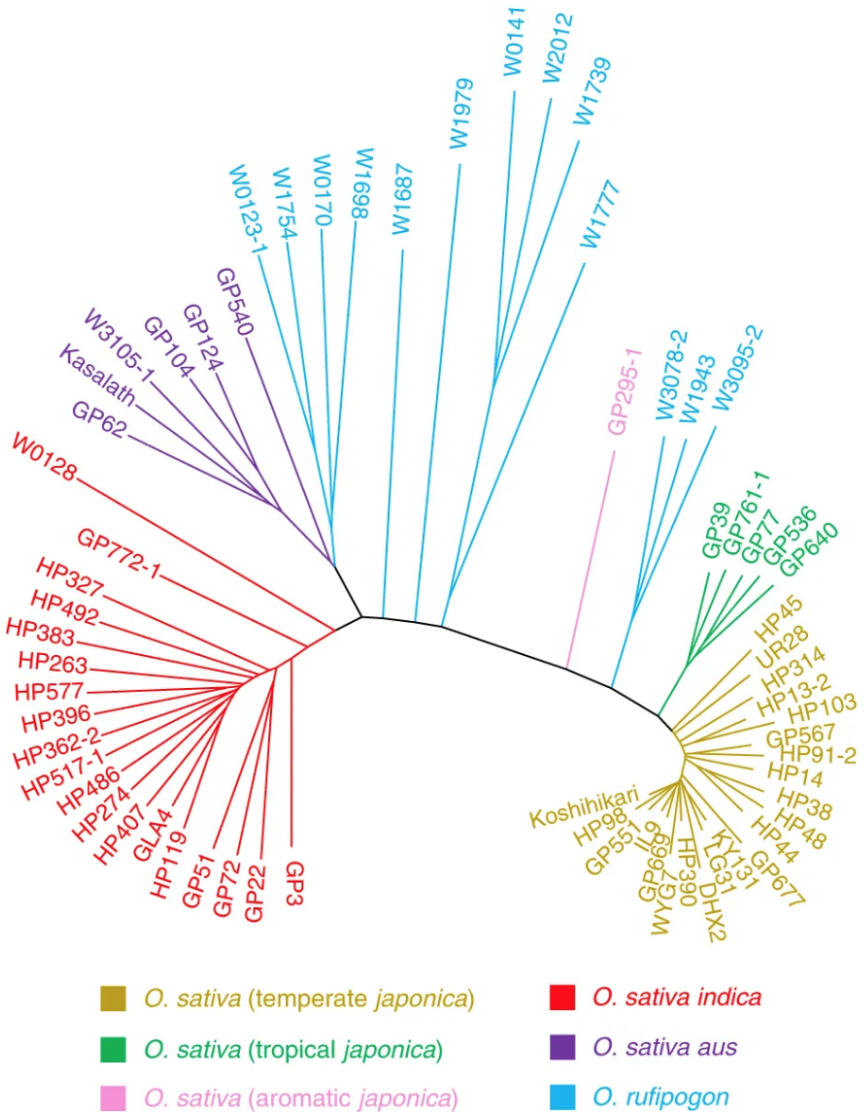


Figure 4.1: Arbre phylogénétique des 66 accessions de riz utilisant les matrices de distances calculées à partir des données de génome assemblées. Les accessions au sein de différents groupes sont indiquées par des couleurs différentes (source : [Zhao et al., 2018]).

4.1.2 Indexabilité

Deux concepts importants doivent être définis avant de pouvoir aborder la représentation des génomes de riz. Le premier concept est celui de l'*indexabilité*.

Définition 4.1 (Indexabilité)**.** L'indexabilité est la capacité d'un texte fini à être réorganisé dans une structure de données finie.

Le second concept qu'il est nécessaire de définir est celui d'*indexation automatique*.

Définition 4.2 (Indexation automatique)**.** L'indexation automatique des mots dans un document est la création de la liste ordonnée de tous les mots apparaissant dans le document avec la localisation de chacune de leurs occurrences.

Un texte T de taille m sur un alphabet de σ lettres équiprobables aura dans le pire des cas $N = m-k+1$ k-mers donc aura besoin de $N \log_2(\sigma)$ bits pour stocker le texte [Shannon, 1948].

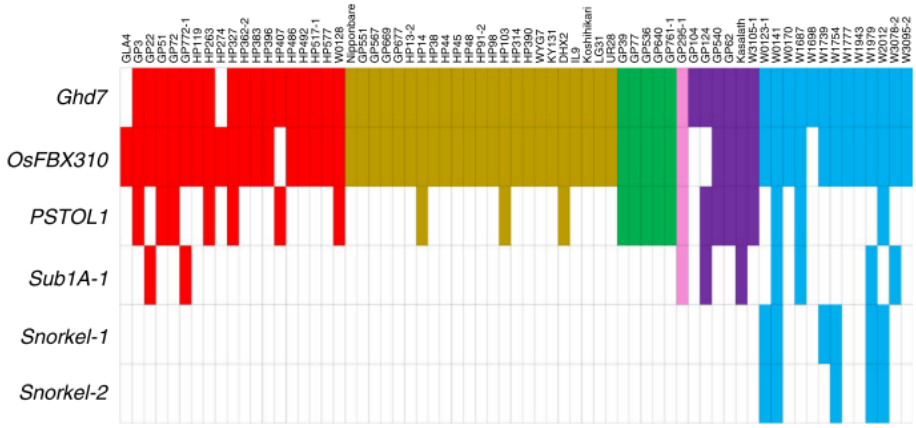


Figure 4.2: PAV de six gènes dans les 67 génomes. Les accessions au sein de différents groupes sont codées par couleur comme sur la figure 4.1. L'absence d'un gène dans le génome est indiquée par une case vide.

Il faut rajouter à cela la structure d'indexation. La structure d'indexation dans le cas naïf, qui consiste à avoir nos données et un tableau contenant les positions de l'ensemble de nos données $N \log_2(N)$, double nos données donc on aura besoin de $(\log_2(\sigma + N))$ bits.

Lorsque l'on souhaite indexer un texte, on cherche à connaître le langage associé, car le langage nous informe sur la fréquence d'apparition des lettres et donc des mots.

Le problème majeur avec l'indexation automatique c'est que le texte est indexé entièrement et que des mots à haute fréquence comme « le », « un », « ou », *etc.* vont prendre énormément de place pour le peu d'informations qu'ils véhiculent. Il est donc judicieux de ne pas les indexer. Il n'est pas nécessaire de garder les suffixes des mots. Si le préfixe est suffisamment discriminant alors lors de la création de l'index il est préférable de ne garder en mémoire que le préfixe au lieu du mot entier (préfixe + suffixe).

Nous avons donc cherché à savoir s'il était possible d'indexer notre texte de manière automatique et pour ça nous avons envisagé une approche par k-mers pour déterminer l'indexabilité de nos données.
LEGO est un outil implémenté en python dans le package ALICIA (Agret LIrmm CIrAd). *LEGO* est la première contribution développée au cours de cette thèse. C'est un outil qui prend un ensemble de génomes assemblés et qui, en faisant appel à la librairie *Jellyfish* [Marcais and Kingsford, 2011], permet de créer des diagrammes de Venn des ensembles de k-mers des séquences fournies en entrée.

4.1.3 *LEGO* : un outil pour visualiser l'organisation ensembliste des k-mers

Un diagramme de Venn est un diagramme qui montre toutes les intersections possibles dans une collection finie de différents ensembles.
Dans le cas des ensembles de génomes, à l'échelle de la génomique, le diagramme de Venn nous montre les gènes partagés par tous les génomes, ceux qui sont partagés partiellement et ceux qui sont propres à un génome seulement. Dans notre cas, ce qui nous intéresse ce sont les k-mers qui représente les mots que l'on souhaite indexer.
Dans le cas des ensembles de génomes, le diagramme de Venn est souvent utilisé afin

de représenter les ensembles de gènes partagés par tout ou partie des génomes. Ici, nous généralisons cette représentation à l'ensemble des k-mers des génomes.

LEGO permet de générer rapidement des diagrammes de Venn à partir de k-mers issus de génomes à partir des comptages calculés par *Jellyfish* (*cf.* section 2.2). Les ensembles sont des mots de longueur k et les ensembles de fichiers contenant les k-mers et leurs comptes sont noté JF. Pour chaque mot on incrémente la combinaison de génomes qui possède le k-mer de la valeur d'occurrence minimale non nulle. Par exemple, supposons que nous étudiions 4 génomes et qu'un k-mer donné est présent 38 fois dans le premier génome, 0 fois dans le second génome, 44 fois dans le troisième génome et 7 fois dans le dernier génome. Alors la variable associée est v_{1011} et prendra la valeur minimale de l'ensemble $(38, 44, 7)$ soit 7.
Pour 4 génomes le nombre de variable est de $2^4 - 1 = 15$. La combinaison 0000 ne peut pas exister parce que tous les k-mers dans JF appartiennent à au moins un génome. Les combinaisons $1000, 0100, 0010, 0001$ représentent le *cloud* respectivement du premier, du second, du troisième et du dernier génome. La combinaison 1111 représente le *core* et les 10 autres combinaisons représentent le *shell*.

algorithme 4.1: High level algorithm *LEGO* : to create Venn Diagram from *Jellyfish* files

```
Input :
  JF = {JF_1, ··· , JF_n} % A set of n Jellyfish count files %
Output :
  MJF % A two column matrix: set / size of the set %
Begin
  For Each K ∈ {w | w is a k-mer of JF}
    ValBin ← ""
    occ ← 0
    For i ← 1 To n
      cpt ← nb_occ(K, JF_i)
      ValBin ← ValBin · ((cpt > 0)?"1":"0")
      occ ← min(cpt, occ)
    End
    MJF_ValBin ← MJF_ValBin + occ
  End
  Print(MJF)
  Return 0
End
```

La figure 4.3 illustre assez clairement la notion de *core*, de *shell* et de *cloud* k-mers introduite par la figure 3.2. On observe clairement que le génome DJ123 se révèle être plus distant en contenu de k-mers. Après analyse, il se trouve que l'assemblage produit par les auteurs de [Zhao et al., 2018] pour ce génome est de moins bonne qualité (il s'agit d'ailleurs d'un assemblage partiel). Il est fort probable que la distance de ce génome par rapport aux autres soit corrélée à la mauvaise qualité de son assemblage.

Toujours en appliquant *LEGO*, mais cette fois ci sur des génomes de plantes différentes (un génome de riz et deux versions sucessives de génomes de bananiers), il est possible d'observer que le nombre de *core* k-mers est insignifiant et que la quasi totalité des *shell* k-mers sont sur les deux versions du génome de bananier (figure 4.4). Ceci tend également à montrer que l'approche par k-mers permet de capter la spécificité des génomes d'une espèce et montre également l'impact de la qualité de l'assemblage.

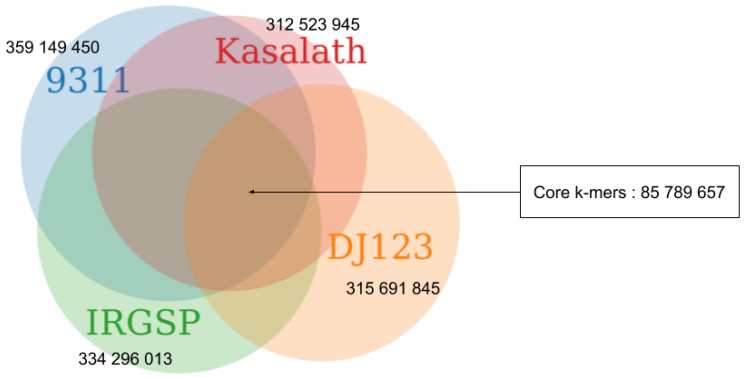


Figure 4.3: Diagramme de Venn des génomes IRGSP (Nipponbare), Kasalath, 9311 et DJ123. Les génomes ont été découpés en k-mers de taille 21, et comparés grâce à *LEGO*.

k-mer	Memoire RAM (GiB)	Temps (h :m :s)
15	5.918	01 :41 :55
45	19.151	03 :33 :37

Table 4.2: Performance de *LEGO*

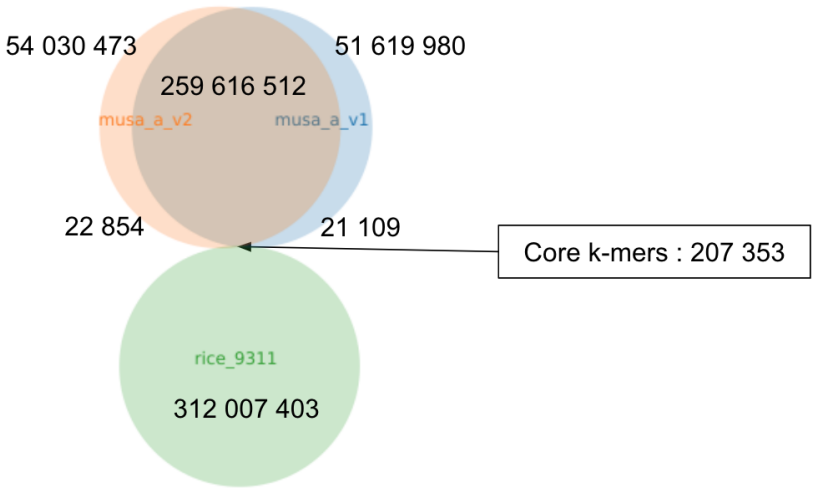


Figure 4.4: Diagramme de Venn des génomes du riz 9311 et des versions 1 et 2 du génome de bananier *Musa acuminata*. Les génomes ont été découpés en k-mers de taille 21, et comparés grâce à *LEGO*.

4.2 Représentation de la matrice de présence/absence avec *PICTURES*

La limitation de l'approche *LEGO* est qu'il est difficile d'introduire plus de quatre génomes, le graphique devient rapidement illisible. Nous avons donc développé un autre outil en C, *PICTURES* qui prend des fichiers de comptage de k-mers triés générés avec *Jellyfish*.

Une fois les fichiers de comptage générés avec *Jellyfish*, *PICTURES* crée une matrice globale, chaque ligne est un k-mer et chaque colonne représente un génome. Si le k-mer est présent dans la colonne alors *PICTURES* l'indique avec un 1 si il est absent il l'indique avec un 0. Cette matrice peut être visualisée sous forme d'une image *bitmap* ou bien être analysée par UpsetR [Conway et al., 2017].

4.2.1 *PICTURES* et UpsetR

UpsetR est un package R qui, à partir d'une matrice de présence/absence, génère un diagramme de Venn revisité. Ce diagramme est représenté sous forme d'un histogramme en deux dimensions, avec en abscisse l'ensemble des sous-ensembles de génomes possibles et en ordonnée la taille (en nombre d'éléments) de l'intersection du sous-ensemble. Les barres de l'histogramme sont triés par taille décroissante (donc de la plus grande intersection à la plus petite). Pour modéliser les différentes combinaisons de sous-ensembles, chaque élément est représenté par une ligne sous l'axe des abscisses avec une pastille noire si l'élément est dans le sous-ensemble et une pastille gris clair sinon. Les pastilles correspondant aux éléments du sous-ensemble sont reliées par des traits noirs. Il est ainsi possible d'identifier rapidement des sous-ensembles d'intérêt. Enfin, les éléments sont superposés selon leur dimension (les plus grands ensembles sont les plus bas).

Le diagramme généré par *PICTURES* est donc trié par taille décroissante des k-mers communs aux sous-ensembles, donc la première barre représente le plus gros sous-ensemble. Dans la figure 4.5 le plus gros sous-ensemble est l'ensemble des k-mers qui appartiennent seulement au génome 3, donc le génome DJ123, le second au génome 4, donc le génome IR64 (*cf.* table 4.1) et le troisième aux k-mers appartenant à DJ123 et IR64. Il se trouve que les assemblages de ces deux génomes sont incomplets et nous les présumons de ce fait également bruités. Il est possible d'observer que les *core* k-mers puis les *shell* k-mers communs à tous les autres génomes autre que DJ123 et IR64 sont les sous-ensembles en position 4 à 7. Cette expérience et cette visualisation permettent clairement une première analyse de la distance entre les génomes ou bien de la qualité des génomes considérés.

4.2.2 *PICTURES* et *bitmap*

La matrice de présence-absence peut être également considérée comme la représentation d'une image en noir et blanc, non compressée.

En choisissant arbitrairement que la valeur 0 code un pixel blanc et la valeur 1 code un pixel noir, il est possible d'obtenir une image telle que la largeur de l'image soit le nombre de génomes considérés et que la hauteur de l'image soit le nombre de k-mers qu'il est possible de créer. Malheureusement, on s'aperçoit assez intuitivement que pour représenter la matrice de 8 génomes avec $k = 21$, on obtient une image de 8 pixels de large par $4^{21} \simeq 4,4 \cdot 10^{12}$ pixels de haut, ce qui rend la visualisation peu aisée.

Une solution consiste donc à découper cette image en « bandelettes » de même hauteur et à les juxtaposer afin de réduire drastiquement la diagonale de l'image. Trivialement, pour obtenir

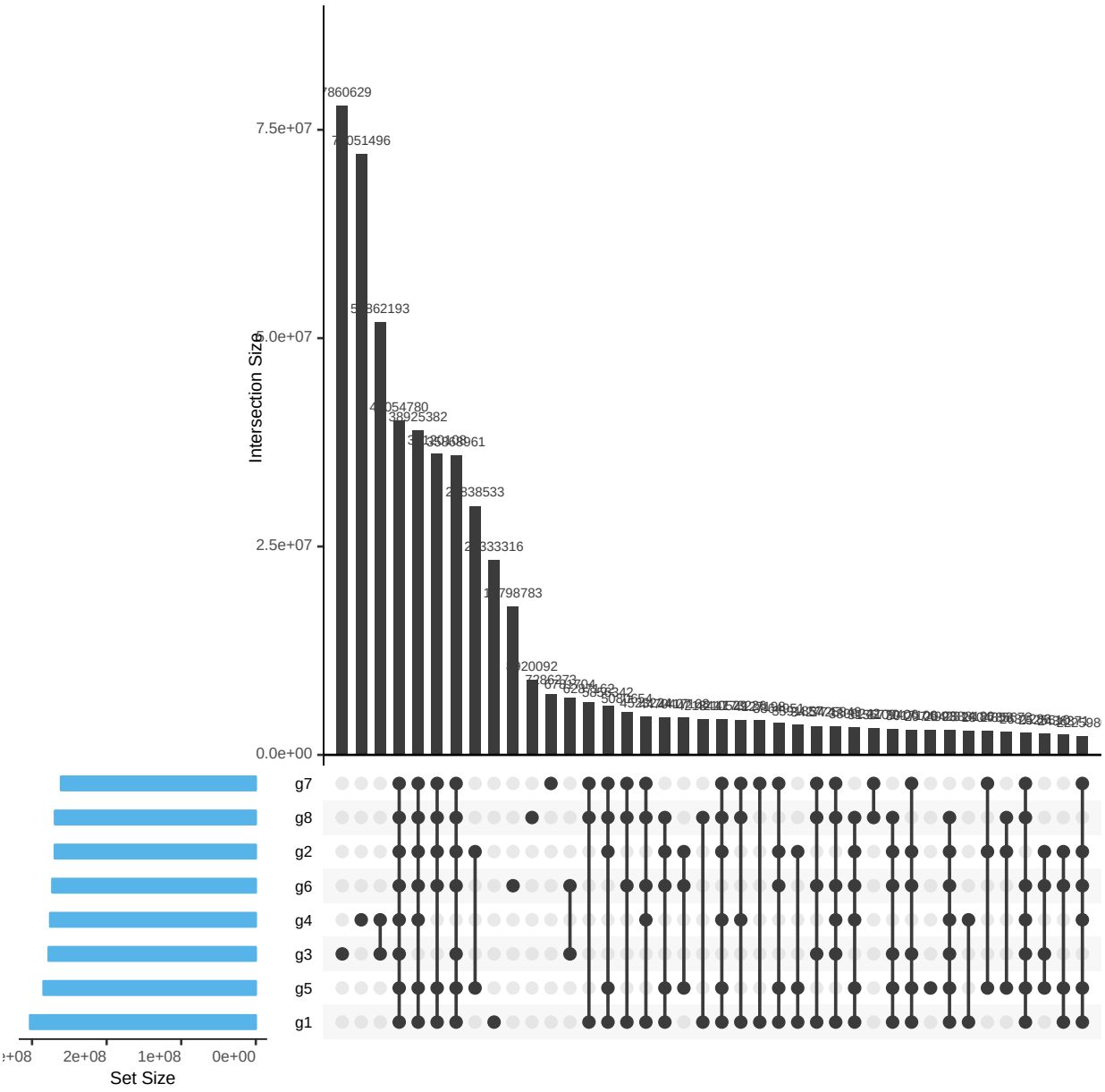


Figure 4.5: Comparaison d'un ensemble de huit génomes, ne pouvant pas être représenté dans un diagramme de Venn classique. Les 8 génomes ont été découpés en 21-mers avec *Jellyfish*.

une image complète, le nombre de bandelettes doit être une puissance de 2 or le nombre de k-mers est une puissance de 4. Donc un moyen assez simple pour réaliser ce découpage consiste à considérer qu'une bandelette est l'image produite pour tous les k-mers partageant un même préfixe de longueur $k_1 < k$. L'image ainsi produite sera donc de dimension $N \times 4^{k_1}$ pixels de large par 4^{k-k_1} pixels de haut (avec N le nombre de génomes).

Pour obtenir une image au ratio ρ, il suffit donc de choisir comme valeur de préfixe, la valeur

$$k_1 = \frac{2\,k + \log_2 \rho - \log_2 N}{4} \quad .$$

Calcul du préfixe permettant d'avoir une image au ratio ρ avec PICTURES .

Le ration ρ est défini comme le rapport de la largeur sur la hauteur, soit :

$$\begin{aligned}
\rho &= \frac{N \times 4^{k_1}}{4^{k-k_1}} \\
N \times 4^{k_1} &= \rho \times \frac{4^k}{4^{k_1}} \\
4^{2\,k_1} &= \rho \frac{4^k}{N} \\
2\,k_1 \log_2 4 &= \log_2 \rho + k \log_2 4 - \log_2 N \\
4\,k_1 &= 2\,k + \log_2 \rho - \log_2 N \\
k_1 &= \frac{2\,k + \log_2 \rho - \log_2 N}{4}
\end{aligned}$$

Pour obtenir un ratio proche de ρ, il faut donc choisir la valeur de k_1 proche de $\frac{2\,k+\log_2 \rho-\log_2 N}{4}$. □

L'image de la figure 4.6 a été générée avec un ratio de 5 : 8 sur les 8 génomes de riz. Cette visualisation permet de constater que la densité de la matrice est très faible. Dans cet exemple, la densité est d'environ $0,82\%$. Cependant, on peut y distinguer des motifs non uniformes, ce qui correspond à des zones permettant potentiellement d'envisager des compressions, par exemple de type *quadtree* [Finkel and Bentley, 1974].

La représentation sous forme d'image devient difficilement exploitable dès que le nombre de génomes devient important en raison de la taille des images décompressées.
Toutefois, une représentation sur le modèle des *quadtrees* demeure judicieuse. Pour rappel, le *quadtree*, dans sa forme la plus simple, est un arbre quaternaire où chaque nœud (racine, nœud interne ou feuille) est associé à une zone de l'image. Si la zone est intégralement blanche ou noire, alors le nœud est une feuille respectivement blanche ou noire. Dans le cas contraire, le nœud est un nœud interne à quatre fils, où chaque fils représente respectivement le quart nord-ouest, le quart nord-est, le quart sud-est et le quart sud-ouest de la zone de l'image associée au nœud interne.
L'idée est de représenter une ligne ou une colonne de l'image comme un vecteur binaire. Les structures succinctes présentées dans la section 2.3 offrent l'avantage de pouvoir représenter des vecteurs binaires en utilisant le moins de mémoire possible tout en disposant d'un accès efficace en consultation au contenu des cases du vecteur. Leur principal inconvénient est que ces structures ne permettent pas la mise à jour dynamique d'un bit quelconque du vecteur puisqu'il faut recalculer toute la structure.
Nous avons donc imaginé de combiner la flexibilité offerte par les *quadtrees* avec l'efficacité des structures succinctes représentant des vecteurs binaires ; avec pour objectif d'encoder, à partir des algorithmes des structures succinctes implémentées pour les données éparses, une méthode pour les données très éparses.

4.3 *RUBIKS : RRR Update for Bit Indexing in K-mer Structure*

RUBIKS est un concept de compression, une surcouche de la RRR pour permettre la notion de mise à jour (modification de la structure). *RUBIKS* n'a pas été implémenté, son algorithme a été présenté à seqbio2018.

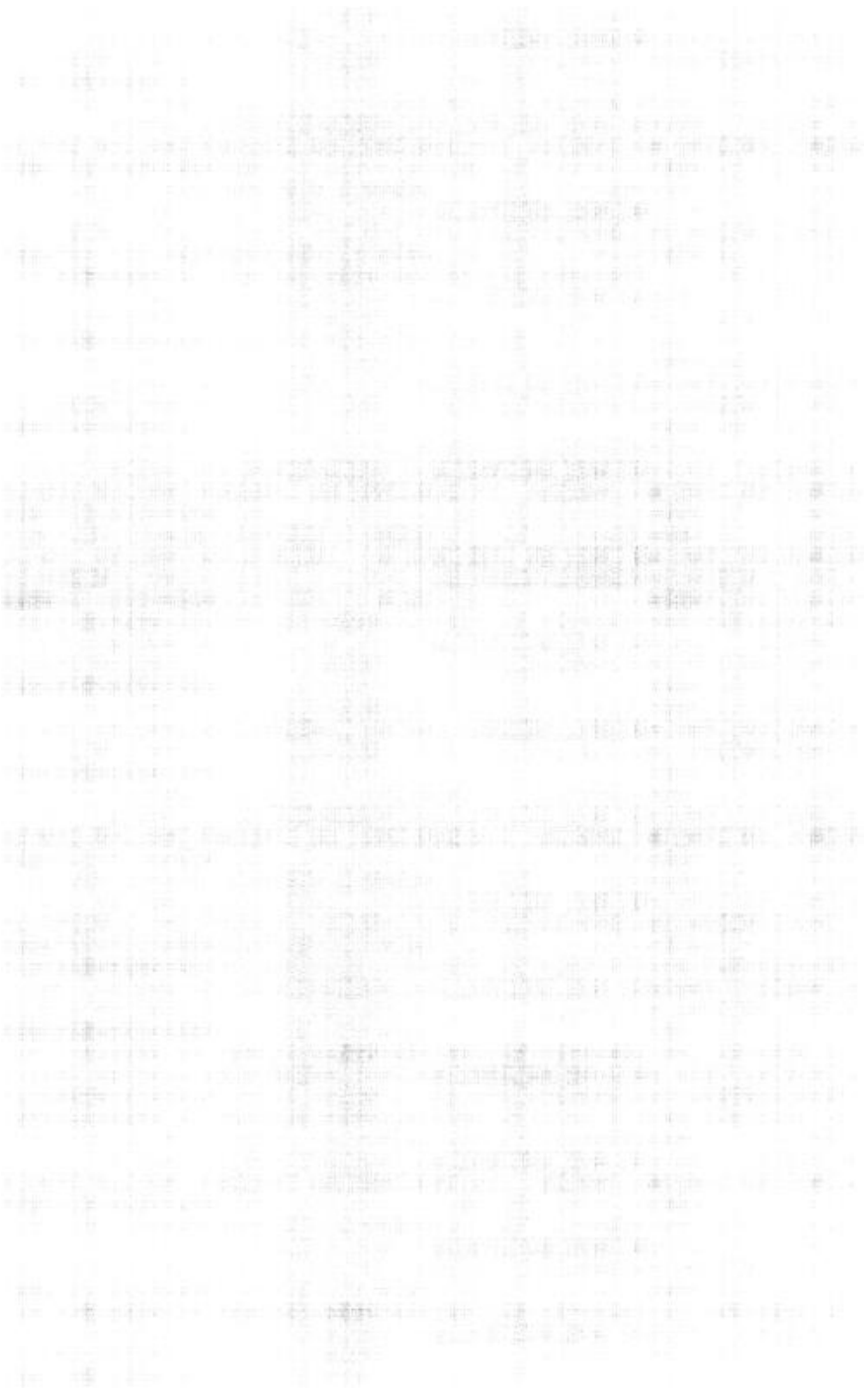


Figure 4.6: Résultat de l'image produite par *PICTURES* à partir des 8 génomes de l'étude préliminaire.

Le principe de fonctionnement de *RUBIKS* (*cf.* figure 4.7) est de diviser le vecteur binaire en blocs de n bits. À chaque bloc est associée une adresse mémoire. Si un bloc ne contient que des 0, alors il est associé à l'adresse mémoire `NULL`. Dans le cas contraire, le bloc est compressé sous la forme d'une RRR et est associé à l'adresse mémoire de cette représentation.

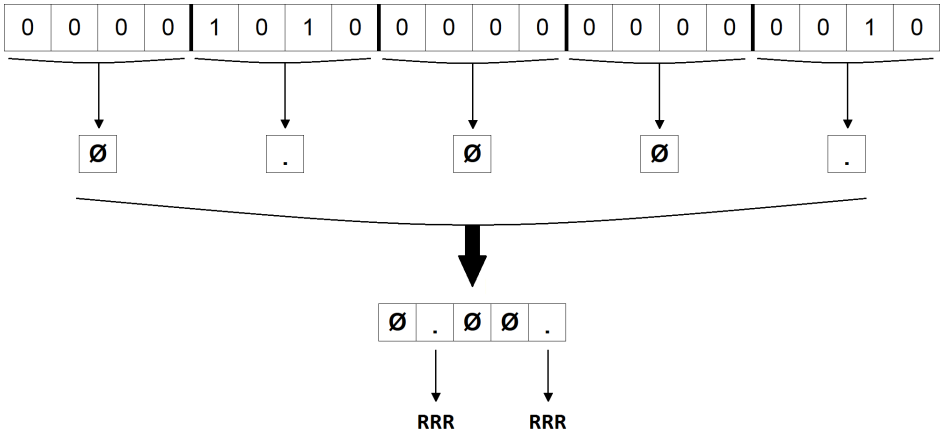


Figure 4.7: Compression hybride

Cette structure est plus facilement modifiable qu'une RRR. Par exemple, dans le cas d'une modification d'un bit à 0 en un bit à 1, ce n'est pas toute la structure qui devra être décompressée, mais uniquement le bloc concerné par la modification.
S'il y a, après modification, un bit à 1 dans un bloc associé à l'adresse `NULL`, il suffit alors de remplacer l'adresse `NULL` par l'adresse d'un nouveau bloc RRR classique, sans avoir à faire une décompression et une recompression de toute la structure.
Si l'on modifie un bloc étant déjà compressé sous forme d'une RRR, il suffit juste de ré-appliquer une RRR sur ce bloc. Enfin, si l'on obtient avec un bloc RRR uniquement composé de bits à 0 après la modification, on remplace ce bloc RRR par un bloc `NULL`.

Pour mémoire (*cf.* table 2.2), pour un vecteur binaire de taille n ayant m bits à 1, la structure RRR associée requiert $m\left(2+\log_2 \frac{n}{m}\right)$ bits. En notant $d=\frac{m}{n}$ la densité du vecteur à représenter, on obtient alors (dans un modèle de Bernouilli) que la probabilité qu'un bit soit à 1 est d et donc la probabilité qu'un bit soit à 0 est $1-d$. La probabilité que tous les bits d'un bloc de taille N soient à 0 est donc de $(1-d)^N$ et la probabilité qu'au moins un bit du bloc soit à 1 est de $1-(1-d)^N$. En choisissant de représenter le vecteur avec *RUBIKS*, avec des blocs de taille N (donc avec $\lceil \frac{n}{N} \rceil$ blocs) et en notant w la taille d'un mot mémoire, on obtient une structure nécessitant :

$$\begin{aligned}&\sum_{i=1}^{\frac{n}{N}} \overbrace{(1-d)^N w}^{\text{blocs de 0}} + \overbrace{\left(1-(1-d)^N\right)\left(w+d\,N\left(2+\log_2\frac{N}{d\,N}\right)\right)}^{\text{blocs contenant des 1}} \text{ bits}\\ =&n\left(\frac{w}{N}+\left(1-(1-d)^N\right)\,d\,(2-\log_2 d)\right) \text{ bits}\end{aligned}$$

En comparaison, utiliser la RRR pour représenter le vecteur binaire entier requiert un espace de $d\,n\,(2-\log_2 d)$ bits.
Aussi, il est possible de montrer que la représentation *RUBIKS* requiert nécessairement plus

de mémoire que la représentation RRR. En effet,

$$
\begin{aligned}
& d\ (2-\log_2 d) - \left(\frac{w}{N} + \left(1-(1-d)^N\right)\ d\ (2-\log_2 d)\right) > 0 \\
\Leftrightarrow\quad & d\ (2-\log_2 d)\ \underbrace{(1-d)^N}_{e^{-d\,N}} > \frac{w}{N} \\
\Leftrightarrow\quad & \underbrace{\frac{d\,N}{e^{d\,N}}}_{<\frac{1}{e}}\ (2-\log_2 d) > w \\
\Leftrightarrow\quad & \log_2 d \ll 2 - w\,e \\
\Leftrightarrow\quad & d \ll \frac{4}{2^{w\,e}} \\
\Leftrightarrow\quad & d \ll 1.7 \cdot 10^{-52} \text{ avec } w = 64
\end{aligned}
$$

Toutefois, en choisissant une valeur de N au moins égale à $2\,w$, on observe un surcoût mémoire inférieur à n bits pour représenter le vecteur par rapport à la structure RRR. Plus la valeur de N est grande moins le surcoût sera notable, mais plus les mises à jour seront fastidieuses ; le bon compromis étant d'avoir un surcoût acceptable (N pas trop proche de w) tout en maximisant le nombre de blocs (N proche de w), puisque plus le nombre de blocs est important, plus les mises à jour sont efficaces.

RUBIKS permet de rendre les données compressées avec une RRR modifiables sans tout devoir décompresser re-compresser avec comme seul inconvénient un léger surcoût en mémoire. *RUBIKS* n'a pas été implémenté parce qu'une structure plus adaptée a été adoptée, exploitant les propriétés combinatoire des k-mers présents, et un découpage adéquat en préfixe et suffixe de ces k-mers.
Dans les chapitres suivants, nous décrirons plus en détails cette idée qui a été implémentée dans la librairie `libGkArrays-MPI` (*cf.* chapitre 5) et dans le logiciel *RedOak* (*cf.* chapitre 6).

Chapitre 5

L'indexation des k-mers d'un génome : l'outil *gkampi*

Nous présentons dans ce chapitre la librairie `libGkArrays-MPI` dont le point de départ s'inspire des *Gk-Arrays* (*cf.* sous-section 2.2.4), qui est la librairie développée pour le logiciel CRAC [Philippe et al., 2011]. Nous présenterons ensuite l'algorithme principal de *gkampi*, un outil implémenté en C++ pour indexer et interroger les k-mers issus de séquences biologiques. L'algorithme derrière cette librairie est massivement parallélisé afin de permettre une exploitation de la structure sur un *cluster* de calcul. Nous illustrerons la puissance de cet outil au travers des requêtes qu'il permet de réaliser. Pour finalement discuter des performances de *gkampi*, montrant qu'il surpasse les outils existants en calculant sa complexité en espace et en temps.

Dans *gkampi*, nous avons développé une nouvelle stratégie d'interrogation et de localisation d'un ensemble de k-mers, qui va au delà du simple comptage de k-mers. Cette stratégie est basée à la fois sur l'observation attentive de l'organisation en k-mers et sur une architecture parallèle dédiée mais hautement portable.
La librairie `libGkArrays-MPI` ainsi que l'outil *gkampi*, acronyme de *Gk-Arrays MPI*, sont disponibles à l'adresse :

`https://gitlab.info-ufr.univ-montp2.fr/doccy/libGkArrays-MPI`

5.1 Descriptif de la librairie `libGkArrays-MPI`

Le logiciel *gkampi* repose sur la librairie `libGkArrays-MPI` qui propose une structure d'indexation des k-mers. Les méthodes introduites dans la section 2.2 (*Jellyfish*, *KMC*, *Khmer*) sont dédiées uniquement au comptage des occurrences de k-mers à partir de séquences biologiques.

Le comptage des k-mers est une tâche moins complexe que l'indexation des k-mers. L'indexation doit permettre, à partir d'une séquence génomique et d'une requête, de récupérer le nombre d'occurrences de chaque k-mer qui composent la requête, mais aussi les positions de chacune de leurs occurrences dans la séquence génomique.

La position d'un k-mer peut être représentée soit par sa position globale dans la ou les séquences d'entrée concaténées, un identifiant unique attribué à chaque k-mer, soit par sa position locale, un identifiant de séquence couplé à une position dans cette séquence.

L'idée principale de l'algorithme d'indexation consiste à découper les k-mers en paires préfixe-suffixe.
Pour un k-mer il y aura un préfixe de taille k_1 et un suffixe de taille k_2 nous noterons $k = k_1 + k_2$ où k_1 et k_2 sont déterminés analytiquement.
Le résultat de l'indexation est constitué de deux tables : la table des préfixes et la table des suffixes.
La table des préfixes est de taille 4^{k_1}. Chaque case de cette table est associée au préfixe de taille k_1, correspondant à son rang lexicographique. Ainsi la case d'indice i ($0 \leq i < 4^{k_1}$) correspond au $(i+1)^{\text{ème}}$ mot de longueur k_1 qu'il est possible de construire sur l'alphabet $\{A, C, G, T\}$ en suivant l'ordre lexicographique. La table des suffixes, quand à elle, permet de stocker les suffixes de longueur k_2 des k-mers à indexer.

Chaque cellule de la table des préfixes contient l'indice du dernier suffixe associé à ce préfixe dans la table des suffixes. Ceci permet d'accéder facilement à la partie de la table des suffixes associée à un préfixe donné (c'est une somme de préfixes du nombre de suffixes associés à chaque préfixe).
Le choix de k_1 doit être tel que l'espace requis pour stocker les 4^{k_1} compteurs et l'ensemble des suffixes de longueur k_2 soit minimal. Intuitivement, cela représente une valeur telle qu'il n'y a pas de case de la table des préfixes qui ne soit pas associée à au moins un suffixe. Cela représente également une valeur telle que les suffixes sont répartis de manière homogène entre les préfixes.
L'algorithme est séparée en quatre étapes principales.

5.2 L'algorithme principal

L'algorithme de la `libGkArrays-MPI` repose sur la cohabitation d'un parallélisme lourd (calcul distribué) et d'un parallélisme léger (*multithreading*). L'architecture informatique est composée de n processus $C_0, ..., C_{n-1}$, capables de communiquer.

Le processus C_0 est considéré comme étant le nœud maître, et effectue des tâches particulières de supervision des autres processus, en plus d'effectuer ses calculs d'indexation à l'instar des autres processus.
On considère maintenant que, ayant la connaissance de $x = \left\lfloor \frac{4^{k_1}}{n} \right\rfloor$, chaque processus C_i est respectivement responsable des k-mers dont les préfixes sont indexés de $i \times x$ à $(i+1) \times x - 1$, sauf le dernier qui va jusqu'au rang $n - 1$.

Première étape : lecture des fichiers d'entrée.

Selon que le fichier d'entrée est compressé ou non, cette tâche est soit exécutée par C_0 ou bien elle est distribuée sur l'ensemble des nœuds.
Lorsque les données sont compressées, C_0 est en charge d'extraire les k-mers (car le format `gzip` ne permet pas nativement l'accès aléatoire en temps constant). Sinon, chaque processus sélectionne directement le fragment attendu du fichier fastq/fasta (qui correspond à $\frac{1}{n}^{\text{ème}}$ de la taille totale du fichier en octets), puis localise la séquence de démarrage suivante et informe le processus précédent que ce dernier aura des octets supplémentaires à lire pour extraire tous les k-mers de sa partie.

Si plusieurs fichiers sont fournis en entrée, éventuellement avec des formats différents, ils sont traités les uns à la suite des autres.
Il est important de souligner que contrairement à tous les autres outils implémentés à ce jour, *gkampi* suit rigoureusement la spécification des fichiers `fastq` (les séquences biologiques comme les séquences qualité peuvent être réparties sur plusieurs lignes, les fichiers peuvent contenir des lignes vides, ...).

Seconde étape : Distribution et comptage des suffixes des k-mers par préfixe.

Chaque processus C_i lit les k-mers de sa portion de fichier (si le fichier est compressé, seul le nœud C_0 lit le fichier et les autres nœuds n'auront rien à lire). Selon le préfixe du k-mer, le processus communique celui-ci au nœud en charge de ce k-mer ou bien le traite directement.

Afin d'optimiser la communication, chaque processus va emmagasiner temporairement les k-mers qui devront être traités par chaque nœud C_j dans un tampon noté B_j^i. Si le préfixe du k-mer en train d'être considéré correspond à la partie du tableau manipulée par C_i, il est directement traité. S'il correspond à la partie du tableau gérée par C_j avec $j \neq i$, alors il est mis dans le tampon B_j^i.

Une fois qu'un tampon est plein ou s'il n'y a pas d'autres k-mers à lire, le processus C_i envoie les k-mers stockés dans B_j^i au nœud C_j, qui les stocke et renvoie en retour ses k-mers stockés dans B_i^j au nœud C_i (l'échange est réalisée sur place [1] pour les deux processus).
Ensuite, chaque k-mer reçu est traité. Le traitement consiste à cet instant à incrémenter le compteur associé au préfixe du k-mer. Cela fonctionne jusqu'à ce que toutes les données aient été lues, puis un dernier tour d'échanges de tampons est effectué.

Le tableau des suffixes est alors alloué pour stocker les suffixes lors de la deuxième étape (figure 5.1).

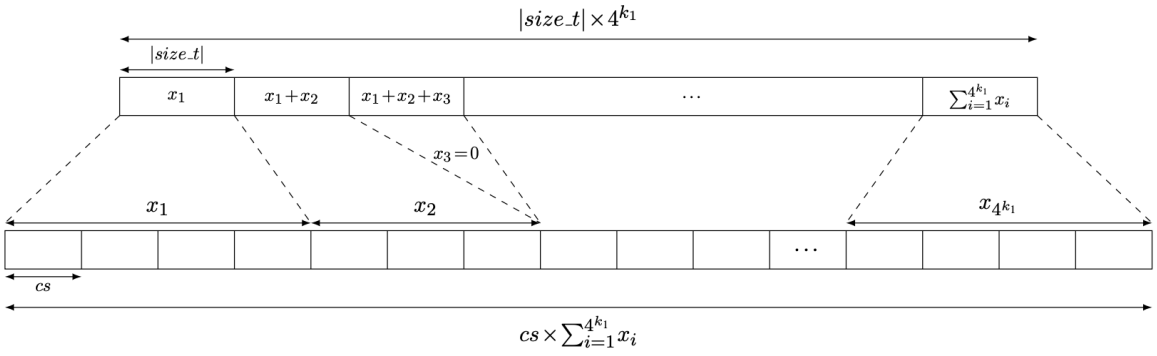


Figure 5.1: Structure générale de l'organisation de la structure de la `libGkArrays-MPI` en mémoire.

Source: [Mancheron et al., 2020]

La figure 5.1 représente la structure des tables (préfixes et suffixes) en mémoire. Chaque cellule contient le décalage de la cellule de début pour le préfixe suivant. Le tableau inférieur représente le stockage des informations sur les suffixes. Chaque cellule utilise un ou plusieurs mots mémoire pour stocker un suffixe (et sa position le cas échéant). Cette taille cs est choisie par le compilateur selon des critères d'efficacité, et n'est pas nécessairement la plus petite possible.
La taille de chaque cellule des suffixes, notée cs, dépend de l'utilisation. La taille cs variera donc en fonction de si l'utilisateur fait du comptage de k-mers ou de l'indexation de k-mers.

1. Sans mémoire additionnelle.

Troisième étape : stockage des suffixes des k-mers par préfixe.

Une fois que la table des suffixes est allouée, chaque processus connaît le nombre de suffixes qu'il doit gérer pour chacun de ses préfixes. Les fichiers doivent être relus selon les mêmes modalités que lors des deux premières phases, chaque processus connaît la partie qu'il doit relire. La principale différence avec la seconde étape réside dans le traitement des k-mers, les suffixes (et leurs positions associées) sont stockés à la première case libre de la zone dédiée au préfixe du k-mer dans la table des suffixes.

Quatrième étape : tri, comptage et indexation des k-mers.

Cette étape est réalisée en parallèle sur chaque C_i, mais il n'y a plus besoin de communication entre eux. Les suffixes des k-mers (et leurs positions le cas échéant) partageant un préfixe commun sont triés par ordre lexicographique.
Étant donné que le tri (pour un préfixe donné, $1 \leq k_1 < k$) affecte une partie contiguë de la table des suffixes indépendamment des autres, cette étape est parallélisée en utilisant un parallélisme léger (*multithreading*).
À la fin de l'étape de tri, le tableau est rempli et prêt à être exploité.
Les tables de préfixes et de suffixes sont réparties sur les n processus, ce qui réduit d'un facteur n la quantité totale de mémoire pour chaque processus.

5.3 Les requêtes

La librairie `libGkArrays-MPI` offre la possibilité d'effectuer des requêtes. Grâce à l'organisation des préfixes/suffixes décrite dans section 5.2, ces requêtes peuvent être exécutées soit localement (sur un nœud C_i donné qui vérifie sur sa portion des deux tables), soit globalement (requête sur l'ensemble des nœuds). Les requêtes proposées vont d'interrogations assez générales (Combien de nucléotides ont été lus ? Combien de k-mers ont été analysés ? . . .) à des interrogations ciblées (Combien de fois un k-mer donné apparaît dans les fichiers ? À quelles positions ? Étant donnée une séquence, à quelles positions apparaissent chacun des k-mers qui la compose ? . . .).

L'interrogation portant sur un k-mer donné mène à la rechercher dans une sous-partie triée de la table des suffixes délimitée par les nombres associés au préfixe de ce k-mer (et son successeur dans le tableau des préfixes).

5.4 Analyse de la complexité

Dans cette partie, nous présentons les complexités en temps et en espace de la librairie `libGkArrays-MPI`, en utilisant les notations ci-dessous :

$$\begin{cases} \mathcal{N} & \text{Nombre total de } k\text{-mers } (= |\mathcal{K}|) \\ n & \text{Nombre d'instances s'exécutant en parallèle} \\ \mu & \text{Taille en bits d'un mot mémoire} \end{cases}$$

Théoreme 1. *L'espace total nécessaire pour compter les occurrences de $\mathcal{N}$ k-mers est égal à :*

$$4^{k_1}\,\mu \;+ 2\,k_2\,\mathcal{N} + O\,(n) \;\; \textit{bits.}$$

L'espace total nécessaire pour indexer $\mathcal{N}$ k-mers est égal à :

$$4^{k_1}\,\mu \;+ 2\,k_2\,\mathcal{N}\,\log\mathcal{N} + O\,(n) \;\; \textit{bits.}$$

En posant $k_1 = \frac{\log \mathcal{N} - \log\log \mathcal{N} + O(1)}{2}$*, ces complexités sont alors respectivement :*

$$2\,k_2\,\mathcal{N} \; + o(\mathcal{N}) \;\; \textit{bits} \qquad \textit{et} \qquad 2\,k_2\,\mathcal{N}\log\mathcal{N} \; + o(\mathcal{N}) \;\; \textit{bits.}$$

Démonstration. Il existe une bijection triviale entre les k-mers et leur rang lexicographique. Comme l'alphabet est de taille 4, seuls deux bits ($\log_2 4$) sont nécessaires pour représenter chaque symbole. Supposons que `A` est codé par 00, `C` est codé par 01, `G` est codé par 10 et `T` est codé par 11 ; toute séquence de symboles de longueur fixe possède un schéma de codage unique, qui la convertit en un entier non signé qui représente également son rang lexicographique parmi toutes les séquences de même taille.
Les préfixes ne sont donc pas stockés puisqu'il sont implicitement représentés par la tables des préfixes, qui est composée de 4^{k_1} compteurs, chaque compteur étant un mot mémoire. Il faut donc $4^{k_1}\,\mu$ bits pour représenter cette table.
Cette table est fractionnée sur les n nœuds, or chaque nœud requiert $O(1)$ bits pour la structure qui contrôle sa partie de cette table.
Pour représenter le suffixe d'un k-mer, il faut donc 2 bits par nucléotide, soient $2\,k_2$ bits par suffixe et donc $2\,k_2\,\mathcal{N}$ bits pour l'ensemble des suffixes. Si les positions sont stockées, il faut également $\lceil \log \mathcal{N} \rceil$ bits additionnels par suffixe.
Cette table est également fractionnée sur les n nœuds et il faut également $O(1)$ bits par nœud pour la structure permettant de manipuler cette seconde table.
Il en résulte une complexité totale en espace de $4^{k_1}\,\mu \; + 2\,k_2\,\mathcal{N} + O(n)$ bits si les positions ne sont pas conservées et $4^{k_1}\,\mu + 2\,k_2\,\mathcal{N}\,\log\mathcal{N} + O(n)$ bits si les positions sont conservées.

En posant $k_1 = \frac{\log \mathcal{N} - \log\log \mathcal{N} + O(1)}{2}$, on a :

$$\begin{aligned} 4^{k_1} &= 4^{\frac{\log \mathcal{N} - \log\log \mathcal{N} + O(1)}{2}} \\ &= 2^{\log \mathcal{N} - \log\log \mathcal{N}} \times O(1) \\ &= O\left(\frac{\mathcal{N}}{\log \mathcal{N}}\right) \\ &= o(\mathcal{N}) \end{aligned}$$

On a également $n \lll \mathcal{N}$, donc $n = o(\mathcal{N})$. Il en résulte que dans ce cas, les complexités en espace sont respectivement $2\,k_2\,\mathcal{N} \; + o(\mathcal{N})$ bits pour le comptage seulement et $2\,k_2\,\mathcal{N}\log\mathcal{N} \; + o(\mathcal{N})$ bits pour l'indexation des positions. □

Pour des raisons de performance en temps, nous avons fait le choix d'utiliser des mots binaires de type `uint_fast8_t` pour le stockage des informations de suffixes et de positions. En notant μ' le nombre de bits d'un entier de type `uint_fast8_t`, l'espace utilisé pour ce stockage dans la librairie `libGkArrays-MPI` est $\mu' \left\lceil \frac{2\,k_2}{\mu'} \right\rceil$ bits par suffixe et $\mu' \left\lceil \frac{\log \mathcal{N}}{\mu'} \right\rceil$ pour le stockage des positions.

Théoreme 2. *Sous l'hypothèse que* $k = O(\log \mathcal{N})$*, le temps nécessaire par nœud pour indexer les occurrences de* $\mathcal{N}$ *k-mers est :*

$$O\left(\frac{\mathcal{N}\,k}{n}\right) \qquad .$$

Démonstration. Rappelons que les k-mers sont représentés par un tableau de mots de type `uint_fast8_t` (de μ' bits), à raison de 2 bits par nucléotide.
Soit u le précédent k-mer lu et α la lettre qui suit ce k-mer dans le fichier. Alors pour

construire le nouveau k-mer, il faut décaler tous les bits de deux rangs vers la gauche et ajouter le codage correspondant au symbole α. L'opération de décalage requiert $O\left(\frac{k}{\mu'}\right)$ opérations et l'ajout du symbole α $O(1)$ opérations.
Ainsi la lecture des $\mathcal{N}$ k-mers requiert $O\left(\frac{\mathcal{N}\,k}{\mu'}\right)$ opérations, soient $O\left(\frac{\mathcal{N}\,k}{n\,\mu'}\right)$ par nœud. Le temps d'échange des tampons B_j^i est une opération qui se fait en $O\left(n\,|B_j^i|\right)$ pour chaque nœud. En notant b la taille d'un tampon, l'ensemble des échanges représente $O\left(n^2\,\frac{\mathcal{N}}{n\,b}\,\frac{k}{\mu'}\right) = O\left(n\,\frac{\mathcal{N}}{b}\,\frac{k}{\mu'}\right)$ opérations. En supposant que les données échangées entre les nœuds soient de volumes équivalents, cela représente donc $O\left(\frac{\mathcal{N}\,k}{b\,\mu'}\right)$ par nœud. Le traitement d'un k-mer lors de la seconde étape est en temps constant (incrément d'un compteur).
L'allocation de la table des suffixes lors de la troisième étape (en $O\left(\frac{\mathcal{N}}{n}\right)$ pour un nœud) requiert de calculer les sommes préfixes de la table des préfixes et d'allouer la mémoire, donc $O, \left(\frac{4^{k_1}}{n}\right)$ opérations par nœud.
Le traitement d'un k-mer lors de la quatrième phase requiert l'écriture du suffixe (et éventuellement de sa position) dans la table des suffixes préalablement allouée. Il faut donc $O\left(\frac{k_2}{\mu'}\right)$ opération pour l'écriture du suffixe et le cas échéant $O\left(\frac{\log\mathcal{N}}{\mu'}\right)$ opérations pour l'écriture de la position. On présumera que $k_2 = O(\log\mathcal{N})$ pour simplifier les calculs.
Le tri des suffixes associés à un préfixe se fait en utilisant une implémentation basée sur une combinaison de tri par tas et de tri par sélection dont la complexité est en $O\left(m\,\log m\,\frac{k_2}{\mu'}\right)$, pour un ensemble de m suffixes. Ainsi, chaque tri requiert en moyenne $O\left(\frac{\mathcal{N}}{4^{k_1}}\,\log\frac{\mathcal{N}}{4^{k_1}}\,\frac{k_2}{\mu'}\right)$ opérations et de ce fait, l'ensemble des tris requiert $O\left(\frac{\mathcal{N}}{n}\,\frac{\log\mathcal{N}}{4^{k_1}}\,\frac{k_2}{\mu'}\right)$ opérations pour un nœud.
On obtient donc une complexité temporelle par nœud de

$$O\left(\frac{\mathcal{N}\,k}{n\,\mu'} + \frac{\mathcal{N}\,k}{b\,\mu'} + \frac{\mathcal{N}}{n} + \frac{4^{k_1}}{n} + \frac{\mathcal{N}}{n}\frac{k_2}{\mu'} + \frac{\mathcal{N}}{n}\,\log\frac{\mathcal{N}}{4^{k_1}}\,\frac{k_2}{\mu'}\right) \quad .$$

En réalité, le terme μ' est une constante ; aussi, en posant $b \geq n$ et sous l'hypothèse raisonnable que $k = O(\log\mathcal{N})$, donc que k_1 et k_2 sont également dans $O(\log\mathcal{N})$, ceci se simplifie considérablement en :

$$O\left(\frac{\mathcal{N}\,k}{n}\right) \quad .$$

□

Théoreme 3. *Le temps nécessaire pour connaître le nombre d'occurrences d'un k-mer donné est*

$$O\left(k_2\,\log\frac{\mathcal{N}}{4^{k_1}}\right) \quad .$$

Le temps nécessaire pour connaître les positions d'un k-mer donné ayant nb_occ occurrences est alors

$$O\left(k_2\,\log\frac{\mathcal{N}}{4^{k_1}} + nb\_occ\right) \quad .$$

En posant $k_1 = \frac{\log\mathcal{N} - \log\log\mathcal{N} + O(1)}{2}$, le temps nécessaire pour connaître respectivement le nombre d'occurrences et les positions d'un k-mer donné est alors :

$$O\left(k_2\,\log\log\mathcal{N}\right) \qquad O\left(k_2\,\log\log\mathcal{N} + nb\_occ\right)$$

Démonstration. Le nombre de k-mers partageant le même suffixe que la requête est en moyenne $m = \frac{\mathcal{N}}{4^{k_1}}$.
Sachant que les suffixes sont triés par ordre lexicographique, il suffit de faire une recherche dichotomique sur ces m éléments, soit $O(\log m)$ comparaisons.
Chaque comparaison requiert $O(k_2)$ opération. Connaître le nombre d'occurrences revient à chercher par dichotomie le premier et le dernier k-mer correspondant à la requête, soit au total $O\left(k_2 \log \frac{\mathcal{N}}{4^{k_1}}\right)$ comparaisons. Obtenir l'ensemble des nb_occ positions revient à énumérer ensuite celles-ci, ce qui se fait en temps linéaire sur nb_occ.

Sous réserve que le préfixe soit choisi tel que $k_1 = \frac{\log \mathcal{N} - \log\log \mathcal{N} + O(1)}{2}$, alors $4^{k_1} = O\left(\frac{\mathcal{N}}{\log \mathcal{N}}\right)$ et donc $\frac{\mathcal{N}}{4^{k_1}} = O(\log \mathcal{N})$. □

Théoreme 4. *La taille optimale de préfixe pour indexer $\mathcal{N}$ k-mers est*

$$k_1 = \frac{\log_2 \mathcal{N} - \log_2 \log_2 \mathcal{N} + O(1)}{2} \quad .$$

Démonstration. L'optimal est obtenu lorsque la valeur $4_1^k \mu + 2 k_2 \mathcal{N}$ est minimale. Soit $f_k(x) = 4^x \mu + 2(k-x)\mathcal{N}$ définie sur $x \in]0; k]$. La différence $2k\mathcal{N} - f_k(x)$ représente la différence d'espace mémoire entre l'utilisation du découpage préfixe/suffixe par rapport au stockage intégral des k-mers.

$$\begin{aligned} 2k\mathcal{N} - f_k(x) &= 2k\mathcal{N} - 4^x \mu - 2(k-x)\mathcal{N} \\ &= 2x\mathcal{N} - 4^x \mu \end{aligned}$$

L'objectif étant d'avoir un gain, on souhaite donc

$$\begin{aligned} 2x\mathcal{N} - 4^x \mu &> 0 \\ \frac{2\mathcal{N}}{\mu} &> \frac{4^x}{x} \\ \log_2 2\mathcal{N} - \log_2 \mu &> \log_2 4^x - \log_2 x \\ 1 + \log_2 \mathcal{N} - \underbrace{\log_2 \mu}_{>\log \mathcal{N}} &> 2x - \log_2 x \\ \log_2 \mathcal{N} - \log_2 \frac{\log_2 \mathcal{N}}{2} &> 2x - \log_2 x \\ x &< \frac{\log \mathcal{N}}{2} \end{aligned}$$

La dérivée de $2x\mathcal{N} - 4^x \mu$ sur $]0; \frac{\log \mathcal{N}}{2}]$ vaut $2\mathcal{N} - \ln 4 \, \mu \, 4^x$. Cette dérivée est strictement décroissante par rapport à x et s'annule en x_0 tel que :

$$\begin{aligned} 2\mathcal{N} - \ln 4 \, \mu \, 4^{x_0} &= 0 \\ 4^{x_0} &= \frac{2\mathcal{N}}{\ln 4 \, \mu} \\ 2^{2x_0} &= \frac{2\mathcal{N}}{2 \ln 2 \, \mu} \\ 2x_0 &= \log_2 \frac{\mathcal{N}}{\ln 2 \, \mu} \\ x_0 &= \frac{\log_2 \mathcal{N} - \log_2 \ln 2 - \log_2 \mu}{2} \\ x_0 &= \frac{\log_2 \mathcal{N} - \log_2 \log_2 \mathcal{N} + O(1)}{2} \end{aligned}$$

Le tableau de variations de $2\,x\,\mathcal{N} - 4^x\,\mu$ sur $]0; \frac{\log \mathcal{N}}{2}]$, nous donne :

x	1	$\longrightarrow$	x_0	$\longrightarrow$	$\frac{\log \mathcal{N}}{2}$
$(2\,x\,\mathcal{N} - 4^x\,\mu)'$	$+$		0	$-$	
$2\,x\,\mathcal{N} - 4^x\,\mu$					

Le gain maximum est donc réalisé pour une valeur de préfixe telle que $k_1 = \frac{\log_2 \mathcal{N} - \log_2 \log_2 \mathcal{N} + O(1)}{2}$.
□

Plusieurs travaux ont été menés entre 1990 et 2010 par Nicodème et Szpankowski pour étudier la topologie des *tries* et sont synthétisés dans [Park et al., 2009]. Les études sont ramenées à l'analyse d'arbres binaires, qui se généralisent facilement aux arbres n-aires en considérant chaque zone de $\log n$ niveau d'un arbre binaire comme un niveau de l'arbre n-aire correspondant. Ainsi, ils mettent en évidence trois zones structurelles. La première zone va de la racine à un niveau appelé le *fill-up* que nous noterons ℓ_1 et est tel que tous les nœuds de cette zone ont exactement deux fils. La seconde zone est une zone intermédiaire qui va jusqu'au niveau ℓ_2 et la troisième zone est telle que les nœuds ont essentiellement un seul fils ou bien sont des feuilles. Ils distinguent deux cas de figure : les arbres symétriques (ce qui se traduit par une fréquence de 0 et de 1 a peu près équivalente) des arbres asymétriques. Nous considérons que dans le cadre des séquences biologiques, les arbres sont symétriques. Dans ce contexte et selon [Park et al., 2009], étant donné un ensemble de m mots binaires :

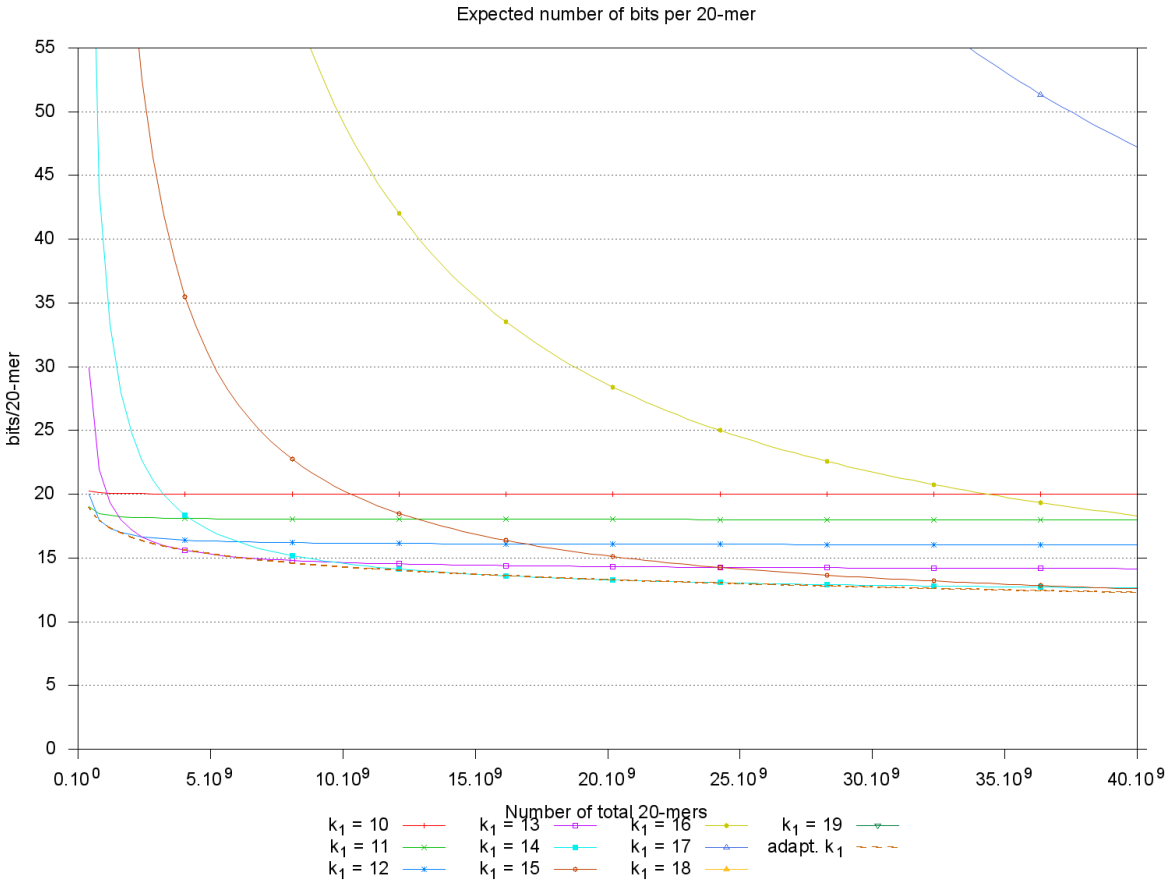
$$\ell_1 = \log_2 m - \log_2 \log_2 m + O\left(\log_2^{-1} m\right)$$
$$\ell_2 = \log_2 m + K_m \quad \text{avec } 1 \ll K_m = o\left(\sqrt{\log_2 m}\right)$$

Dans le cadre d'un alphabet de taille 4, il convient donc de considérer que deux niveaux du *trie* binaire correspondent à 1 niveau de l'arbre quaternaire et donc le *fill-up* est alors $\frac{\ell_1}{2}$. Ceci corrobore donc le fait que l'optimal correspond au *fill-up* du *trie* des k-mers, ce qui est intuitivement cohérent.

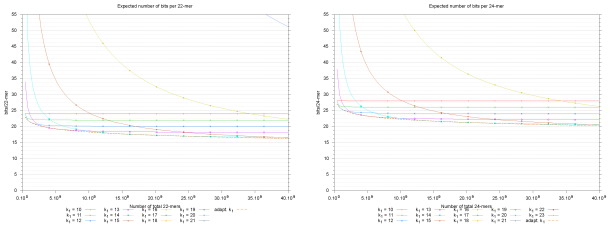
Afin d'étayer visuellement ce résultat, nous avons observé les courbes associées aux fonctions $f_k(\mathcal{N}, k_1) = (64 + 32)\,4^{k_1} + 2\,k_2\,\mathcal{N}$ pour différentes valeurs de k (*cf.* figure 5.2). Pour être plus précis, nous avons calculé le nombre de bits par k-mer que prendrait la structure globale. Pour une valeur de k donnée, nous avons tracé les courbes correspondant aux valeurs fixes de $k_1 \in [10; k[$, ainsi que la valeur adaptative en fonction de $\mathcal{N}$ qui est $\frac{\log_2 \mathcal{N} - \log_2 \log_2 \mathcal{N} + c}{2}$, avec $c = \log_2 \log_2 e - \log_2 e \simeq -0,913$. On peut aisément constater que toutes les courbes sont minorées par la courbe correspondant au choix adaptatif du préfixe. On observe également que plus le nombre de k-mers est important, plus l'économie réalisée l'est également et qu'on est proche de 1 bit par nucléotide indexé, grâce à la factorisation par préfixe.

5.5 Résultats

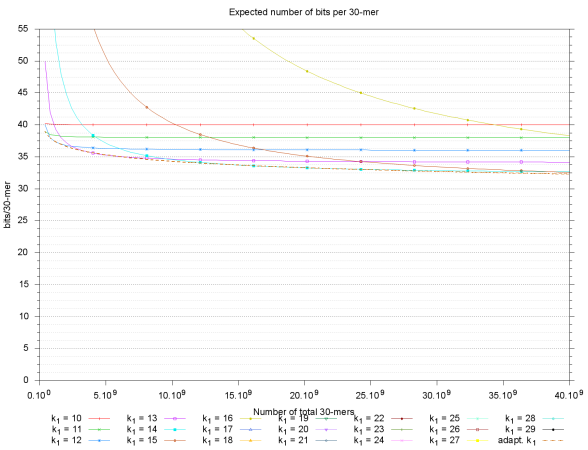
Les expériences présentées dans ce chapitre ont été réalisées sur le *cluster* de l'équipe MAB et administré avec `SGE`. Ce *cluster* est hétérogène et comporte 3 types de machines. Tous les nœuds du *cluster* ont cependant 2 processeurs à 6 cœurs chacun (*via* la virtualisation des



(a) Nombre de bits par 20-mers



(b) Nombre de bits par $\{22, 24, 26, 28\}$-mers



(c) Nombre de bits par 30-mers

Figure 5.2: Nombre théorique de bits par k-mers en fonction de la taille du préfixe. La figure (a) correspond à $k = 20$, la figure (c) à $k = 30$ et les figures (b) aux valeurs de k allant de 22 à 28 par pas de 2. Lorsque le préfixe est choisi en fonction du nombre de k-mers attendus, on observe que l'utilisation de la mémoire est optimale.

cœurs, les machines apparaissent comme ayant 24 cœurs disponibles [2]). Parmi les machines, 7 disposent de 72GiB de RAM, 4 disposent de 128GiB et 2 disposent de 192GiB de RAM (ces dernières disposent également d'un espace de stockage propre de 250GiB).
Enfin, les nœuds du *cluster* sont administrés par une machine mettant à disposition un stockage réseau de 150TiB dont une partie utilise le système de fichiers *BeeGFS* (système dédié aux calculateurs hautes performances pour offrir des accès concurrents sans ralentissements pour les utilisateurs).

Pour réaliser les tests, nous avons utilisés les jeux d'essais décrits dans [Erbert et al., 2017], eux mêmes largement basés sur les jeux d'essais décrits dans [Deorowicz et al., 2014] et [Marcais and Kingsford, 2011].

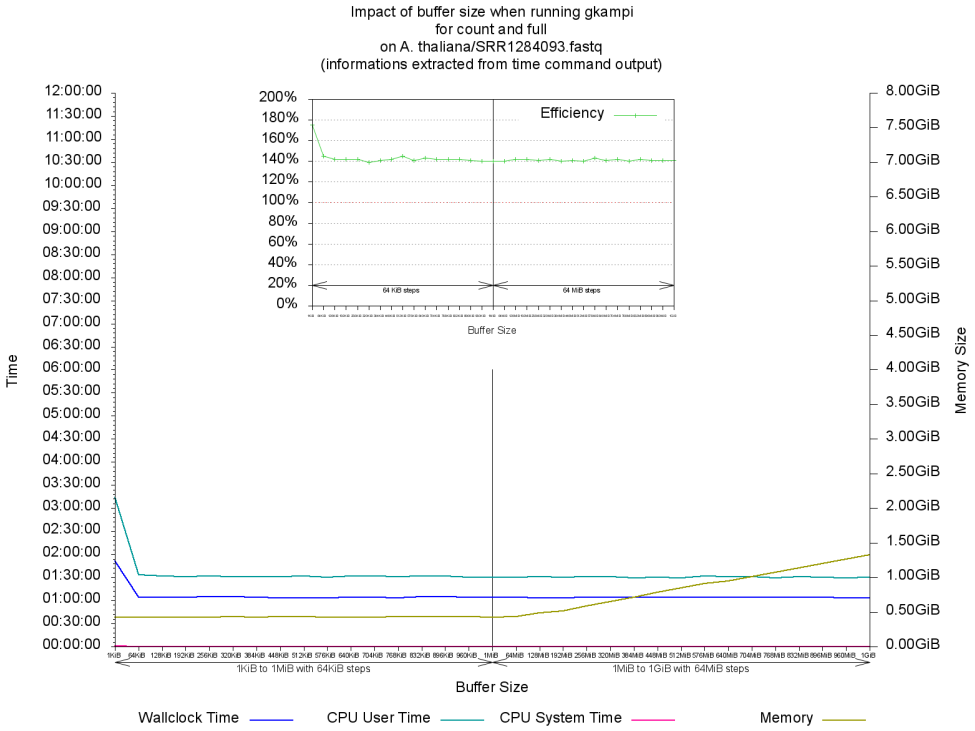
Sur la figure 5.3, on peut observer que la mémoire utilisée est peu impactée par la taille du tampon sauf lorsque celui-ci dépasse plusieurs Mo (ce qui est logique). On constate également que le choix de ce paramètre a peu d'incidence sur les performances dès lors que celui-ci atteint la dizaine de Ko par nœud.
L'efficacité mesurée correspond au temps CPU divisé par le temps réel. Les fluctuations observables dépendent de la charge du *cluster* au moment de l'expérimentation.
Par défaut dans la librairie `libGkArrays-MPI`, chaque tampon est calculé de sorte à pouvoir stocker 1024 k-mers par nœud et pour chaque nœud. Ce qui représente quelques dizaines de Ko par nœud.

Sur la figure 5.4, on peut observer que la courbe de mémoire évolue selon une courbe ayant la même tendance que la courbe théorique. La variation de mémoire entre la figure 5.4a et la figure 5.4b correspond aux stockage des positions. On observe également de grandes disparités dans les temps de calcul. Ceux-ci s'expliquent essentiellement en raison de l'hétérogénéité du *cluster*.

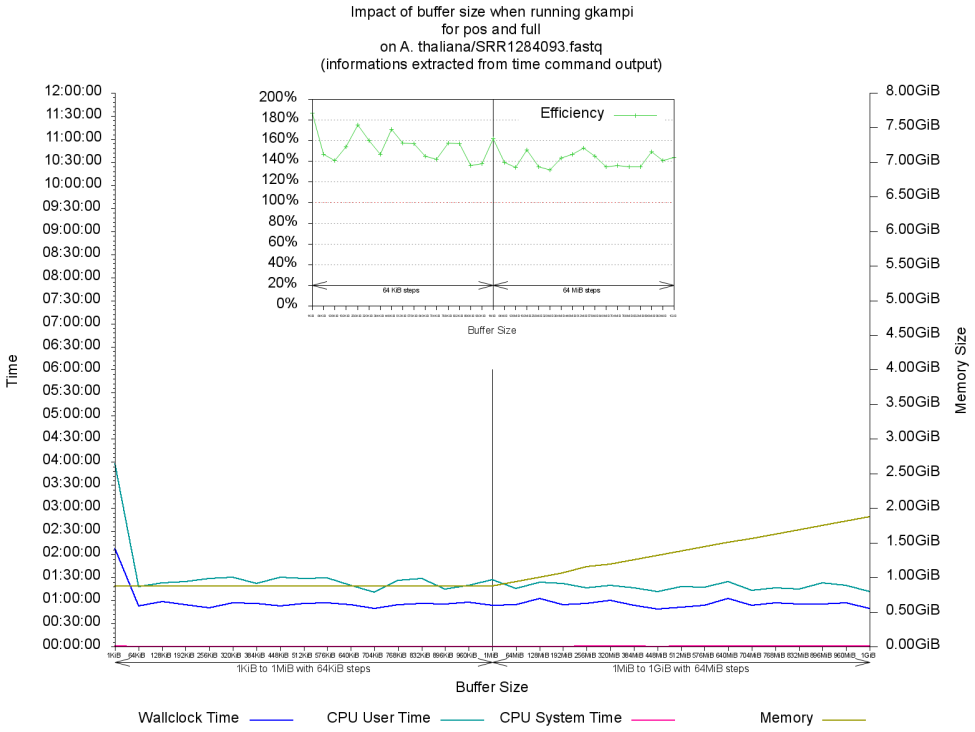
Dans la figure 5.4 les carrés représentent le temps utilisateur et les points le temps CPU par nœud. Les croix représentent la mémoire utilisée par nœud. La courbe de tendance est calculée par régression. Les graphiques inclus en haut à gauche de chaque graphiques représentent la même information de mémoire exprimée en bits par k-mer, ainsi que le nombre de k-mers uniques, de k-mers distincts, le nombre de k-mers totaux et le nombre maximum d'occurrences d'un k-mer. Les communications entre les processus sont gérés par Open MPI différemment selon que les processus sont sur le même nœud physique du *cluster* ou bien sur des nœuds différents. En effet, les échanges entre deux processus vont à terme passer par une zone de mémoire partagée ou (non exclusif) un fichier d'échange. La différence réside dans l'application de la stratégie de communication. Si les processus sont exécutés sur le même nœud physique, c'est la configuration d'Open MPI qui orchestrera seule l'échange. *A contrario*, si les processus sont sur deux machines distinctes, alors les communications vont passer par le réseau et c'est le système d'exploitation des nœuds qui assurera en priorité l'orchestration.

La figure 5.5 montre qu'en réalité l'impact en mémoire est faible que les processus soient exécutés sur un seul nœud physique ou bien sur des nœuds différents. En revanche, la figure 5.5c montre qu'en fixant $k_1 = 12$, *gkampi* a pu traiter $33 \cdot 10^9$ 20-mers (*reads* pairés du génome humain, jeux d'essais `SRR359301_1` et `SRR359301_2`) avec moins de 16 bits par k-mer, ce qui a nécessité environ 60GiB de RAM, répartis sur 8 nœuds, soit moins de 8GiB par nœud.

2. Les processeurs sont très majoritairement des processeurs Intel® $X-5650$ cadencés à $2,66$GHz et disposant de 12MB de mémoire cache.

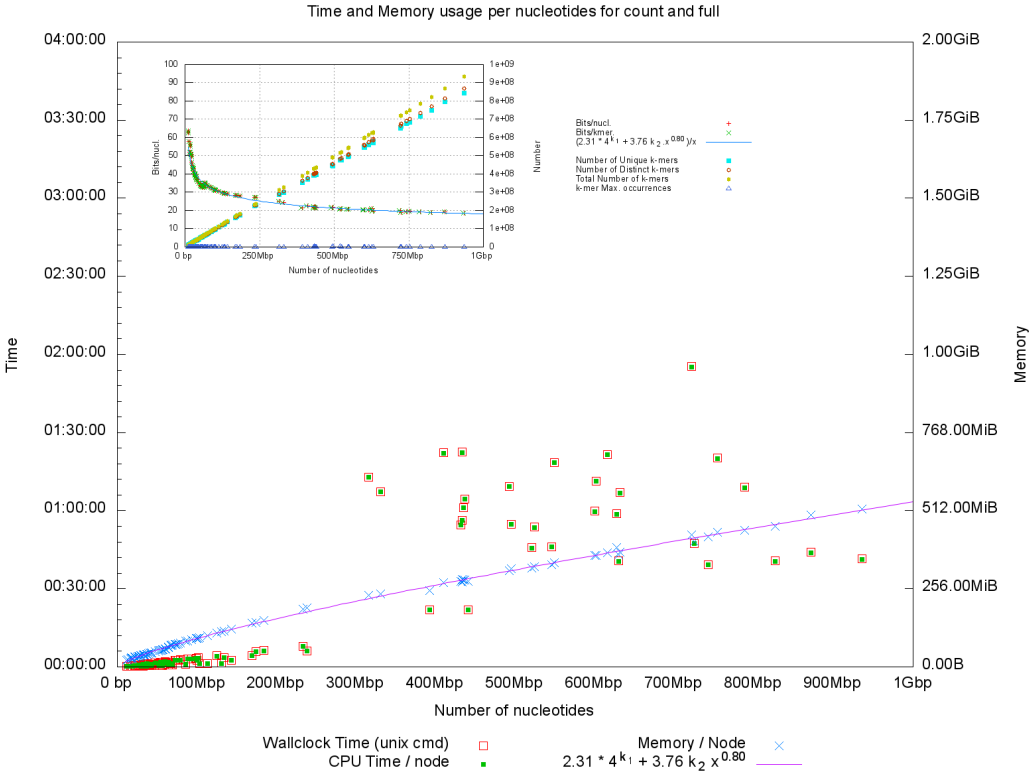


(a) Comptage seulement

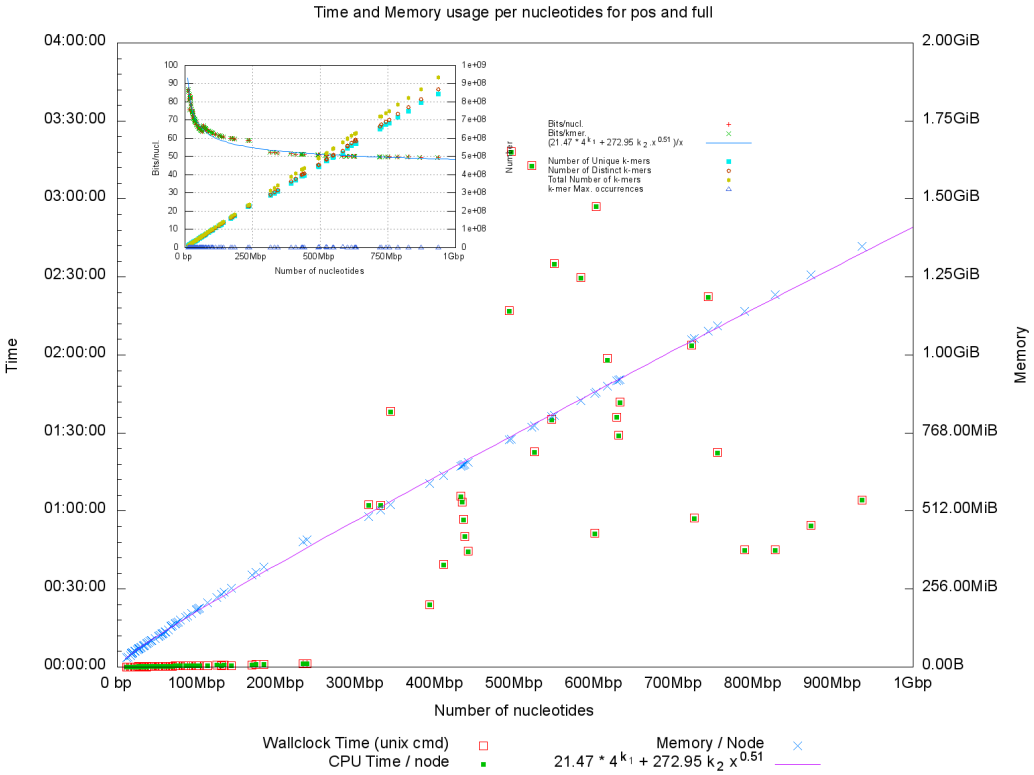


(b) Indexation des positions

Figure 5.3: Impact de la taille du tampon d'échange sur les performance. Les mesures ont été effectuées en lançant 8 instances de *gkampi* sur 8 nœuds différents du *cluster*. Le jeu de données utilisé (`SRR1284093`) provient d'un séquençage de la plante *Arabidopsis thaliana*. Ce jeu d'essai contient 160 059 séquences au format `fastq`, pour un total de 436 770 051pb. Les séquences ont une taille moyenne de 2 829pb, allant de 22pb à 86 350pb. Le fichier non compressé représente 842Mo. La figure (a) correspond à l'indexation des k-mers sans les positions tandis que la figure (b) correspond à l'indexation des k-mers avec les positions.

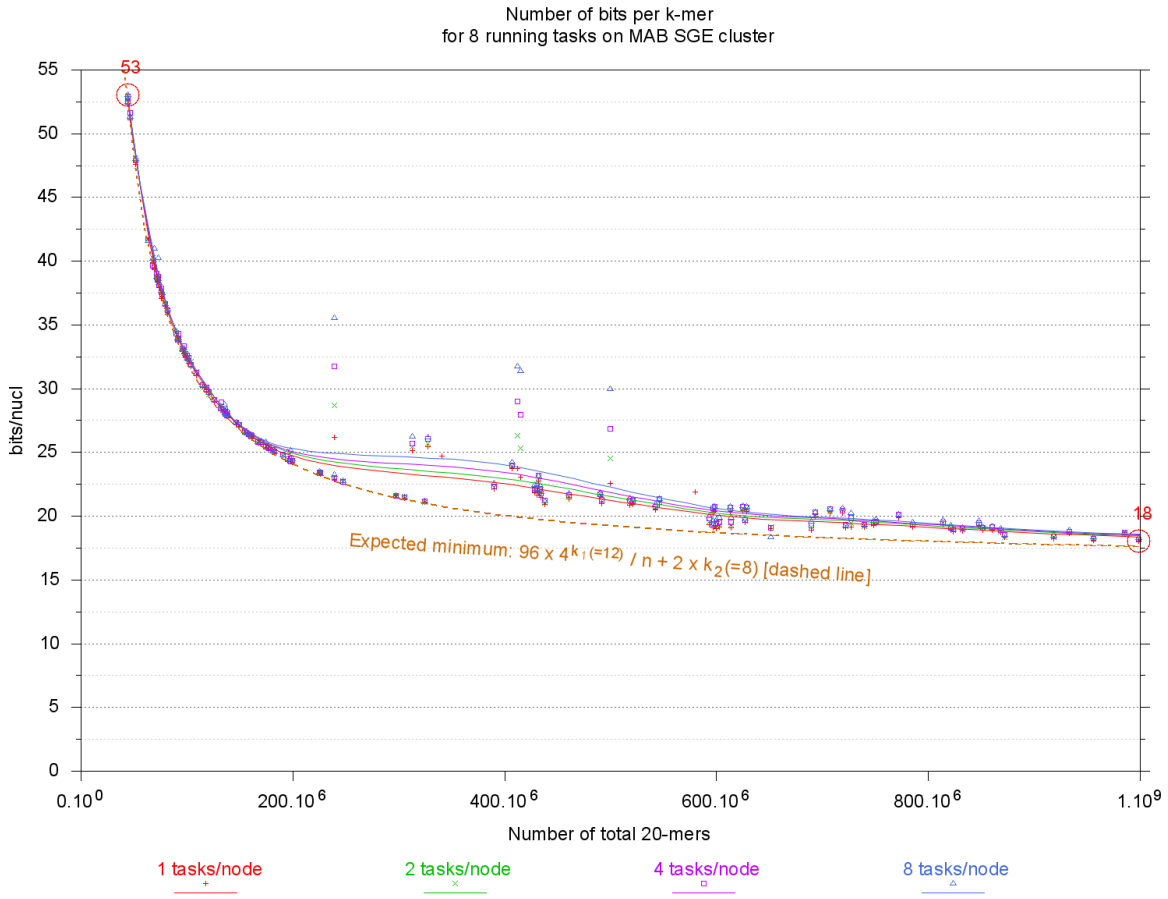


(a) Comptage seulement

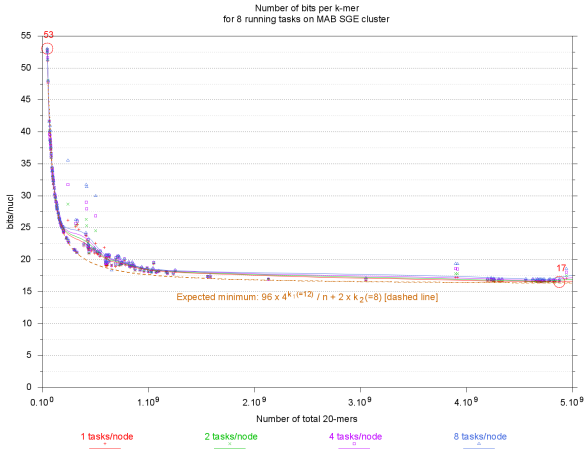


(b) Indexation des positions

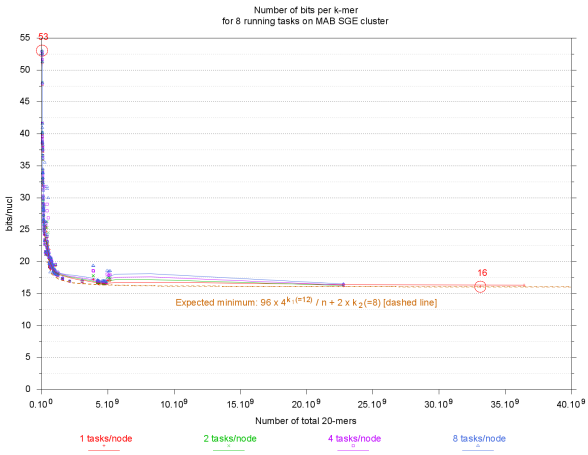
Figure 5.4: Temps de calcul et mémoire utilisée en fonction de la taille du fichier en nombre de nucléotides. Les mesures ont été effectuées en lançant 8 instances de *gkampi* sur 8 nœuds différents du *cluster*. Le graphique (a) correspond à l'indexation des k-mers sans les positions tandis que la figure (b) correspond à l'indexation des k-mers avec les positions.



(a) Détail pour un nombre de nucléotide allant jusqu'à 1Gbp.



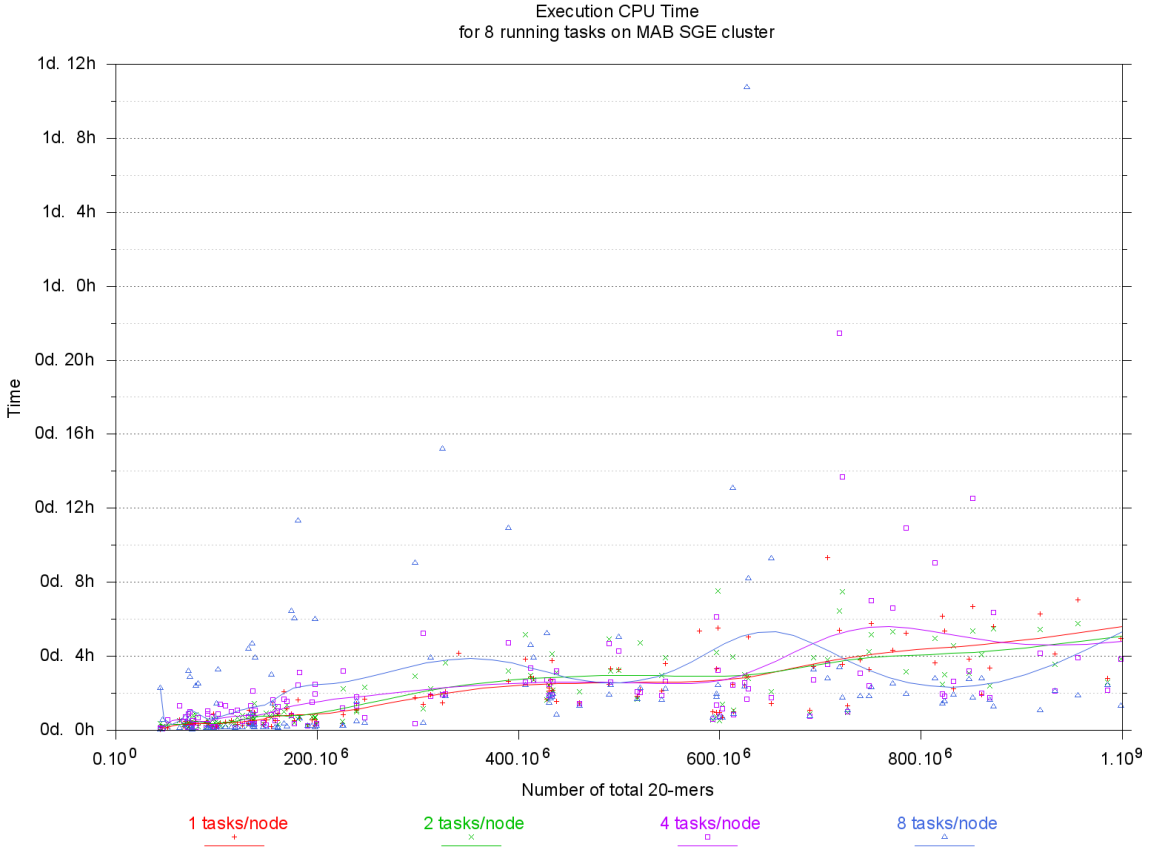
(b) Détail pour un nombre de nucléotide allant jusqu'à 5Gbp.



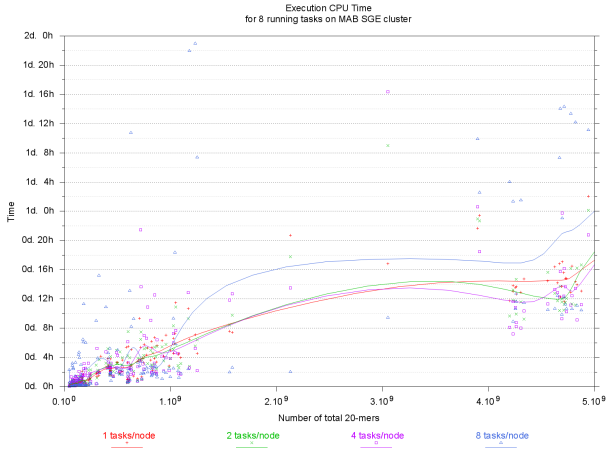
(c) Graphique complet.

Figure 5.5: Impact du nombre de machines physiques utilisées sur la mémoire par processus. Les mesures ont été effectuées en lançant 8 instances de *gkampi* sur respectivement 1, 2, 4 et 8 nœuds différents du *cluster* avec un préfixe fixé à 12. Les figures (a) et (b) correspondent respectivement au détail de la figure (c) pour les données allant jusqu'à 1Gpb et 5Gpb. La mémoire est exprimée en bits par k-mer. La courbe théorique attendue apparaît en pointillés.

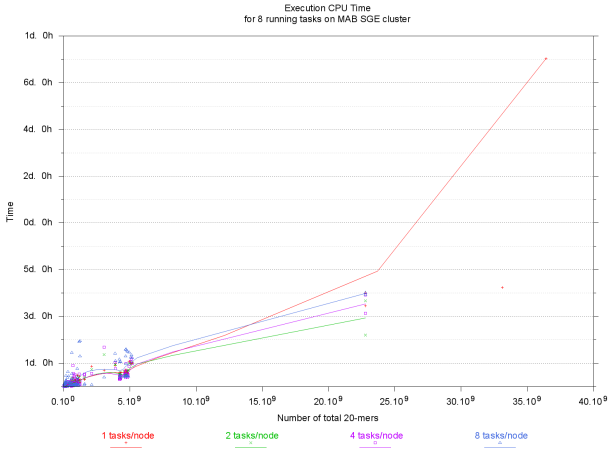
Sur la figure 5.6, on observe que le temps CPU total est légèrement impacté par le nombre de nœuds physiques utilisés (notamment sur la figure 5.6c). Cela est très probablement dû aux communications réseau. Toutefois le plus notable est la variabilité des temps de calcul indépendamment du nombre de k-mers à traiter. Cette variabilité est en partie explicable par l'hétérogénéité du *cluster*, mais également par la fluctuation de charge de celui-ci.



(a) Détail pour un nombre de nucléotide allant jusqu'à 1Gbp.



(b) Détail pour un nombre de nucléotide allant jusqu'à 5Gbp.



(c) Graphique complet.

Figure 5.6: Impact du nombre de machines physiques utilisées sur le temps d'indexation des k-mers. Les mesures ont été effectuées en lançant 8 instances de *gkampi* sur respectivement 1, 2, 4 et 8 nœuds différents du *cluster* avec un préfixe fixé à 12. Les figures (a) et (b) correspondent respectivement au détail de la figure (c) pour les données allant jusqu'à 1Gpb et 5Gpb. Le temps correspond au temps CPU total.

La figure 5.7 montre clairement que la performance en temps CPU par rapport au temps utilisateur est impacté lorsque les processus s'exécutent sur des nœuds différents du *cluster*. La performance attendue devrait en théorie être bornée par 800% pour 8 processus s'exécutant en parallèle. Cependant, certaines parties du traitement sont également exécutées en parallèle à l'intérieur d'un processus (*multithreading* assuré par OpenMP). Ceci permet d'exploiter l'*hyperthreading* des cœurs des nœuds et donc dans certains cas de dépasser cette limite. Ceci explique pourquoi il est possible d'observer une meilleure performance de 1254%. On observe clairement une performance moyenne de l'ordre de 400% à 500%, ce qui permet objectivement d'affirmer que la parallélisation de l'algorithme est cohérente et a été correctement implémentée dans la librairie `libGkArrays-MPI`.

5.6 Synthèse

La librairie `libGkArrays-MPI` est distribuée sous couvert de la licence logicielle libre et ouverte `CeCILL-C`[3] permettant ainsi son intégration, sa réutilisation, sa distribution et sa modification au sein de tout logiciel. Le logiciel *gkampi* est quant à lui distribué sous la licence logicielle libre et ouverte `CeCILL`[3], reconnue par l'OSI[4]

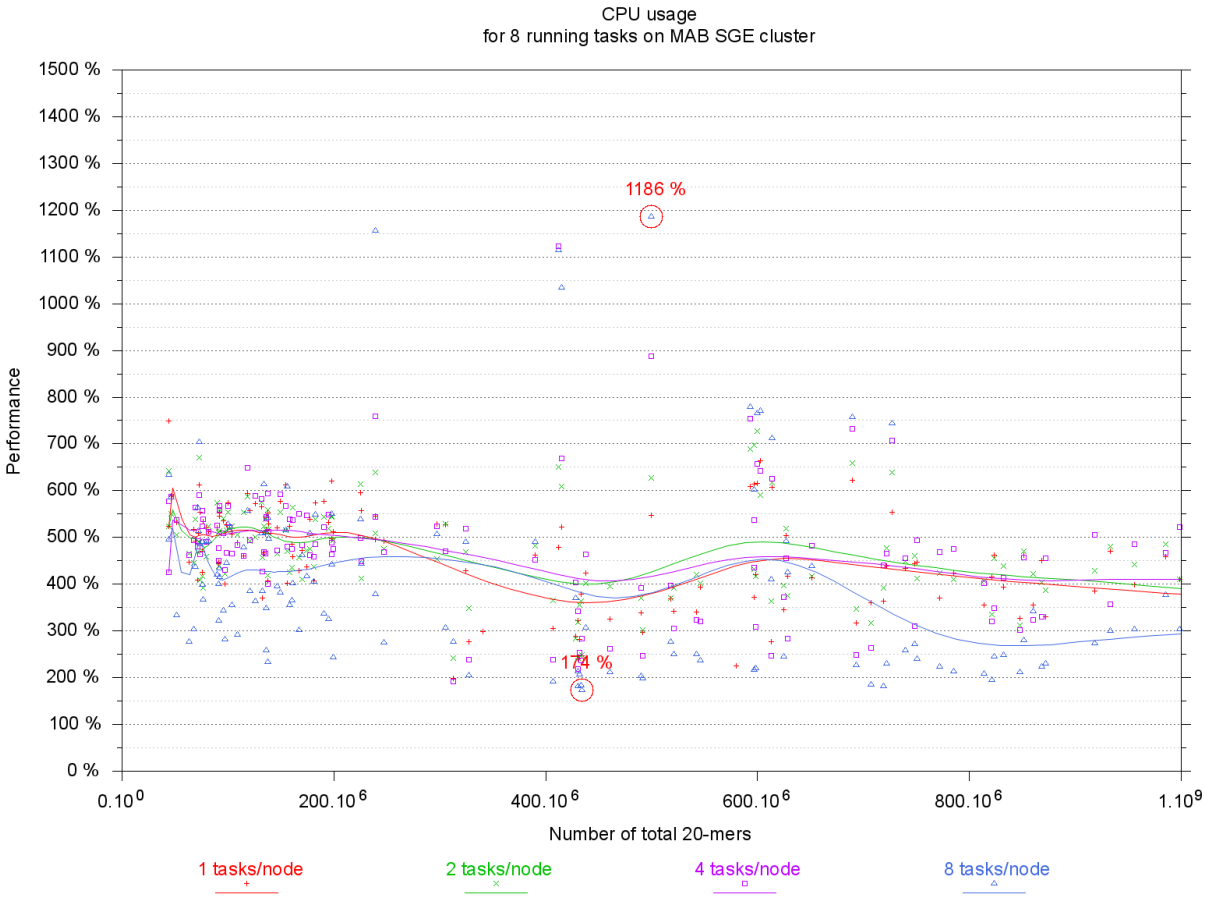
L'architecture distribuée de cette librairie et de ce logiciel est basée sur Open MPI, ce qui permet leur utilisation sur de gros volumes de données qu'il ne serait pas possible d'analyser sur une machine ordinaire. La parallélisation légère est assurée par OpenMP, ce qui permet également une exploitation des capacités d'*hyperthreading*.

L'ensemble est documenté avec Doxygen et l'empaquetage est basé sur les `autotools` (`autoconf`, `automake`, `libtool`, ...). Ainsi, leur utilisation et leur diffusion est facilitée.

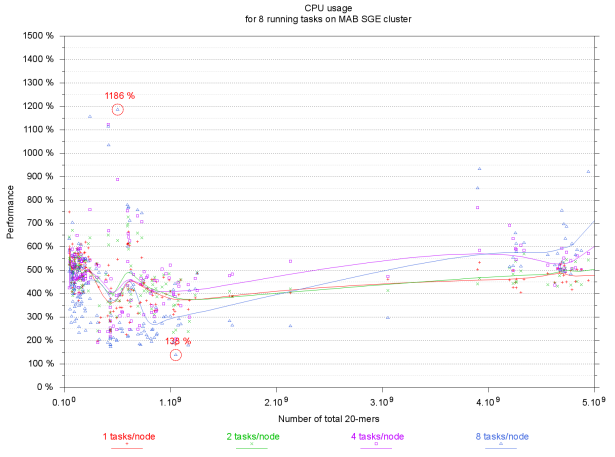
Enfin, parmi les spécificités de la librairie et de *gkampi*, outre le fait qu'à notre connaissance, ce soit la seule solution logicielle qui respecte intégralement les formats de fichiers `fasta` et `fastq` (compressés avec `gzip` ou non), figurent la possibilité d'indexer les positions des k-mers dans les fichiers ainsi que la possibilité d'indexer des k-mers espacés [Noé and Kucherov, 2005]. Cette dernière fonctionnalité semble présenter un réel intérêt dans le cadre d'analyses métagénomiques [Břinda et al., 2015].
Une autre focntionnalité non détaillée ici de la librairie `libGkArrays-MPI` est de pouvoir fournir la liste des suffixes associés à un préfixe donné en temps linéaire sur la longueur de la liste, soit $O\left(\frac{\mathcal{N}}{4^{k_1}}\right)$, donc en $O\left(\log \mathcal{N}\right)$ si $k_1 \simeq \frac{\log \mathcal{N} - \log\log \mathcal{N} - 1}{2}$.

Enfin, le principe sous-jacent utilisé pour l'indexation est le découpage en préfixe/suffixe des k-mers en se basant sur une valeur de préfixe calculée analytiquement, ce qui permet également d'optimiser l'espace mémoire. Ce principe sera donc repris dans le chapitre suivant détaillant *RedOak*.

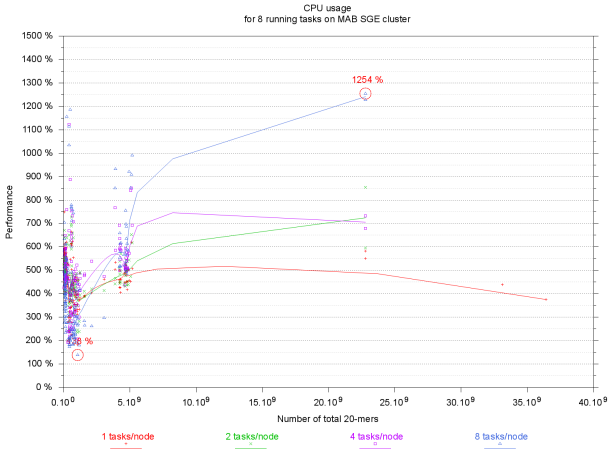
3. `https://cecill.info/`
4. *Open Source Initiative*, `https://opensource.org/`.



(a) Détail pour un nombre de nucléotide allant jusqu'à 1Gbp.



(b) Détail pour un nombre de nucléotide allant jusqu'à 5Gbp.



(c) Graphique complet.

Figure 5.7: Impact du nombre de machines physiques utilisées sur la performance en temps CPU par rapport au temps utilisateur. Les mesures ont été effectuées en lançant 8 instances de *gkampi* sur respectivement 1, 2, 4 et 8 nœuds différents du *cluster* avec un préfixe fixé à 12. Les figures (a) et (b) correspondent respectivement au détail de la figure (c) pour les données allant jusqu'à 1Gpb et 5Gpb.

Chapitre 6

L'indexation des k-mers d'une collection de génomes : l'outil *RedOak*

Dans ce chapitre nous présentons la structure d'indexation que nous avons pensée, implémentée et testée afin de répondre à la problématique de l'indexation d'une collection de génomes similaires. Nous présentons ensuite l'algorithme principal de *RedOak*, un outil implémenté en C++ pour indexer et interroger les k-mers issus d'une collection de génomes.
RedOak se base sur une structure de données similaire à celle de la librairie `libGkArrays-MPI` (*cf.* chapitre 5), pour permettre un parallélisme intensif et efficace. Nous présentons la robustesse de cet outil au travers des requêtes qu'il permet de réaliser, pour finalement discuter de ses performances, montrant sa complexité en espace et en temps.

Il s'agit d'une nouvelle méthode pour indexer une large collection de génomes basée sur la présence/absence de k-mers. Cette stratégie est basée à la fois sur l'observation attentive de l'organisation en k-mers et conçue pour une architecture parallèle (typiquement un *cluster* de calcul).
Cette méthode est mise en œuvre dans un outil que nous allons vous présenter nommé *RedOak* pour *Reference-free[d] Optimized approach by k-mers*, cet outil est disponible sur :

`https://gitlab.southgreen.fr/GenomeHarvest/RedOak.git`

6.1 Description de l'approche

RedOak est un logiciel exact implémenté en C++ qui utilise du parallélisme léger et lourd respectivement avec OpenMP et Open MPI pour indexer une grande collection de génomes similaires. *RedOak* fonctionne aussi bien avec des génomes assemblés (`fasta`) qu'à partir des fichiers bruts issus directement du séquenceur (`fastq`), les fichiers pouvant être compressés

avec `gzip`.
Ce logiciel étant basé sur une approche par k-mer, il peut donc aussi bien utiliser la librairie du logiciel *Jellyfish*, cependant cela n'est pas compatible avec une approche distribuée.

Nous avons fait le choix de la librairie `libGkArrays-MPI` (celle du logiciel *gkampi*) parce qu'elle intègre nativement les fonctionnalités de distribution des tâches avec Open MPI, qui permettent de fortement paralléliser et distribuer les calculs sur plusieurs nœuds d'un *cluster* de calcul.
RedOak, en plus des fonctions d'interrogation de séquences nucléotidiques, propose un *Shell* interactif. L'idée de cet algorithme est d'utiliser en plus du parallélisme léger (*multithreading*), le parallélisme lourd qui consiste à multiplier des processus indépendants capables de communiquer entre eux et ainsi de partager la lourde charge de travail que représente une large collection de génomes similaires.

Sans reprendre le détail de la librairie `libGkArrays-MPI` (*cf.* section 5.2), il convient de rappeler que les fichiers `fasta` ou `fastq` vont être lus en parallèle par les différents processus et chacun va procéder à la répartition des k-mers rencontrés suivant leur préfixe de taille k_1. Chaque processus s'occupe d'une partie lexicographique de l'ensemble de ces préfixes, ainsi les processus se transmettront les k-mers dont ils n'ont pas la charge.

La taille du préfixe k_1 est un paramètre d'entrée qui peut également être défini à partir du nombre estimé de k-mers à indexer (*cf.* théorème 4). Pour rappel, si $\mathcal{N}$ représente le nombre de k-mers à indexer, la valeur optimale (en espace) pour la taille des préfixes est $k_1 \simeq \frac{\log(\mathcal{N})-\log(\log(\mathcal{N})-1)}{2}$.

Une fois k_1 estimé, chaque processus s'occupera de traiter $\frac{4^{k_1}}{n}$ préfixes (où n est le nombre de processus s'exécutant en parallèle) et les k-mers correspondants. Tous les préfixes spécifiques d'un processus seront représentés en ordre lexicographique.

Afin de rester le plus générique possible, nous admettrons dans la suite que toute méthode capable de compter les k-mers et de fournir une liste ordonnée de k-mers correspondant à un préfixe donné sera réputé avoir une complexité en temps et en espace au plus égale à celles de la librairie `libGkArrays-MPI` (*cf.* section 5.4).

Avant de décrire la façon dont les k-mers sont indexés, nous introduisons quelques notations supplémentaires que nous utiliserons dans cette partie.
Il est utile de préciser que la notation $A \uplus B$ est utilisée dans ce manuscrit pour dénoter le résultat de l'union des ensembles A et B ($A \cup B$) tout sachant qu'ils sont disjoints ($A \cap B = \emptyset$). De même, la notation $\biguplus_{i=a}^{b} X_i$ dénote l'union de tous les ensembles X_i avec $a \leq i \leq b$ ($\bigcup_{i=a}^{b} X_i$) tout en sachant que tous les ensembles sont disjoints deux à deux ($\forall a \leq i < j \leq b, X_i \cap X_j = \emptyset$).
Étant donné un ensemble de N génomes $\mathcal{G} = \{G_1, \cdots, G_N\}$, les *core* k-mers correspondent au sous-ensemble, noté $\mathcal{K}^*(\mathcal{G})$, de ses k-mers partagés par tous les génomes ; les *shell* k-mers correspondent au sous-ensemble, noté $\mathcal{K}^+(\mathcal{G})$, des k-mers partagés par au moins deux génomes mais pas par tous, les *cloud* k-mers correspondent au sous-ensemble, noté $\mathcal{K}^-(\mathcal{G})$, des k-mers présents dans un seul génome, les *cloud* k-mers du génome i sont notés $\mathcal{K}^i(\mathcal{G})$.
L'ensemble des *cloud* k-mers est donc tel que $\mathcal{K}^-(\mathcal{G}) = \biguplus_{i=1}^{N} \mathcal{K}^i(\mathcal{G})$.
L'ensemble de tous les k-mers présents est noté $\mathcal{K}(\mathcal{G})$ et est tel que $\mathcal{K}(\mathcal{G}) = \mathcal{K}^*(\mathcal{G}) \uplus \mathcal{K}^+(\mathcal{G}) \uplus \mathcal{K}^-(\mathcal{G})$.
Étant donné un préfixe $pref$ (de longueur $k_1 \leq k$), le sous-ensemble des *core* k-mers dont le préfixe est $pref$ est désigné par $\mathcal{K}^*_{pref}(\mathcal{G})$, le sous-ensemble des *shell* k-mers dont le préfixe est $pref$ est indiqué par $\mathcal{K}^+_{pref}(\mathcal{G})$, le sous-ensemble de tous les *cloud* k-mers dont le préfixe est $pref$ est noté $\mathcal{K}^-_{pref}(\mathcal{G})$, le sous-ensemble de tous les *cloud* k-mers présents dans le génome

i dont le préfixe est $pref$ est noté $\mathcal{K}^i_{pref}(\mathcal{G})$, et le sous-ensemble de tous les k-mers dont le préfixe est $pref$ est noté $\mathcal{K}_{pref}(\mathcal{G})$.
Étant donné un k-mer w, on note $B^{\mathcal{G}}_w$ le tableau de N booléens tel que $B^{\mathcal{G}}_w[i]$ $(0 \leq i < N)$ est `Vrai` si et seulement si w apparaît dans le $(i+1)^{\text{ème}}$ génome indexé (le génome G_i).
Dans le reste de cette partie, les notations seront abrégées en $\mathcal{K}^*$, $\mathcal{K}^+$, $\mathcal{K}^-$, $\mathcal{K}^i$, $\mathcal{K}$, $\mathcal{K}^*_{pref}$, $\mathcal{K}^+_{pref}$, $\mathcal{K}^-_{pref}$, $\mathcal{K}^i_{pref}$, $\mathcal{K}_{pref}$, et B_w lorsqu'il l'ensemble $\mathcal{G}$ ne prête pas à confusion.

Pour les stocker et les interroger efficacement, chaque k-mer est divisé en deux parties : son préfixe de taille k_1 et son suffixe de taille k_2, avec $k_1 + k_2 = k$ (à l'instar de la librairie `libGkArrays-MPI`). En fait, les k-mers sont regroupés par leur préfixe commun et, pour chaque préfixe, seuls les suffixes sont stockés.

6.2 Présentation de la structure de données

À chaque k-mer est associé un vecteur de bits de taille N pour indiquer sa présence (1) ou son absence (0) dans chaque génome. Étant donné que les k-mers sont regroupés par des préfixes communs de longueur k_1, il existe 4^{k_1} groupes distincts (tableau (1) de la figure 6.1).

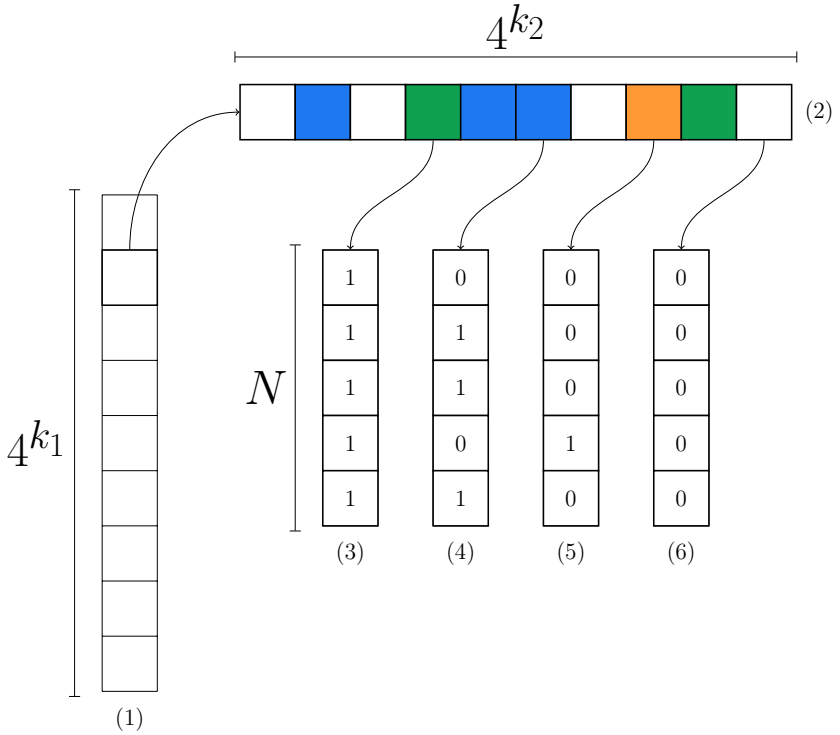


Figure 6.1: Représentation abstraite des k-mers indexés à partir de N génomes.

Pour chaque groupe, il existe 4^{k_2} suffixes possibles (tableau (2) de la figure 6.1). Chaque suffixe peut correspondre à un k-mer absent de tous les génomes indexés (cellules blanches), un *cloud* k-mer (cellules oranges), un *shell* k-mer (cellules bleues) ou bien un *core* k-mer (cellules vertes).
Le vecteur binaire associé à un *core* k-mer ne contient nécessairement que des bits à 1 (tableau (3) de la figure 6.1).
Le vecteur binaire associé à un *shell* k-mer contient nécessairement des bits à 0 et des bits à 1 (tableau (4) de la figure 6.1).
Le vecteur binaire associé à un *cloud* k-mer contient nécessairement un seul bit à 1 (tableau (5) de la figure 6.1).

Enfin, le vecteur binaire associé à un k-mer qui n'apparaît dans aucun génome ne contient nécessairement que des bits à 0 (tableau (6) de la figure 6.1).

Il est évident que les vecteurs ne contenant que de 0 ou que des 1, de même que les vecteurs ne contenant qu'un seul bit à 1 sont implicitement représentables.
Il semble assez évident également que représenter les 4^{k_2} suffixes de k-mers n'est pas opportun. En effet, il y a 4^{k_1} groupes de k-mers (le $i^{\text{ème}}$ groupe correspond à l'ensemble de k-mers dont le préfixe de longueur k_1 est le $i^{\text{ème}}$ dans l'ordre lexicographique). En moyenne, il y a $\frac{|\mathcal{K}|}{4^{k_1}}$ k-mers associé à chaque préfixe. Même pour de petites valeurs de k_1, cette valeur est très faible par rapport aux nombre de suffixes possibles ($=4^{k_2}$). Par exemple, en supposant que l'on souhaite indexer des 30-mers avec $\mathcal{N}$ de l'ordre de 10^9. Alors en posant $k_1 = 11$ (*cf.* théorème 4) cela représente un peu moins de 240 suffixes en moyenne par préfixe tandis que $4^{19} \simeq 2,7 \cdot 10^{11}$. Cela représente une densité de 1 dans le tableau (2) de la figure 6.1 de $8,7 \cdot 10^{-10}$. Ainsi, un vecteur de bits (même avec une structure de données succincte – *cf.* section 4.3) ne peut pas représenter efficacement le tableau (2) de la figure 6.1.
Cependant, les k-mers absents de tous les génomes sont implicitement obtenus par complément des k-mers indexés dans la structure et ne sont donc pas représentés physiquement.

Une distinction est faite entre *core* k-mers, les *shell* k-mers et les *cloud* k-mers (cellules resp. vertes, bleues et orange dans la figure 6.1). En effet, les *core* k-mers sont par définition présents dans tous les génomes et il n'est donc pas nécessaire de les associer explicitement au tableau binaire ne contenant que des bits à 1. Les *core* k-mers sont donc simplement représentés par un tableau contenant leurs suffixes. Par construction ce tableau est trié (tableau (A) de la figure 6.2) comme cela sera expliqué dans la section 6.3. Les *cloud* k-mers sont associés à un seul génome et donc sont autant de tableaux de suffixes (tableaux (C) de la figure 6.2). Ces tableaux sont également triés par construction. Enfin, les *shell* k-mers sont une structure composite comprenant d'une part le tableau des suffixes et d'autre part le tableau de vecteurs binaires associés (tableaux (B) de la figure 6.2). Ces tableaux sont triés par ordre lexicographique des suffixes également.

6.3 Fonctionnement de *RedOak*

Le choix de cette structure de données a été guidé par la volonté de permettre la mise à jour de l'index lors de l'ajout d'un génome sans avoir à reconstruire l'ensemble de la structure.
En effet, l'indexation d'un nouveau génome peut être représentée par certaines opérations ensemblistes simples, comme décrit dans l'algorithme 6.1.

6.3.1 Analyse de l'algorithme

Le fonctionnement itératif de l'ajout de génomes est illustré dans la figure 6.3.
Au départ, $N = 0$ et donc $\mathcal{G}_0 = \emptyset$. Il n'y a aucun k-mers indexés. Lors de l'ajout du génome G_1, cela correspond aux opérations des lignes 11 à 13 et l'on obtient bien la premère initialisation des *core* k-mers.
Lors de l'ajout du second génome, une partie des k-mers de G_1 qui étaient dans le *core* vont devenir les *cloud* k-mers de G_1. Il sont calculés à la ligne 15 (ensemble K') et supprimés aussitôt des *core* k-mers (ligne 16) tandis que ceux qui restent dans les *core* k-mers sont supprimés des k-mers restant à traiter (ligne 17). Enfin, les *cloud* k-mers de G_1 sont initialisés à partir de K' (ligne 19) et ceux de G_2 à partir des k-mers restant dans K (ligne 30).
Lors de l'ajout du troisième génome, à nouveau sont calculés les k-mers qui étaient dans le *core* et qui ne le sont plus désormais (ensemble K' calculé à la ligne 15 et supprimés aussitôt

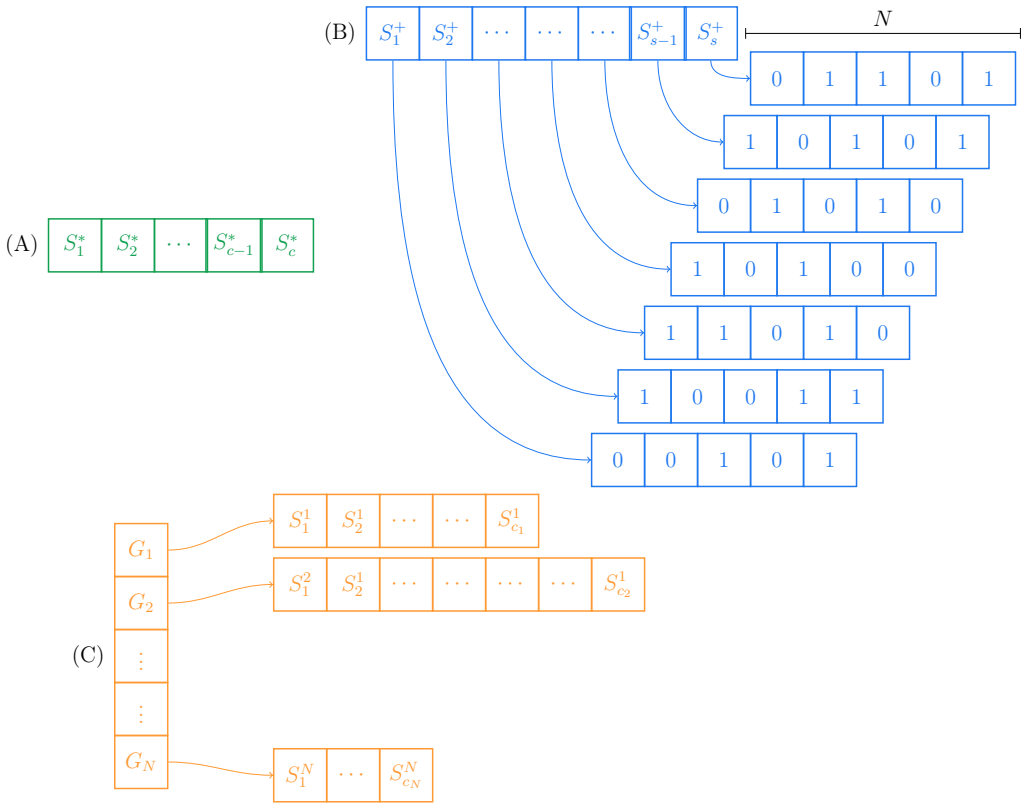


Figure 6.2: Représentation concrète des k-mers indexés de N génomes partageant le même préfixe. Les *core* k-mers apparaissent en vert, les *shell* k-mers en bleu, les *cloud* k-mers en orange.

des *core* k-mers à la ligne 16). Les k-mers de G_3 qui étaient déjà dans le *core* y demeurent et sont donc supprimés de K (ligne 17). Il n'y a pas de *shell* k-mers, donc l'opération de la ligne 21 est sans effet. Les k-mers qui sortent des *core* k-mers sont communs à tous les autres génomes précédemment indexés et sont donc ajoutés aux *shell* k-mers (ligne 22). Ensuite, les k-mers restant de G_3 (ensemble K) sont comparés aux *cloud* k-mers de G_1. Les k-mers communs (ensemble K' calculé à la ligne 24) deviennent des *shell* k-mers (puisqu'ils sont partagés dorénavant par 2 génomes). Ils sont donc respectivement supprimés des *cloud* k-mers de G_1 (ligne 25), des k-mers à traiter de G_3 (ligne 26) pour être ajoutés à $\mathcal{K}^+$ (ligne 27). L'opération est répétée avec les *cloud* k-mers de G_2. Au final, les k-mers restant dans K composent les *cloud* k-mers de G_3 (ligne 30).
La seule variation lorsque l'on ajoute G_4 (par rapport à l'ajout de G_3) apparaît lors du calcul des k-mers de G_4 qui figurent déjà dans les *shell* k-mers. Cette fois-ci, ces k-mers déjà présents dans $\mathcal{K}^+$ seront supprimés lors de la mise à jour de la ligne 21.

6.3.2 Détails sur les opérations ensemblistes

L'implémentation de l'algorithme 6.1 est basée sur le pré-requis que la liste des k-mers d'un génome (ou des k-mers associés à un préfixe donné) est ordonnée et que les opérations ensemblistes peuvent être réalisées en préservant la relation d'ordre. Ainsi, les k-mers récupérés à la ligne 9 sont supposés classés dans l'ordre lexicographique.

Les affectations des lignes 11, 19 et 30 préservent la relation d'ordre et peuvent s'effectuer en temps constant (permutation du contenu des ensembles) dans la mesure où l'ensemble en partie droite de l'affectation n'est plus utilisé jusqu'à la fin de l'algorithme dans ces trois cas. Les opérations de suppression d'éléments sont également réalisables sur place et en maintenant la relation d'ordre (pour représenter la relation d'ordre nous utiliserons les symboles $\succ$ et $\prec$).

algorithme 6.1: High level algorithm to incrementally update the index

```
1  Input :
2    K* % The core k−mers of G = {G_1, ··· , G_N} %
3    K+ % The shell k−mers of G = {G_1, ··· , G_N} %
4    K− = ⊎_{i=1}^{N} K^i % The cloud k−mers of G = {G_1, ··· , G_N} %
5    g % A new genome to add %
6  Output :
7    ⟨K*, K+, K− = ⊎_{i=1}^{N+1} K^i⟩ % The updated index of G ∪ {g} = {G_1, ··· , G_{N+1}} %
8  Begin
9    K ← {w|w is a k-mer of g}
10   If N = 0 Then
11     K* ← K % All k−mers are in core %
12     K+ ← ∅ % There is no shell k−mers %
13     K^1 ← ∅ % There is no cloud k−mers %
14   Else
15     K' ← K* \ K % Those k−mers are not in core anymore %
16     K* ← K* \ K' % Only core k−mers that are in g remains in core %
17     K ← K \ K* % Removing core k−mers from K %
18     If N = 1 Then
19       K^1 ← K' % Move old core k−mers to cloud k−mers of G_1 %
20     Else
21       K ← K \ K+ % The shell k−mers that are in g remains shell %
22       K+ ← K+ ⊎ K' % Moving old core k−mers to shell k−mers %
23       For i in {1, ··· , n}
24         K' ← K^i ∩ K % Those k−mers are both in G_i and g %
25         K^i ← K^i \ K' % So they are removed from the cloud of G_i %
26         K ← K \ K' % and from the cloud of g %
27         K+ ← K+ ⊎ K' % Finally they are added to the shell k−mers %
28       End For
29     End If
30     K^{N+1} ← K % Add remaining k−mers from g to its cloud k−mers %
31   End If
32   Return ⟨K*, K+, K− = ⊎_{i=1}^{N+1} K^i⟩
33 End
```

En effet, pour calculer $A \setminus B$, il suffit de parcourir les deux ensembles A et B linéairement et de comparer l'élément courant de chaque ensemble (respectivement e_A et e_B). Si $e_A \prec e_B$ alors il faut supprimer e_A de l'ensemble A. Si $e_A \succ e_B$, il faut passer à l'élément suivant pour l'ensemble B. Sinon, cela signifie que e_A et e_B sont identiques et donc il faut passer aux éléments suivants de A et de B. S'il ne reste plus d'éléments à parcourir dans B, l'élément e_A et les éléments suivants de A doivent être supprimés. Ainsi, les mises à jour des ensembles de k-mers aux lignes 16, 21, 17, 25 et 26 préservent également la relation d'ordre ; de même que le calcul des k-mers devant être supprimés des *core* k-mers (ligne 15) qui en revanche ne s'effectue pas sur place. L'opération d'intersection de la ligne 24 est assez similaire dans son principe puisqu'elle consiste à parcourir les éléments de deux ensembles ordonnés et ne conserver que les éléments communs, qui par conséquent seront également ordonnés.
Il n'y a que l'union menant à la constitution des *shell* k-mers (lignes 22 et 27) qui nécessitent un traitement particulier afin de pouvoir maintenir la relation d'ordre (la fusion de 2 ensembles ordonnées requiert de la mémoire additionnelle pour pouvoir être réalisée efficacement en temps).

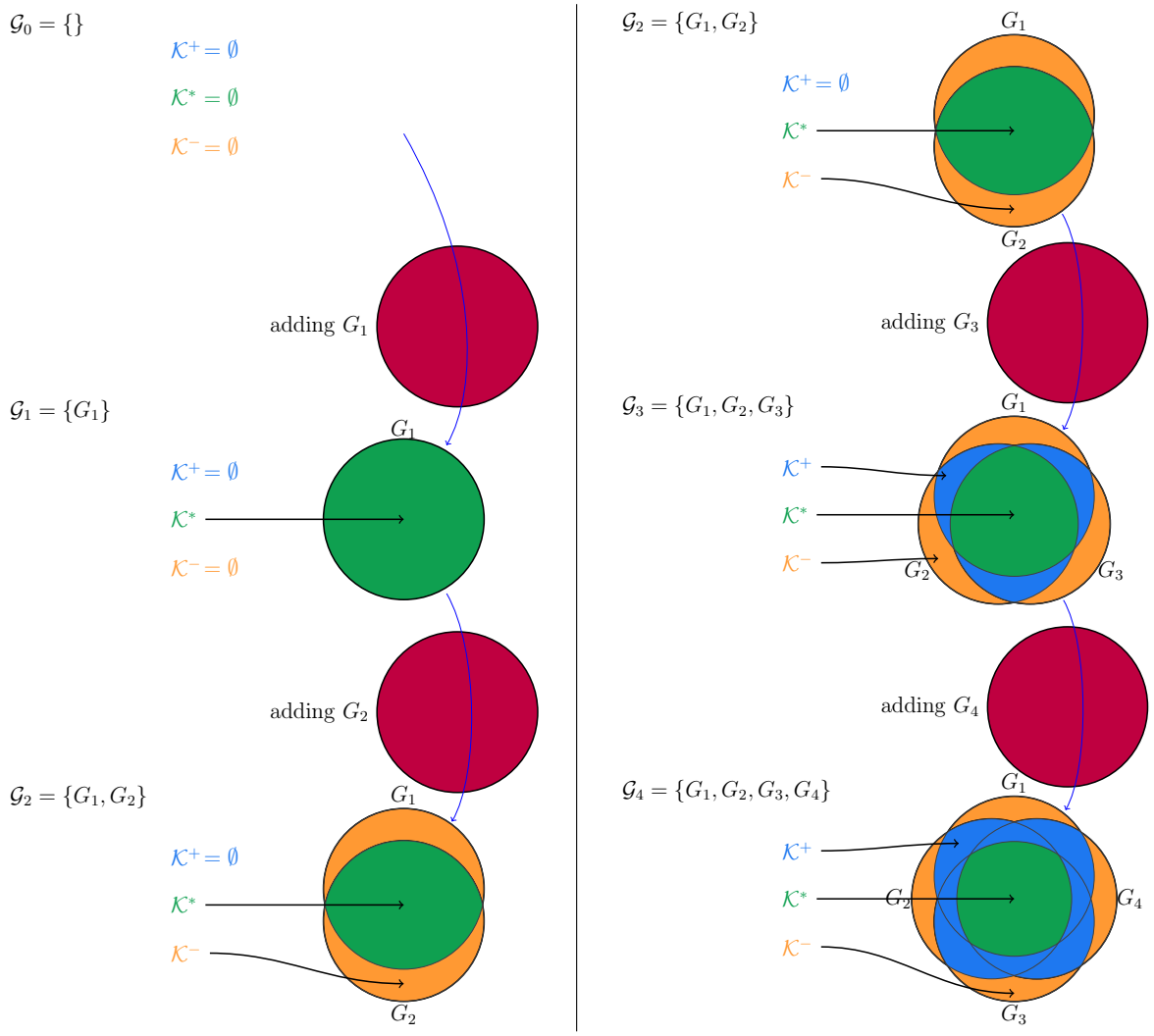


Figure 6.3: Illustration de l'évolution des k-mers indexés par *RedOak* au fur et à mesure de l'ajout des génomes.

Dans l'implémentation de *RedOak*, les tableaux de suffixes des *core*, des *shell* et des *cloud* k-mers sont représentés comme des suites contiguës de mots mémoire de μ' bits avec 2 bits par nucléotide. Il faut donc $\left\lceil \frac{2\,k_2}{\mu'} \right\rceil$ mots mémoire pour représenter un suffixe et $x \left\lceil \frac{2\,k_2}{\mu'} \right\rceil$ mots mémoire pour représenter x suffixes. Les bits inutilisés restants (ceux de poids faibles) sont positionnés à 0. Ce choix améliore considérablement le temps de comparaison entre les suffixes des k-mers. Les mots mémoire sont des `uint_fast8_t`, dont la taille est définie par le compilateur dans un souci d'optimisation des performances en temps.
Les tableaux binaires sont également représentés comme des suites contiguës de mots mémoire de μ' bits et requièrent donc $\left\lceil \frac{N}{\mu'} \right\rceil$ mots mémoire chacun.
Dans l'optique de pouvoir modifier la dimension des tableaux binaires, il est possible d'utiliser des librairies standard (classe `vector<bool>` du C++ par exemple) qui implémentent déjà le codage sur des mots binaires. Cependant, ces librairies standards incluent obligatoirement le stockage de la dimension du nombre d'éléments au sein de chaque structure/objet. Dans la mesure où tous les tableaux binaires sont ici de même taille N, il est évident qu'en raison du grand nombre de tableaux attendus, il est préférable de redéfinir une classe spécifique afin de diminuer l'empreinte mémoire sans perte de performance. Cette implémentation émule également les vecteurs de bits remplis 0, de 1 et ceux associés aux *cloud* k-mers de chaque génome (les tableaux (3), (5) et (6) de la figure 6.1).

En réalité, certaines opérations peuvent être réalisées simultanément. Par exemple, les opérations des lignes 11 à 13 ont pour objectif de calculer les 3 parties $\mathcal{K}^* \setminus K$, $\mathcal{K}^* \cap K$ et $K \setminus \mathcal{K}^*$. Cela peut donc être réalisé en parcourant les k-mers de $\mathcal{K}^*$ et ceux de K dans l'ordre et de comparer les éléments courants des deux ensembles ordonnés (resp. $e_{\mathcal{K}^*}$ et e_K). Si ce sont les mêmes k-mers, alors $e_{\mathcal{K}^*}$ reste dans le *core*, on passe à l'élément suivant pour $\mathcal{K}^*$ et e_K est supprimé de K. Si $e_{\mathcal{K}^*} \prec e_K$ alors $e_{\mathcal{K}^*}$ est déplacé de $\mathcal{K}^*$ vers K'. Enfin, dans le dernier cas de figure ($e_{\mathcal{K}^*} \succ e_K$) alors on passe à l'élément suivant de K. Si l'ensemble K a été intégralement parcouru, l'élément $e_{\mathcal{K}^*}$ et tous les suivants doivent être déplacés de $\mathcal{K}^*$ vers K'. Le même schéma s'applique pour les opérations des lignes 24 à 26.

6.3.3 Distribution et parallélisation des calculs

Comme expliqué précédemment, les vecteurs binaires associés aux *core* k-mers ainsi qu'aux *cloud* k-mers sont implicites. Ce n'est pas le cas des vecteurs binaires associés aux *shell* k-mers. Il faut donc pourvoir à la mise à jour de tous les vecteurs à chaque nouvel ajout de génome.
Afin d'éviter des phénomènes de réallocation mémoire, l'implémentation de *RedOak* permet de réserver pour chaque vecteur le nombre de mots mémoire nécessaires pour représenter N_{max} génomes (avec tous les bits positionnés à 0 par défaut). Ainsi, lors de l'ajout d'un génome, si $N \leq N_{max}$, il n'est pas besoin de modifier la taille des vecteurs associés aux k-mers. Si toutefois $N > N_{max}$, alors chaque vecteur est redimensionné (action effectuée en parallèle – *multithreading*). Dans tous les cas, par défaut, ces k-mers sont supposés absents du nouveau génome puisque le bit d'indice $N-1$ est par défaut à 0.
Les mise à jour des *shell* k-mers interviennent aux lignes 22 et 27. Dans le premier cas, les k-mers sont déjà dans le *shell*. Il suffit juste de positionner le bit à l'indice $N-1$ à la valeur 1. Dans le second cas, ce sont des k-mers qui étaient dans le *cloud* du génome G_i et qui vont devenir des *shell* k-mers car présents dans le génome G_i et le génome G_N. Il faut donc leur associer un nouveau vecteur binaire (tous les bits sont à 0 par défaut) et positionner les bits aux indices $i-1$ et $N-1$ à la valeur 1.

Il est important de noter que comme la structure de données partitionne l'ensemble des k-mers indexés en fonction de leur préfixe commun de taille k_1, l'algorithme 6.1 peut trivialement être adapté pour ne traiter que les *core*, les *shell* et les *cloud* k-mers associés à un préfixe donné.
Il devient donc facile de paralléliser l'algorithme présenté en indexant au sein d'un processus les k-mers associés à un sous-ensemble des préfixes (parallélisme lourd géré avec Open MPI). De même, au sein d'un processus, l'indexation des k-mers associés à un préfixe donné se parallélise facilement (parallélisme léger géré avec OpenMP).
Cela offre la possibilité d'exécuter *RedOak* sur un *cluster*, sur une architecture multi-cœurs ou sur une combinaison des deux.
Cette fonctionnalité présente deux avantages majeurs : la mémoire requise est répartie sur les instances en cours d'exécution, ce qui permet d'adapter la méthode à une très grande collection de génomes et de réduire considérablement le temps d'attente.

6.3.4 Les requêtes

La structure de données présentée dans cette partie a également une application intéressante : elle permet des requêtes simples et efficaces.
Rechercher une séquence s consiste à signaler, pour tous ses k-mers, dans quel(s) génome(s) ces k-mers apparaissent.
À partir de ce rapport, il est possible de calculer le nombre de k-mers de la séquence de

requête appartenant à chaque génome ou le nombre de bases couvertes par les k-mers de ces génomes.
Pour interroger la structure de données au sujet d'un k-mer, l'algorithme sélectionne le processus en charge du préfixe $pref$ du k-mer, puis recherche (par dichotomie) son suffixe dans les *cloud* k-mers $\mathcal{K}^{*}_{pref}$, les *shell* k-mers $\mathcal{K}^{+}_{pref}$, et les N parties composants les *cloud* k-mers $\mathcal{K}^{-}_{pref}$. La recherche est effectuée en parallèle dans ces sous-ensembles si *RedOak* a été compilé avec le support OpenMP. Dans le cas contraire, les ensembles sont parcourus dans l'ordre énoncé, les uns après les autres, jusqu'à ce que le k-mer ait été trouvé ou bien que tous les ensembles aient été parcourus.

6.4 Complexité

Dans cette partie, nous présentons la complexité de l'algorithme 6.1, en utilisant les notations ci-dessous :

$$\begin{cases} \mathcal{N} & \text{Nombre total de } k\text{-mers distincts } (= |\mathcal{K}|) \\ \mathcal{N}^{*} & \text{Nombre total de } \textit{core } k\text{-mers } (= |\mathcal{K}^{*}|) \\ \mathcal{N}^{+} & \text{Nombre total de } \textit{shell } k\text{-mers } (= |\mathcal{K}^{+}|) \\ \mathcal{N}^{-} & \text{Nombre total de } \textit{cloud } k\text{-mers } (= |\mathcal{K}^{-}|) \\ n & \text{Nombre d'instances s'exécutant en parallèle} \\ \mu & \text{Nombre de bits d'un mot mémoire (registre)} \end{cases}$$

Théoreme 5. *L'espace nécessaire pour indexer N génomes avec RedOak est égal à :*

$$2\,k_2\,\mathcal{N} + \mathcal{N}^{+}\,(N+\mu) + O\left(4^{k_1}\,N\right) \;\; \textit{bits.}$$

Si k_1 est défini comme étant $k_1 = \frac{\log\mathcal{N} - \log\log\mathcal{N} + O(1)}{2}$, alors l'espace mémoire requis par RedOak pour indexer les k-mers de N génomes est majoré par

$$\mathcal{N}\,(2\,k_2 + N) + o\,(N\,\mathcal{N}) \;\; \textit{bits.}$$

Démonstration. La structure associée à un préfixe de k-mer est identique à celle décrite à la figure 6.2.
Afin de stocker les suffixes de des k-mers associés au préfixe $pref$ (qu'il s'agisse des *core*, des *shell* ou bien des *cloud* k-mers) la structure utilise 2 bits par nucléotide, ce qui donne $2\,k_2\,\mathcal{N}_{pref}$ bits.
De plus, un vecteur binaire de taille N ainsi qu'un mot mémoire est associé à chaque suffixe du *shell* k-mers, soient $\mathcal{N}^{+}_{pref}\,(N+\mu)$ bits.
Ainsi pour un préfixe $pref$ donné, la structure requiert $2\,k_2\,\mathcal{N}_{pref} + \mathcal{N}^{+}_{pref}\,(N+\mu)$ bits.
À cela doivent être ajoutées les structures de données permettant d'encapsuler des informations, représentant donc $O\,(N)$ octets d'espace additionnel.
Enfin, un unique vecteur binaire de taille N est associé aux *core* k-mers, de même qu'un unique vecteur binaire de taille N est associé aux k-mers absents de la structure ainsi qu'un unique vecteur binaire de taille N est associé aux *cloud* k-mers de chaque génome G_i $(1 \leq i \leq N)$, soit un total de $(N+2)\;N + O(1)$ bits.
L'espace mémoire requis par *RedOak* pour indexer les k-mers de N génomes est donc de $2\,k_2\,\mathcal{N} + \mathcal{N}^{+}\,(N+\mu) + O\left(4^{k_1}\,N\right)$ bits.
En posant $k_1 = \frac{\log\mathcal{N} - \log\log\mathcal{N} + O(1)}{2}$, alors $4^{k_1} = O\left(\frac{\mathcal{N}}{\log\mathcal{N}}\right) = o\,(\mathcal{N})$. Il est aussi possible de remarquer que $\mathcal{N}^{+}\,(\mu+N) \leq N\,\mathcal{N} + o\,(N\,\mathcal{N})$. Ainsi, l'espace mémoire requis par *RedOak*

pour indexer les k-mers de N génomes est donc majoré par $\mathcal{N}\ (2\,k_2 + N) + o\,(N\,\mathcal{N})$ bits.
À cela, il faut ajouter un espace en $O\,(1)$ par nœud. Toutefois, il semble raisonnable de considérer que $n = o\,(N\,\mathcal{N})$. □

À l'instar de la librairie `libGkArrays-MPI`, pour des raisons de performance en temps, nous avons fait le choix d'utiliser des mots binaires de type `uint_fast8_t` pour le stockage des informations de suffixes ainsi que les tableaux binaires. En notant μ' le nombre de bits d'un entier de type `uint_fast8_t`, l'espace utilisé pour ce stockage est $\mu' \left\lceil \frac{2\,k_2}{\mu'} \right\rceil$ bits par suffixe et $\mu' \left\lceil \frac{NmN}{\mu'} \right\rceil$ par tableau binaire.

Théoreme 6. *Le temps nécessaire pour indexer les $\mathcal{N}$ k-mers distincts de N génomes avec RedOak est de*

$$O\,(N\,\mathcal{N}\,k) \quad .$$

Démonstration. Pour étudier le temps de construction de la structure de données, concentrons-nous sur le temps nécessaire pour ajouter l'ensemble des k-mers partageant un préfixe commun $pref$ provenant d'un nouveau génome G_{N+1} dans un index déjà existant de N génomes.
Notons cet ensemble K_{pref} et sa taille M_{pref}. Supposons que cet ensemble est déjà trié dans l'ordre lexicographique. Le calcul des parties ($\mathcal{K}^* \setminus K$, $\mathcal{K}^* \cap K$ et $K \setminus \mathcal{K}^*$) résultant de la comparaison entre $\mathcal{K}^*_{pref}$, trié par construction, et K_{pref} (lignes 15 à 17) nécessite $O\left(\frac{\mathcal{N}^*}{4^{k_1}} + M_{pref}\right)$ comparaisons de suffixes, soient $O\left(k_2\ \left(\frac{\mathcal{N}^*}{4^{k_1}} + M_{pref}\right)\right)$ opérations (*cf.* sous-section 6.3.2).
De même, le calcul des *shell* k-mers présents dans les candidats restants (lignes 21 et 22) se fait en temps linéaire sur la taille des deux ensembles donc en $O\left(k_2\ \left(\frac{\mathcal{N}^+}{4^{k_1}} + M_{pref}\right)\right)$.
Pour un génomes G_i, le calcul des parties résultant de la comparaison entre $\mathcal{K}^i_{pref}$ et les suffixes restant de K (donc $\mathcal{K}^i_{pref} \setminus K$, $\mathcal{K}^i_{pref} \cap K$ et $K \setminus \mathcal{K}^i_{pref}$; lignes 24 à 26) requiert $O\left(k_2\ \left(\frac{\mathcal{N}^-}{N\,4^{k_1}} + M_{pref}\right)\right)$.
La fusion des listes ordonnées $\mathcal{K}^+_{pref}$ et K' (ligne 27) se fait également en temps linéaire sur la taille des ensembles $O\left(k_2\ \left(\frac{\mathcal{N}^-}{N\,4^{k_1}} + M_{pref}\right)\right)$ et requiert un espace additionnel de $O\left(\min\left(\mathcal{K}^+_{pref}, M_{pref}\right)\right)$.
Ainsi, l'ensemble de ces opérations nécessite $O\left(k_2\ \left(\frac{\mathcal{N}}{4^{k_1}} + M_{pref}\right)\right)$ pour un préfixe $pref$ donné.
À cette complexité, il convient d'ajouter le temps nécessaire pour produire l'ensemble ordonné de k-mers groupés par préfixe. *RedOak* utilise l'implémentation `libGkArrays-MPI`, qui requiert un temps en $O\left(\frac{X\,k}{n}\right)$ pour indexer X k-mers (*cf.* théorème 2). Ce temps étant linéairement proportionnel au nombre de k-mers, il est possible d'imputer $O\left(\frac{M_{pref}\,k}{n}\right)$ opérations pour l'indexation des k-mers associés au préfixe $pref$. Il est raisonnable de considérer que cette grandeur est dans $O\,(k\,\mathcal{N})$.
Il est évident que $\mathcal{N}$ croît avec l'ajout de génomes et constitue un majorant du nombre de k-mers distincts d'un génome. Aussi la complexité de l'algorithme 6.1 pour l'ensemble des k-mers est dans $O\,(k\,\mathcal{N})$.
Il en découle que l'indexation des k-mers des N génomes se fait en $O\,(k\,N\,\mathcal{N})$.
En supposant que le nombre de génomes à indexer est connu *a priori*, il n'y a pas d'opération de réallocation des vecteurs binaires. L'allocation des vecteurs est donc faite une seule fois pour chaque *shell* k-mers ; or le nombre de *shell* k-mers ne fait que croître au fur et à mesure de l'ajout des génomes, donc l'allocation de l'ensemble des vecteurs binaires se fait en $O\,(N\,\mathcal{N}^+)$.
La complexité pour indexer N génomes avec *RedOak* est donc bien $O\,(k\,N\,\mathcal{N})$. □

Le temps d'exécution dépend clairement du nombre de k-mers indexés. En se basant sur un modèle statistique simple, il est possible de donner une approximation du nombre de k-mers distincts d'un ensemble de N génomes, et donc d'estimer la complexité temporelle de *RedOak*.

Théoreme 7. *En notant M le nombre moyen de k-mers par génome et en supposant que le nombre de génomes partageant un même k-mer suit une distribution de Poisson de paramètre λ (où λ est le nombre moyen de génomes partageant un k-mer), la taille de $\mathcal{N}$ est*

$$O\left(\frac{N\,M}{\lambda}\right) \quad .$$

Démonstration. Supposons que chaque génome possède M k-mers distincts et que chaque k-mer a une probabilité fixe p_i d'être partagée exactement par i génomes parmi N. Le nombre total de k-mers indexés est alors

$$\mathcal{N} = N \sum_{i=1}^{N} \frac{p_i\,M}{i} = N\,M \sum_{i=1}^{N} \frac{p_i}{i} \quad .$$

Dans le pire des cas, chaque k-mer est spécifique à chaque génome ($p_1 = 1$ et $p_{i,i\neq 1} = 0$), ce qui conduit à $\mathcal{N} = N\,M$. En revanche, le meilleur des cas se produit lorsque tous les k-mers sont dans le *core* ($p_N = 1$ et $p_{i,i\neq N} = 0$). Dans une telle situation, $\mathcal{N} = M$. Si la probabilité $p_i = p = \frac{1}{N}$, puisque $\sum_{i=1}^{N} \frac{1}{i} = \Theta\left(\log N\right)$, alors $\mathcal{N} = O\left(M\,\log N\right)$.
Affinons le modèle et notons λ le nombre moyen de génomes partageant un k-mer indexé ($1 \leq \lambda \leq N$). Les probabilités p_i suivent alors une loi de Poisson de paramètre λ et donc $p_i = \frac{\lambda^i}{i!}\,e^{-\lambda}$).
Ainsi,

$$\begin{aligned}\mathcal{N} &= N\,M \sum_{i=1}^{N} \frac{\lambda^i\,e^{-\lambda}}{i\,i!} \\ &= N\,M\,e^{-\lambda} \sum_{i=1}^{N} \frac{\lambda^i}{i\,i!} \quad .\end{aligned}$$

Rappelons que l'exponentielle intégrale $Ei(x) = \int_{-\infty}^{x} \frac{e^t}{t}\,\mathrm{d}t$ est telle que

$$Ei(x) = \gamma + \ln|x| + \sum_{i=1}^{\infty} \frac{x^i}{i\,i!}$$

où γ est la constante d'Euler-Mascheroni.
Rappelons également que le logarithme intégral $li(x) = \int_0^x \frac{1}{\ln t}\,\mathrm{d}t$ (avec $x \neq 0$) est tel que

$$\lim_{x\to+\infty} li(x) = O\left(\frac{x}{\log x}\right) \quad .$$

En notant que

$$Li(e^u) = Ei(u) \quad (\text{avec } x \neq 1)$$

on peut réécrire

$$\begin{aligned}\sum_{i=1}^{\infty} \frac{\lambda^i}{i\,i!} &= Ei(\lambda) - \gamma - \ln e^{\lambda} \\ &= li(e^{\lambda}) - \gamma - \lambda \\ &= O\left(\frac{e^{\lambda}}{\lambda}\right)\end{aligned}$$

Ainsi,

$$\mathcal{N} = N\,M\,e^{-\lambda}\,O\left(\frac{e^{\lambda}}{\lambda}\right) = O\left(\frac{N\,M}{\lambda}\right) \quad .$$

□

Théoreme 8. *Étant donné l'index généré avec RedOak de $\mathcal{N}$ k-mers distincts provenant de N génomes avec $k_1 = \frac{\log\mathcal{N} - \log\log\mathcal{N} + O(1)}{2}$, interroger l'index pour tous les k-mers de la séquence s nécessite $O\left(|s|\,\log\log\mathcal{N}\right)$ opérations.*

Démonstration. Extraire le premier k-mer de la requête s (et calculer son préfixe *pref* de taille k_1) nécessite $O\left(k\right)$ opérations. L'extraction des autres k-mers (et le calcul de leur préfixe) peuvent être effectués en $O\left(1\right)$ opérations chacun.
Ainsi, extraire tous les k-mers de la séquence de requête nécessite $O\left(|s|\right)$.
Comme déjà indiqué ci-dessus, pour chaque k-mer, il y a en moyenne $\frac{\mathcal{N}}{4^{k_1}}$ suffixes associé à son préfixe ; ainsi, effectuer une recherche dichotomique nécessite, en moyenne, $O\left(\log\frac{\mathcal{N}}{4^{k_1}}\right)$ opérations (*cf.* preuve du théorème 3).
Il faut donc $O\left(|s|\,\log\frac{\mathcal{N}}{4^{k_1}}\right)$ opérations pour effectuer la requête sur l'ensemble des k-mers de s.
Sous l'hypothèse que $k_1 = \frac{\log\mathcal{N} - \log\log\mathcal{N} + O(1)}{2}$, alors cette complexité devient $O\left(|s|\,\log\log\mathcal{N}\right)$.
□

6.4.1 Prédictions théoriques du coût d'indexation en mémoire de *RedOak*

Afin d'estimer plus précisément l'utilisation de la mémoire attendue par *RedOak* pour indexer N génomes avec n processus, il suffit d'une part de connaître la taille des objets définis dans le programme et d'autre part de pouvoir estimer le nombre de *core*, *shell* et *cloud* k-mers.

Sachant qu'il est possible d'estimer la taille des objets de chaque classe créés par *RedOak* sur la base de leurs attributs (*cf.* figure 6.4), il est donc possible d'approximer l'espace mémoire requis comme étant :

$$\begin{aligned}
& n\left|\texttt{redoak::Index}\right| \\
+\;& N\left(\left|\texttt{redoak::Genome}\right| + \left|\mathit{filename}\right|\right) \\
+\;& 4^{k_1} \times \left(\left|\texttt{redoak::ExtendedSuffixes}\right| + n\,N\left|\texttt{redoak::SpecificSuffixes}\right|\right) \\
+\;& \left(n\,(2+N) + \mathcal{N}\right)\left\lceil\frac{2\,(k-k_1)}{8\left|\texttt{gkampi::KMer::word_t}\right|}\right\rceil \\
+\;& \left(n\,(2+N) + \mathcal{N}^{+}\right)\left(\left|\texttt{redoak::GenomeBitVector}\right| + \left\lceil\frac{N}{\left|\texttt{gkampi::KMer::word_t}\right|}\right\rceil\right)
\end{aligned}$$

Sous l'hypothèse très réaliste que $k < N \ll \mathcal{N}^{+} \leq \mathcal{N}$, cette formule peut être approximée et

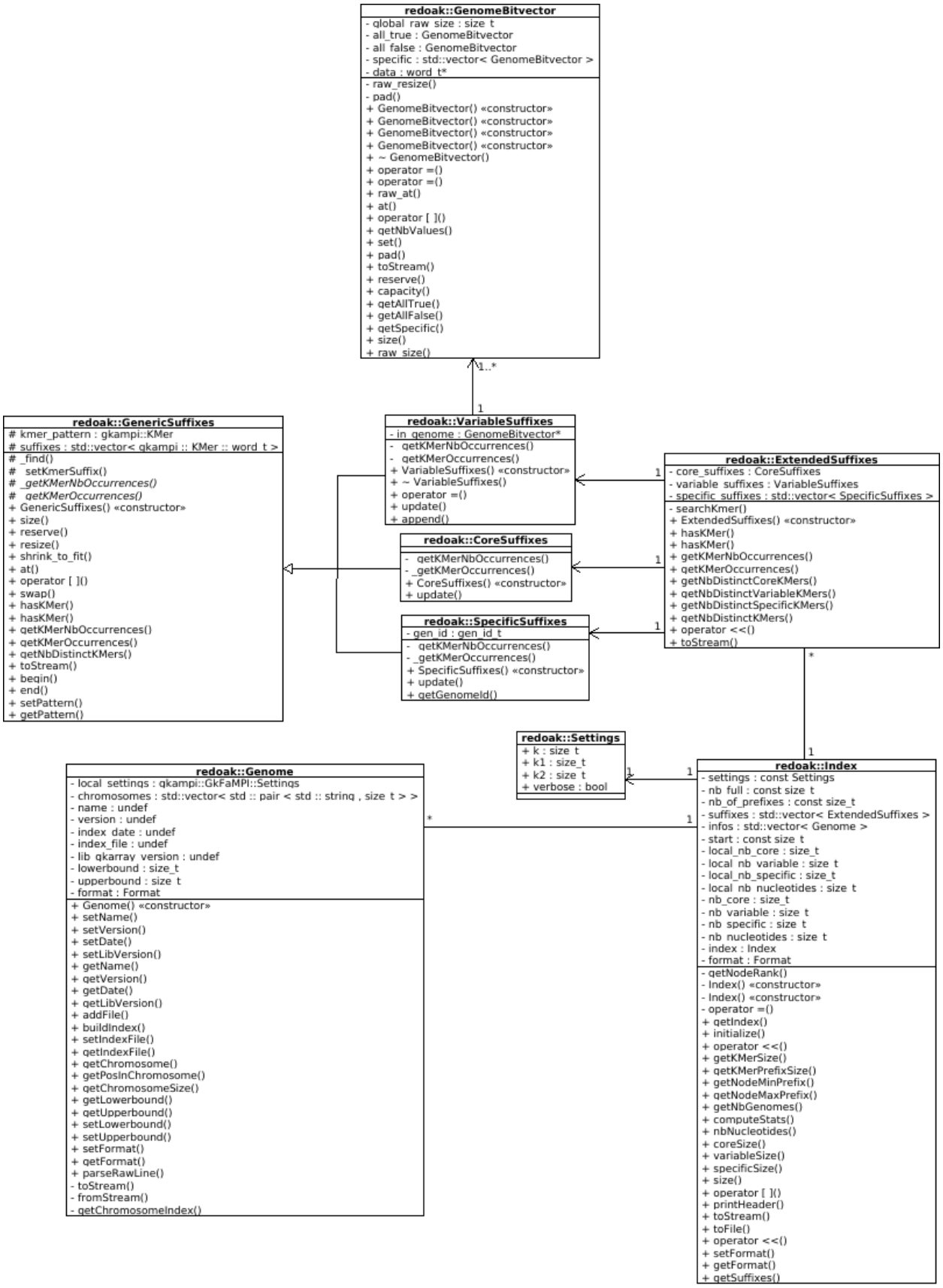


Figure 6.4: Diagramme des principales classes utilisées dans *RedOak*.

simplifiée considérablement en :

$$\begin{aligned} & 4^{k_1} \times \left(\left|\texttt{redoak::ExtendedSuffixes}\right| + N \left|\texttt{redoak::SpecificSuffixes}\right| \right) \\ + \quad & \mathcal{N} \left\lceil \frac{2\ (k-k_1)}{8 \left|\texttt{gkampi::KMer::word\_t}\right|} \right\rceil \\ + \quad & \mathcal{N}^+ \left(\left|\texttt{redoak::GenomeBitVector}\right| + \left\lceil \frac{N}{8 \left|\texttt{gkampi::KMer::word\_t}\right|} \right\rceil \right) \end{aligned}$$

En posant [1] $\begin{cases} \left|\texttt{redoak::ExtendedSuffixes}\right| & = 96 \text{ octets} \\ \left|\texttt{redoak::SpecificSuffixes}\right| & = 40 \text{ octets} \\ \left|\texttt{gkampi::KMer::word\_t}\right| & = 1 \text{ octet} \\ \left|\texttt{redoak::GenomeBitVector}\right| & = 8 \text{ octets} \end{cases}$
nous obtenons donc l'approximation suivante :

$$(6.1)\quad 4^{k_1} \times \left(40\,N + 96\right) + \mathcal{N} \left\lceil \frac{(k-k_1)}{4} \right\rceil + \mathcal{N}^+ \left(8 + \left\lceil \frac{N}{8} \right\rceil \right)$$

6.4.2 Prédictions de la mémoire requise pour l'indexation des génomes de riz par *RedOak*

Pour les 67 génomes de riz dont nous disposons, sachant que la taille d'un génome de riz est de l'ordre de $4 \cdot 10^8$ pb (donc le nombre de k-mers d'un génome est du même ordre), en estimant le nombre moyen de génome par k-mer entre 25 et 30, nous obtenons une estimation de $\mathcal{N}$ comprise entre $\frac{67 \times 4 \cdot 10^8}{30}$ et $\frac{67 \times 4 \cdot 10^8}{25}$ (*cf.* théorème 7), soit un nombre moyen de k-mers compris entre $8,9 \cdot 10^8$ et $1,1 \cdot 10^9$.

Posons $\mathcal{N}^+ = \mathcal{N} = 10^9$, alors en appliquant la formule Équation 6.1, en fixant $k = 30$ et $k_1 = 11$ nous obtenons une mémoire totale de l'ordre de 31 GiB, à répartir sur les n processus. Ceci représente donc un coût moyen de 270 bits par k-mer indexé, ou encore un coût de moins de 20 bits par nucléotide lu en entrée. Ceci est bien évidemment une estimation assez grossière et d'après nos résultats (*cf.* chapitre 7), pour indexer les 67 génomes, nous avons obtenu un index utilisant 336 bits par k-mer indexé, soit 29 bits par nucléotide traité, les résultats semblent donc indiquer qu'il faille majorer l'estimation théorique de 50%.
Cet aspect sera développé plus en détail au chapitre suivant.

1. Les mesures ont été réalisées sur le code compilé sur plusieurs ordinateurs différents, avec différentes versions de `g++`, sur des architectures 64 bits tournant sous GNU/Linux.

CHAPITRE 7

Validation, test et résultats biologiques

Dans ce chapitre sont présentées les performances du logiciel *RedOak*, logiciel qui a été introduit au chapitre 6. L'évaluation des performances en temps d'indexation et en temps de requêtes sont effectuées sur un *cluster* de calcul de type SGE. Nous présenterons un *benchmark*, c'est-à-dire une étude comparative avec d'autres outils existants. Cette étude comparative sera faite sur l'indexation d'une large collection de génomes. Seront évalués, premièrement le temps et l'espace nécessaires pour l'indexation et deuxièmement le temps nécessaire pour effectuer un certain nombre de requêtes.
Nous illustrerons finalement les requêtes qu'il est possible de faire avec *RedOak* sur un exemple biologique concret. Dans la dernière partie de ce chapitre, nous montrons les résultats qu'il est possible d'avoir avec *RedOak*.

7.1 Les performances

Les expériences ont été effectuées sur un *cluster* SGE sous Debian. Le *cluster* (`SGE` 8.1.8) a deux files d'attente. La file d'attente « normale » possède 23 nœuds, avec 196 GiB de RAM et 48 cores [1] chacun.
Cette file d'attente représente 4, 4 TiB de mémoire et 1104 cœurs. La file d'attente « bigmem » possède 1 nœud, disposant de 2 TiB de RAM et de 96 cores [2].
Les expériences ont été réalisées à la fois sur les files d'attente « bigmem » et « normale ».

1. Intel ® Xeon ® CPU $E5-2680$ processeur $v3$ cadencé à $2,50$ GHz.
2. Intel ® Xeon ® CPU $E7-4830$ processeur $v3$ cadencé à $2,10$ GHz.

Notre ensemble de données est composé de 66 accessions de riz (*cf.* figure 4.1) et du génome assemblé IRGSP (*cf.* table 4.1). Cet ensemble de fichiers non-compressés représente 25 GiB de fichiers textes. Pour simplifier la lecture nous parlerons par la suite de 67 génomes de riz. Ces 67 génomes représentent 26 194 967 769 nucléotides.

Indexation d'une large collection de génomes non assemblés. Nous avons souhaité connaître les limites en pratique de notre outil d'indexation. Nous avions évalué les limites de manière purement théorique et il nous a semblé judicieux de tester notre outil sur l'ensemble des données disponibles. Nous avons à notre disposition 3000 accessions de riz sous forme de fichiers FASTQ compressés et qui représentent 12 TiB de données.
En utilisant les lectures des fichiers FASTQ compressés, *RedOak* a pu indexer un sous-ensemble de 110 génomes non assemblés choisis au hasard dans le projet 3000 génomes de riz. Avec un paramétrage précis du *cluster*, *RedOak* a pu s'exécuter sur 140 instances en parallèle réparties équitablement sur 14 nœuds, chacun des nœuds utilisant 10 cœurs. Il a utilisé un total de 47 254 459 secondes en temps CPU ($\simeq$ 547 jours de calcul et 3,90 jours effectif) et $\simeq$ 683 GiB de mémoire RAM.

7.1.1 À propos de l'implémentation de *RedOak*

Cette étude pratique, a mis en avant, par la mémoire mesurée, une anomalie. Cette anomalie nous a amené à revoir l'implémentation de *RedOak*. Dans la version 0.0.0 de *RedOak* (premier prototype fonctionnel, sorte de version α) qui correspond à l'expérience que nous venons de décrire, la gestion des *shell* k-mers faisait intervenir une structure additionnelle pour gérer l'ordre lexicographique. Cette structure additionnelle avait pour but de permettre d'optimiser le temps de calcul lors des mises à jour.
Il se trouve que cette optimisation de temps étant réellement coûteuse en mémoire, celle-ci a été supprimée et de nombreux correctifs ont été appliqués. Cette mise à jour a donné la version 0.1.0 de *RedOak*.
En poursuivant les expérimentations, nous nous sommes aperçus que la consommation mémoire était encore trop élevée et qu'elle était notamment due à la gestion des tableaux de suffixes. En effet, jusqu'à la version 0.1.0, celle-ci était assurée avec la classe `gkampi::Suffixes`, qui pour chaque instance possède un attribut `gkampi::KMer`. La présence de cet attribut est complètement fondée dans le cadre de l'implémentation de la `libGkArrays-MPI` mais est très sur-consommatrice de mémoire dans le cas de *RedOak*. Par conséquent, la gestion des tableaux de suffixes a été revue et corrigée dans la version 0.1.1, ce qui a permis par ailleurs d'améliorer également les temps de calcul puisque certaines opérations (dans le cas de la fusion des listes de *shell* k-mers) sont faites sur place (en utilisant la mémoire déjà allouée) au lieu de passer par des structures temporaires additionnelles.

Pour comparer efficacement les deux versions de *RedOak*, nous avons mis a contribution le *cluster Lamarck* (*cf.* description à la section 5.5) et nous avons indexé les 67 génomes de riz.

La figure 7.1 nous indique très clairement que la dernière version au moment de la rédaction de ce manuscrit de *RedOak* est plus performante que la version 0.1.0. Le gain est considérable en temps et en espace mémoire. Pour 67 génomes, la mémoire pour un nœud est de 6,2 GiB pour la version 0.1.0 contre 3,2 GiB pour un nœud pour la version 0.1.1, donc une réduction d'un facteur de presque 2. Pour le temps d'indexation, il est difficile d'interpréter les courbes, il y a des fluctuations (qui dépendent très probablement de la charge du *cluster*). La version 0.1.1 permet de diviser l'empreinte mémoire par un facteur 2 et de réduire significativement le temps de calcul.

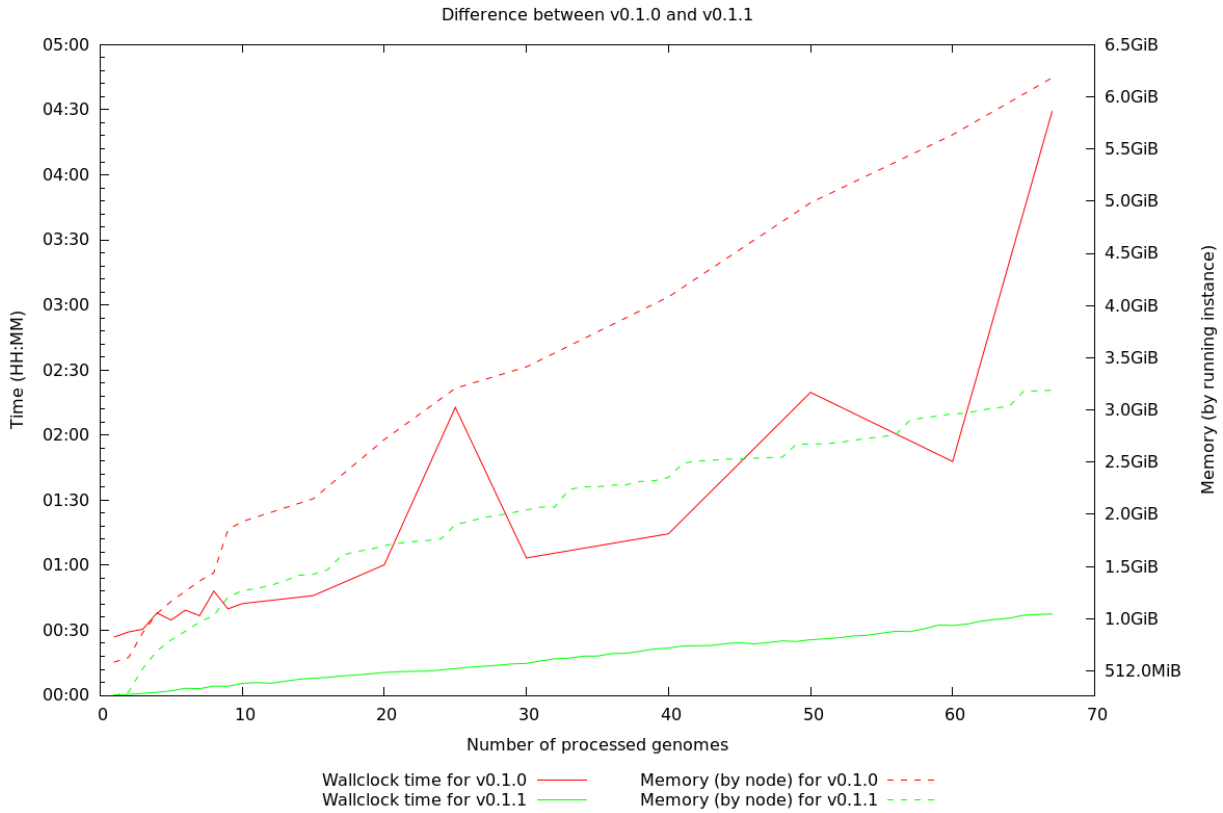


Figure 7.1: Comparaison des performances d'indexation en temps et en mémoire RAM de la version 0.1.1 et la version 0.1.0 de *RedOak*.

L'ensemble des expérimentations qui vont suivre dans cette section ne concernent que la version v0.1.1 de *RedOak*.

7.1.2 Étude de l'impact de la taille du préfixe en nucléotide sur *gkampi*

Nous avons décidé d'étudier l'impact de la taille du préfixe en nucléotide en utilisant *gkampi* sur le comptage de k-mers. Il aurait été possible d'effectuer cette étude avec *RedOak* mais nous nous sommes focalisés sur le comptage et pas l'indexation des génomes qui comprend une étape supplémentaire, celle de la fusion des comptes de k-mers.

Dans un premier temps nous mesurons la performance des calculs effectué par le processeur, donc nous étudions le gain lié à la parallélisation OpenMP couplée avec Open MPI.

La figure 7.2 nous présente six boîtes à moustaches. Parmi ces six boîtes à moustaches la taille du préfixe pour laquelle *gkampi* a été le plus performant est la taille de préfixe égale à 9 nucléotides, et le moins performant étant avec une taille de préfixe de 11 avec respectivement une médiane à un peu plus de 600% et une médiane 475%. Il semble intéressant de noter que l'ensemble des boîtes à moustaches, pour les préfixes allant de 9 à 14 nucléotides, varie avec une faible amplitude de $\pm 100\%$. Ces résultats ont été obtenues sur un seul nœud de calcul avec 8 processeurs donc l'optimum devrait être d'au plus 800%. Comme la parallélisation légère est faite avec OpenMP et que l'attribution des cœurs par SGE n'est pas très rigoureuse, on peut parfois avoir un peu plus de temps CPU fourni par le *cluster*, ce qui explique certains dépassements au delà des 800%. Mais un rapport de performance de 600% ou même de 500% est parfaitement acceptable.

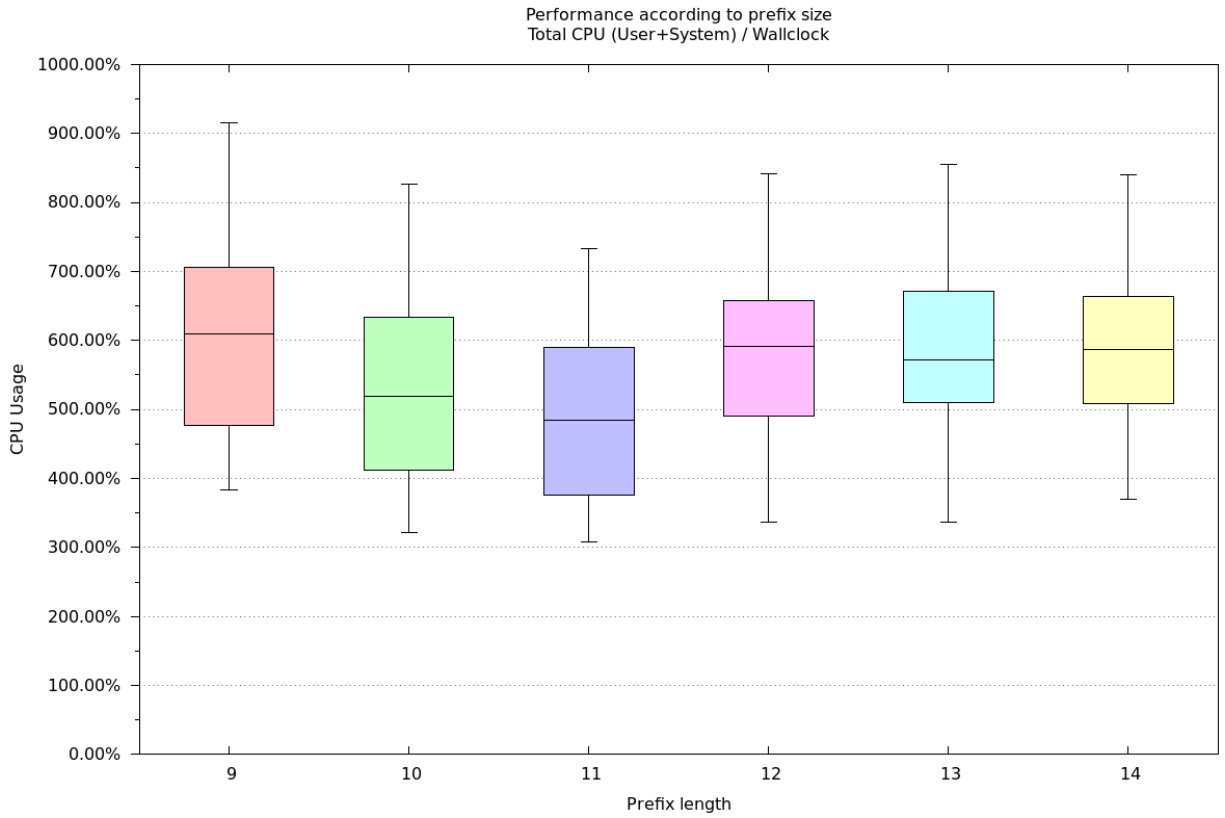


Figure 7.2: Utilisation des ressources des processeurs en fonction des différentes taille de préfixes variants de 9 nucléotides à 14.

Ensuite, nous avons mesuré la mémoire utilisée en fonction de la taille des préfixes en nucléotides variants de 9 à 14. Dans la figure 7.3, il est notable qu'indexer un génome requiert plus de mémoire avec un préfixe de taille 9 nucléotides que de 10 nucléotides. Il est également intéressant de noter que pour des tailles de préfixes entre 10 et 11 nucléotides, le besoin en mémoire est approximativement le même. Une autre observation notable est qu'au delà d'un préfixe de taille 11 nucléotides, les besoins en mémoire augmentent. Ce que les boîtes à moustache nous permettent d'évaluer rapidement c'est que la variation entre les génomes est faible pour une taille de préfixe de 11 nucléotides, 25% des génomes nécessitent une mémoire allant de 208 à 224 MiB, 25% des génomes nécessitent une mémoire allant de 224 à 232 MiB, 25% des génomes nécessitent une mémoire allant de 232 à 256 MiB et finalement 25% des génomes nécessitent une mémoire allant 256 à 288 MiB. Pour le comptage d'un génome de riz, il semblerait judicieux en terme de mémoire d'opter pour une taille de préfixe de 10 ou 11 nucléotides.

Enfin, nous nous sommes intéressés au temps d'horloge nécessaire pour le comptage des k-mers pour les 67 génomes. Nous réalisons grâce à la figure 7.4 qu'il n'y a pas beaucoup de variations du temps moyen. Ce qui diffère c'est davantage l'amplitude de la variation du temps d'horloge. Il faut environ 6 minutes par génome pour un préfixe de 9, 10 ou 11 nucléotides et environ 8 minutes par génome pour un préfixe de 12 ; 13 ou 14 nucléotides. Il est important de souligner que pour les tailles de préfixe de 12, 13 et 14 nucléotides, certains génomes ont mis entre 20 et 25 minutes pour être indexés.

Ce que ces trois figures nous indiquent, c'est que sur le comptage des k-mers des 67 génomes de riz, le meilleur compromis pour le choix du préfixe semble être situé entre 10 et 11 nucléotides.

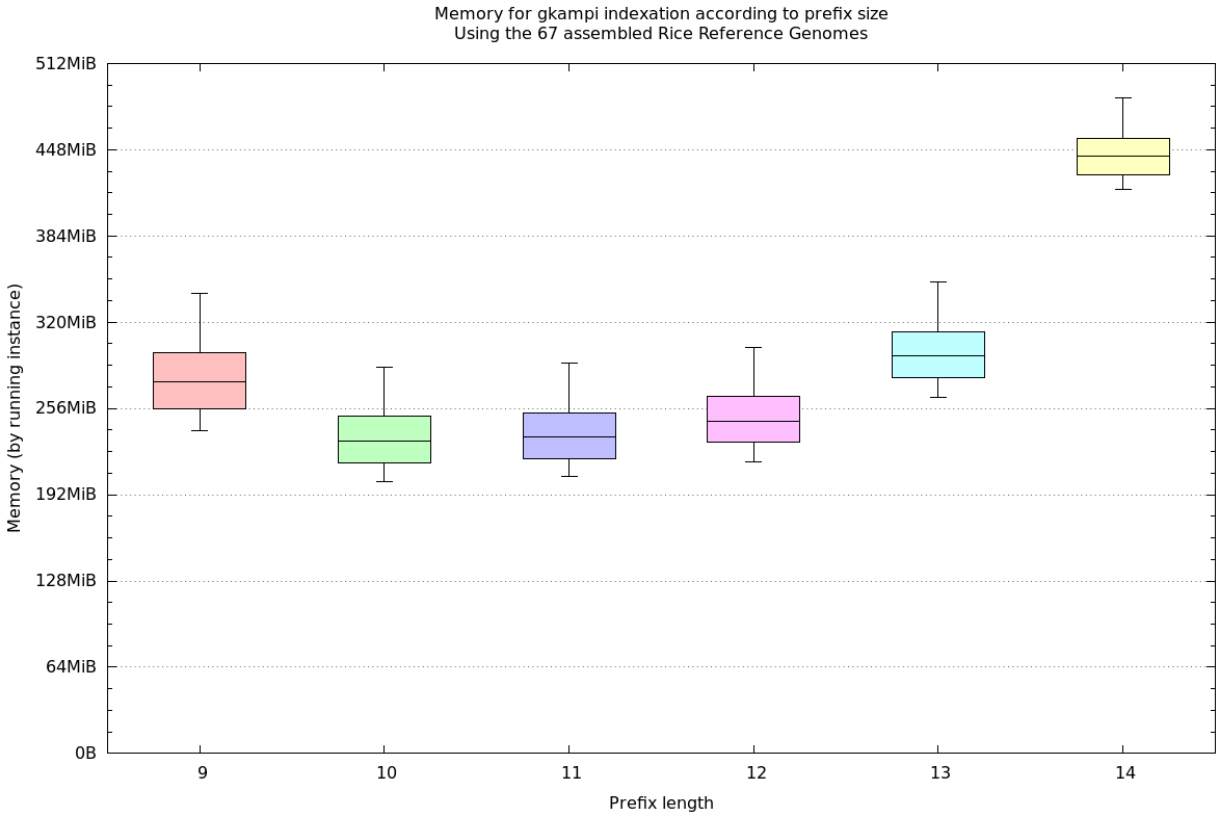


Figure 7.3: Utilisation de la mémoire en fonction des différentes tailles de préfixes variants de 9 nucléotides à 14.

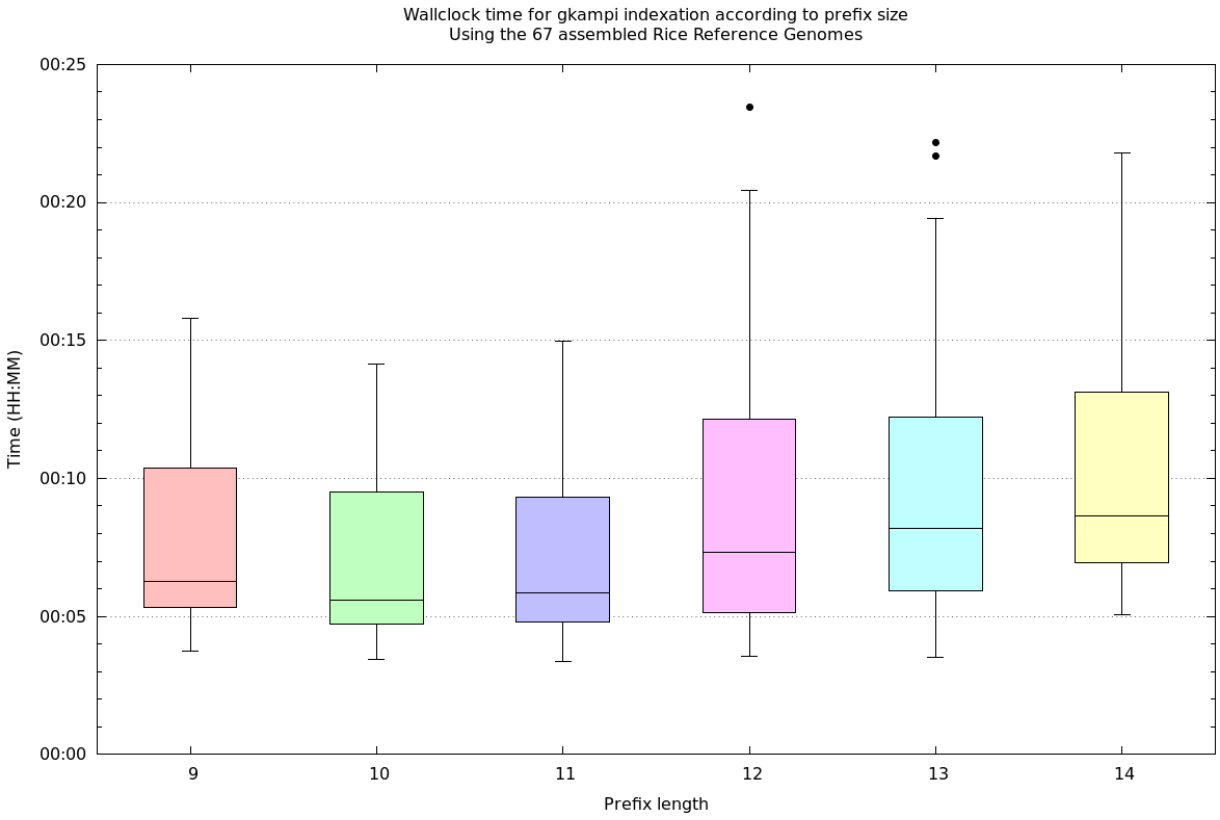


Figure 7.4: Variations du temps en fonction des différentes tailles de préfixes variants de 9 nucléotides à 14.

D'après la formule du théorème 4 pour un génome d'environ $500Mpb \simeq 5 \times 10^8$ k-mers

$$\begin{aligned} k_1 &= \frac{\log_2 5 \times 10^8 - \log_2 \log_2 5 \times 10^8 + O(1)}{2} \\ &= \frac{28.9 - 4.85 + O(1)}{2} \\ &= \frac{24 + O(1)}{2} \\ &= 12 + O(1) \end{aligned}$$

Dans la sous-section précédente nous avons conclu qu'expérimentalement il fallait un préfixe entre 10 et 11 ; la formule nous aiguille sur un préfixe de 12 nucléotides. L'écart constaté tient *a priori* à la valeur de la constante $O(1)$ que nous avons négligé ici.

7.1.3 Étude de l'impact de la taille du préfixe en nucléotide sur *RedOak*

L'étude suivante porte sur l'indexation des 67 génomes. Cette étude est réalisée sur le *cluster Lamarck* avec 2 nœuds et 20 instances par nœud. Les 67 génomes ont été au préalable indexés par *gkampi*. L'indexation préalable par *gkampi* permet ainsi de ne mesurer que la mémoire et le temps nécessaire au chargement de l'index du génome et son intégration dans *RedOak*. Ceci a pour effet d'accentuer le contraste entre les deux versions et de mesurer le bénéfice apporté par la réimplémentation de la gestion des suffixes.

Sur la figure 7.5 il est notable que le temps de calcul basé sur le temps d'horloge, représenté par des lignes pleines, pour indexer n génomes est globalement linéaire par rapport à n. Les perturbations sur la fin de la courbe correspondant aux préfixes de taille 11 nucléotides, représentée par la ligne verte pleine, sont probablement dues à une surcharge du *cluster*.
L'espace mémoire, représenté par des lignes pointillées, utilisé pour indexer n génomes est sous-linéaire par rapport à n. Les paliers sont dus au codage des vecteurs binaires sur 1 octet. Donc tous les 8 génomes, *RedOak* utilise 1 octet de plus par vecteur binaire.
Les meilleurs temps de calcul (temps d'horloge) sont atteints pour une valeur de préfixe entre 11 et 12. L'occupation mémoire augmente avec la taille du préfixe. À partir de 25 génomes avec un préfixe à 14, il n'y avait plus assez de ressource mémoire disponible pour effectuer le calcul sous cette configuration. Il est bien évidemment possible de répartir le calcul sur plus de nœuds.

Nous appliquons ici également la formule du théorème 4 avec les 2 207 262 175 k-mers indexés (donc $2,2 \times 10^9$ k-mers).

$$\begin{aligned} k_1 &= \frac{\log_2(2,2 \cdot 10^9) - \log_2 \log_2(2,2 \cdot 10^9) + O(1)}{2} \\ &= \frac{31,035 - 4,96 + O(1)}{2} \\ &= \frac{26 + O(1)}{2} \\ &\simeq 13 + O(1) \end{aligned}$$

L'écart constaté tient ici encore *a priori* à la valeur de la constante $O(1)$ que nous avons négligé.

Ces expériences semblent indiquer qu'une bonne valeur pour le choix de la longueur du préfixe est $\frac{\log_2 \mathcal{N} - \log_2 \log_2 \mathcal{N}}{2} - 2$ pour indexer $\mathcal{N}$ k-mers.

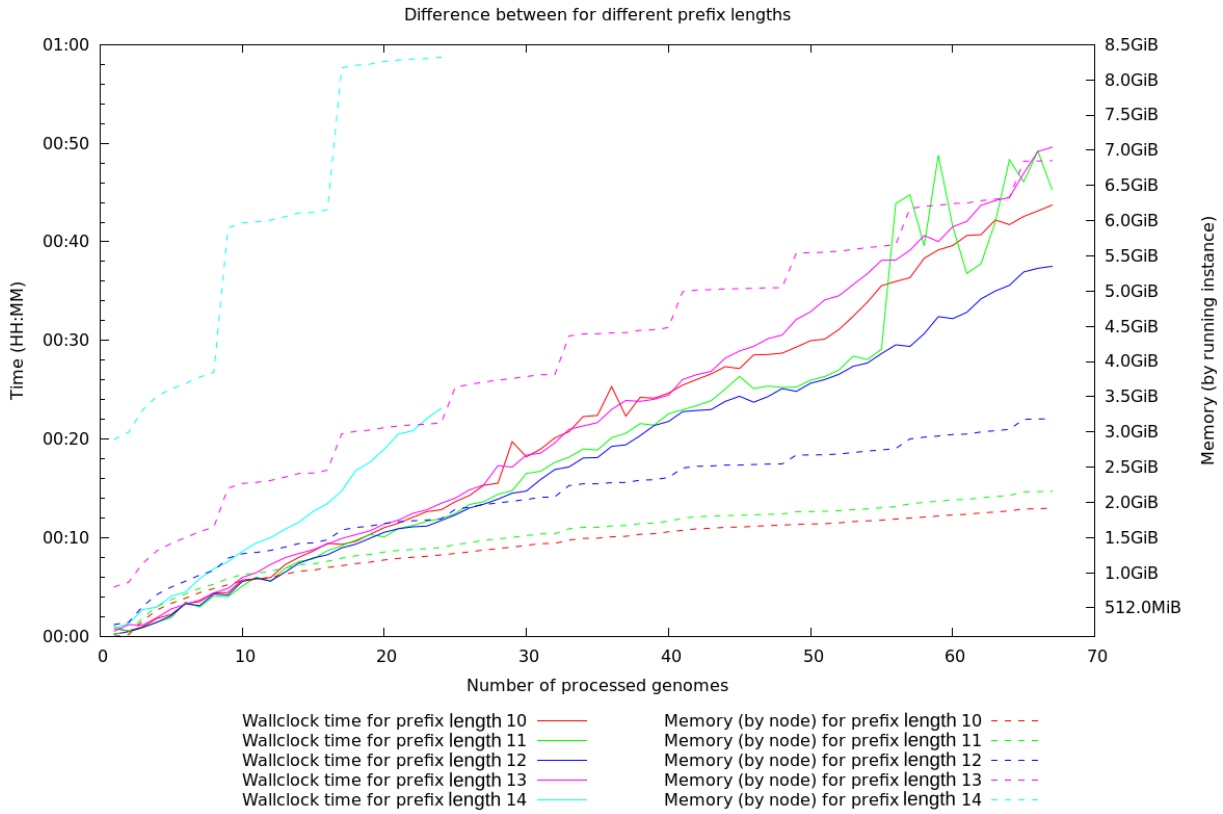


Figure 7.5: Indexation réalisée avec 2 nœuds et 20 instances par nœud à partir des génomes préalablement indexés par *gkampi*.

7.1.4 Évolution de l'index en fonction du nombre de k-mers indexés

La figure 7.6 illustre les proportions qu'occupent les *core* k-mers, les *shell* k-mers et les *cloud* k-mers en fonction du nombre de génomes indexés. Il semblerait qu'a partir de 10 génomes il y a une stabilisation de la répartition. Cette figure est dépendante de l'ordre d'ajout des génomes mais l'allure globale et les valeurs des pourcentages pour 67 génomes nous informe du pourcentage qu'occupent les *core*, *shell* et *cloud* k-mers. Il est important de noter la fluctuation lors de l'ajout des 10 génomes de *rufipogon*. Comme en valeur absolue le nombre de *shell* k-mers ne fait qu'augmenter, cela signifie que chaque génome de *rufipogon* comporte un nombre plus significatif de *core* k-mers que les autres variétés. Ceci est assez cohérent lorsque l'on observe les longueurs de branches des *rufipogon* par rapport aux autres variétés de l'arbre phylogénétique présenté à la figure 4.1.

Dans la figure 7.7 il est intéressant de noter l'utilisation de la mémoire pour une taille de préfixe de 13 nucléotides, tous les 8 génomes il y a un pic de consommation mémoire. Les valeurs optimales (celle qui nécessitent le moins de bits par k-mer) sont celles correspondant à un préfixe de taille 10 ou 11 nucléotides.

La figure 7.8 nous montre que le nombre de bits par nucléotide traité décroît et ce, peu importe la taille en nucléotide du préfixe. Plus le nombre de génome augmente plus le nombre de bits par nucléotide diminue jusqu'à atteindre un plateau de ± 25 pour l'indexation faite avec des prefixes de taille de 10-11 nucléotides, ± 50 pour une taille de 12 nucléotides et ± 100 pour un préfixe de taille 13 nucléotides.

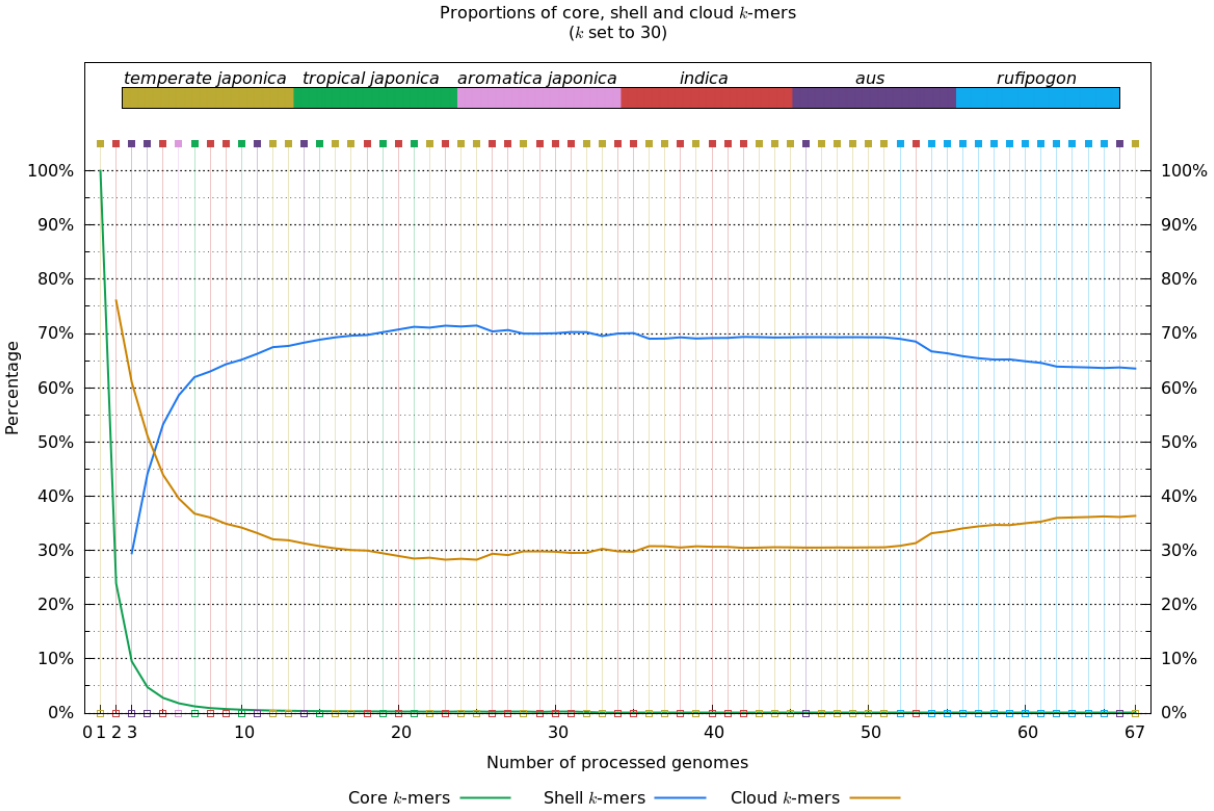


Figure 7.6: Évolution du pourcentage de *core*, *shell* et *cloud* k-mers en fonction du nombre de génome. Pour cette figure, nous avons fixé k =30 et les couleurs correspondent à celles de la phylogénie proposée dans l'article des 66 génomes de riz [Zhao et al., 2018] (et reproduite à la figure 4.1). Ici la taille du préfixe n'a pas d'influence, on travaille avec le nombre de k-mers de chaque catégorie et le nombre de k-mers n'est pas lié à la taille du préfixe

7.1.5 Synthèse des résultats

Les figures 7.2, 7.3 et 7.4 correspondent à l'indexation individuelle des 67 génomes à partir des fichiers au format fasta non compressés. Chaque génome a été indexé sur 1 nœud avec 8 instances en parallèle en faisant varier la taille du préfixes pour l'indexation.

Sur la figure 7.4 on observe que le meilleur temps d'horloge pour l'indexation est obtenu pour un préfixe entre 10 et 11 avec une moyenne aux environs de 6 minutes par génome. Sur la figure 7.3 on observe que la meilleure occupation mémoire pour l'indexation est obtenue pour un préfixe entre 10 et 11 également. Sur la figure 7.2 on observe que la meilleure performance (relative) du CPU est atteinte pour un préfixe à 9 et les moins bonnes pour un préfixe entre 10 et 11. Ceci étant, sur 8 instances, on peut difficilement espérer mieux que 800% et on est tout de même à 500%, ce qui est très convenable. Les fluctuations sont très probablement dues aux variations de la charge du *cluster*, ainsi qu'à son hétérogénéité.

Les figures 7.7 et 7.8 nous confirment que pour l'indexation des génomes de riz, il est préférable d'utiliser un préfixe de taille 10 ou 11 nucléotides. De ces deux figures et de la figure 7.6, nous notons un palier à partir de $\pm$10 génomes indexés.

De ces résultats, on peut supposer que le bon compromis pour indexer les génomes de riz serait d'utiliser des préfixes de taille 11 nucléotides, ce qui engendrerait un léger surcoût de mémoire par rapport à un préfixe de taille 10, en contrepartie d'un gain de temps.

Il serait même envisageable de proposer un outil d'analyse qui détermine, en fonction d'un

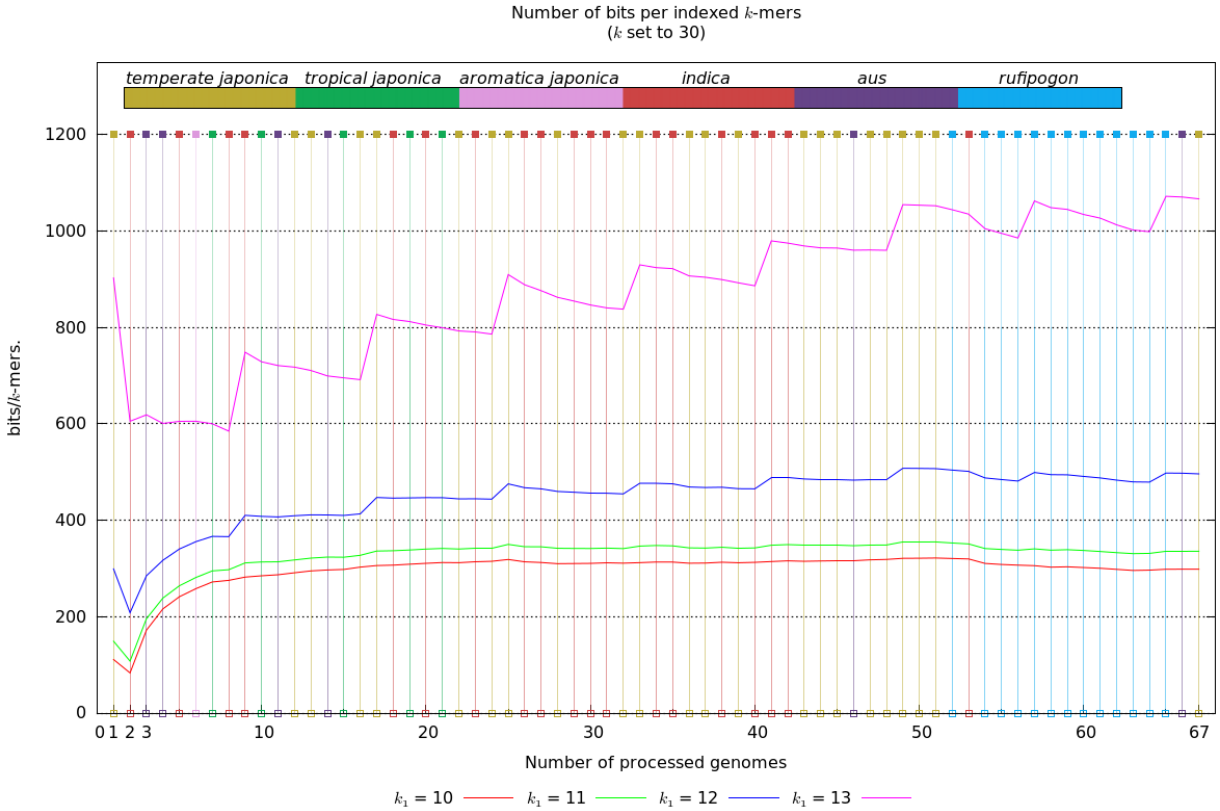


Figure 7.7: Étude du nombre de bits par k-mer pour différentes tailles de préfixe allant de 10 à 13 en fonction des génomes. Pour cette figure, nous avons fixé k =30 et les couleurs correspondent à celles de la phylogénie proposée dans l'article traitant les 66 génomes de riz [Zhao et al., 2018] (et reproduite à la figure 4.1).

jeu d'essais représentatif, la taille idéale du préfixe en nucléotides pour nos données.

Cet outil reprendrait en partie le protocole expérimental présenté dans cette section, à savoir indexer quelques génomes afin d'estimer les proportions de *core* k-mers, *shell* k-mers et *cloud* k-mers, puis sur cette base, estimer le nombre total de k-mers attendus pour enfin appliquer la formule du théorème 4.

7.2 Comparaison avec les autres outils

Nous avons téléchargé, installé et utilisé CHICO [Valenzuela, 2016] sur un ensemble de fichiers FASTQ extraits pour seulement 10 génomes du projet des 3 000 génomes de riz [Li et al., 2014]. Les tests de CHICO ont été effectués avec les mêmes conditions que l'indexation réussie des 110 génomes faite avec la version 0.0.0 de *RedOak* sur le *cluster* du Cirad. Bien que les tests aient été réalisés sur le même *cluster* de calcul, nous avons manqué de mémoire engendrée par les réorganisations/réécritures successives.

7.2.1 Création de l'index

Nous avons comparé *RedOak* à deux autres méthodes, à savoir une première méthode basée sur *Jellyfish* [Marcais and Kingsford, 2011] (*cf.* sous-section 2.2.1), ainsi qu'avec les *Bloom Filter Trie* (`BFT`) [Holley et al., 2016] (*cf.* sous-section 3.2.2). Notre comparaison a porté sur le temps et la mémoire vive nécessaire pour l'indexation.

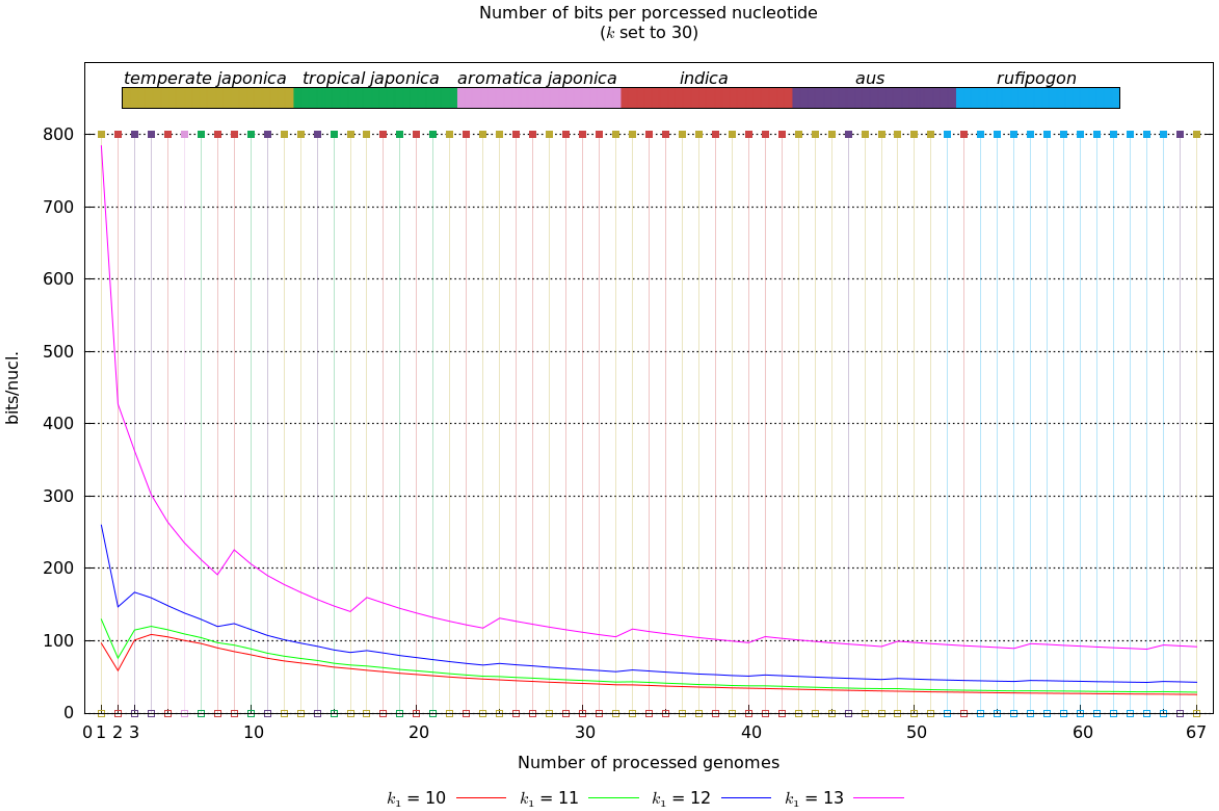


Figure 7.8: Étude du nombre de bits par nucléotide pour différentes tailles de préfixe allant de 10 à 13 en fonction des génomes. Pour cette figure, k=30 et les couleurs correspondent à la phylogénie faite dans l'article des 67 génomes de riz [Zhao et al., 2018].

La comparaison a été effectuée sur les 67 génomes de riz en comparant le temps utilisé pour la phase de construction de l'index et la consommation de mémoire maximale. La taille de l'ensemble de données a été définie successivement sur 10, 20, 30, 40, 50, 60 et 67 génomes de l'ensemble de données d'origine.

Jellyfish construit un index pour chaque génome, puis ces index ont été fusionnés pour produire une matrice dans laquelle les comptages de chaque k-mer de chaque génome sont stockés (petite modification de l'implémentation de l'outil de fusion de *Jellyfish*).

`BFT` a besoin des fichiers de comptage de k-mers au format texte (ASCII) pour construire son index. Il est possible d'utiliser les fichiers produits par des outils comme *Jellyfish*, `KMC`, ... (*cf.* sous-section 2.2.1). Pour *Jellyfish* et `BFT`, les valeurs indiquées sont le temps total pris pour les étapes de comptage et de fusion. Pour toutes les expériences, la taille de k-mer a été définie sur $k = 27$, car `BFT` nécessite que k soit multiple de 9.
Pour *RedOak*, la longueur du préfixe a été définie sur 12 (paramètre par défaut), ce qui donne un tableau de préfixes de taille totale très raisonnable (*i.e.*, $4^{12} = 128$ MiB). Chaque index de préfixe de chaque instance en cours d'exécution représente $3,2$ MiB soit 32 MiB par nœud. Cela réduit considérablement le risque de saturation au cours des expériences.

Pour chaque sous-ensemble, nous avons utilisé *RedOak* en parallèle sur 10 nœuds « normaux » du *cluster* et nous avons réservé 4 cœurs sur chaque nœud. Pour chaque sous-ensemble, nous avons également utilisé *Jellyfish* avec une taille de k-mers de 27, une table de hachage de 500 millions d'éléments sur 10 fil d'exécutions (« bigmem » options `-m 27 -s 500 M -t 10`) sur 40 génomes en parallèle à l'aide de 40 nœuds.

`BFT` ne permet pas la fusion des index créés et ne propose pas de parallélisation.

Par conséquent, nous avons exécuté chaque instance de BFT en parallèle en utilisant un nœud « bigmem » pour chaque sous-ensemble.

Les résultats sont résumés dans la table 7.1 et représenté graphiquement dans la figure 7.9. BFT n'a pas été en mesure d'indexer les jeux de données de 40 génomes et plus. Globalement, *RedOak* a montré de meilleures performances que *Jellyfish*. *RedOak* a utilisé 2 GiB par instance et, comme il est parallélisé sur 40 instances, il a utilisé 80 GiB pour tous les sous-ensembles et pour les 67 génomes assemblés. Le temps de construction de l'index en secondes était constant, à environ 1 470s (env. 25 minutes) pour dix génomes, et prenait un temps total de 8 100s (env. 2h15) pour les 67 génomes. Ces résultats ont été obtenus avec la version 0.0.0 de *RedOak*, c'est-à-dire avant optimisation de la mémoire. Nous n'avons malheureusement pas pu renouveler les expériences avec la dernière version de *RedOak*, les deux *clusters* utilisés ayant été inaccessibles en raison de problèmes essentiellement techniques qui n'ont pu être résolus au moment de la rédaction de ce manuscrit.

	Memoire RAM (GiB)			Temps (sec)		
$\mathcal{G}$	*RedOak*	*Jellyfish*	BFT	*RedOak*	*Jellyfish*	BFT
10	$4 \times 10 \times 2$	$10 \times 10,8$	42	1467,8	6617	748371,2
20	$4 \times 10 \times 2$	$20 \times 10,8$	65	2657,4	6638	854223,3
30	$4 \times 10 \times 2$	$30 \times 10,8$	91	3865,2	6637	1023657,2
40	$4 \times 10 \times 2$	$40 \times 10,8$	N/A	4952,3	6617	N/A
50	$4 \times 10 \times 2$	$40 \times 10,8$	N/A	6281,0	7074	N/A
60	$4 \times 10 \times 2$	$40 \times 10,8$	N/A	7609,6	6638	N/A
67	$4 \times 10 \times 2$	$40 \times 10,8$	N/A	8092,8	8591	N/A

Table 7.1: Comparaison des performances entre *RedOak* v0.0.0, *Jellyfish* v2.2.3 et BFT pour l'étape de création de l'index. La taille de l'entrée a été successivement définie sur 10, 20, 30, 40, 50, 60 et 67 génomes assemblés. L'utilisation de la RAM est exprimée en GiB. Les temps indiqués sont les temps d'exécution de l'horloge « wall-clock » en secondes qui correspond au temps utilisateur.

La figure 7.9 reprend ces résultats et illustre l'évolution du temps en fonction du nombre de génomes que l'on cherche à indexer pour chaque outil.

7.2.2 Les requêtes

Nous avons évalué la performance de *RedOak* pour l'interrogation, avec des séquences de différentes longueurs, de l'index des 67 génomes de riz assemblés.

Nous avons comparé la version v0.0.0 de *RedOak* et la dernière version de *Jellyfish* en utilisant des séquences de requêtes aléatoires dont la longueur varie de 10 fois la taille de k à 1 000 fois celle de k.
Les résultats sont présentés dans la table 7.2, ils montrent l'utilisation maximale de la RAM et le temps d'exécution en temps d'horloge (temps utilisateur) requis pour rechercher les k-mers d'une séquence créée de manière aléatoire dans l'index des 67 génomes de riz assemblés.

Pour évaluer le temps de requête de *Jellyfish*, nous avons dû interroger chaque fichier individuellement. La table 7.2 indique, pour *Jellyfish*, le maximum de mémoire RAM (y compris le temps d'écriture de tous les k-mers), le temps de la requête la plus longue et le temps total (somme de tous les temps qui donne le temps moyen : $4403,7$ sec par fichier).
Les résultats ont montré que *RedOak* offre de meilleures performances que *Jellyfish* pour l'interrogation de cet ensemble de données.

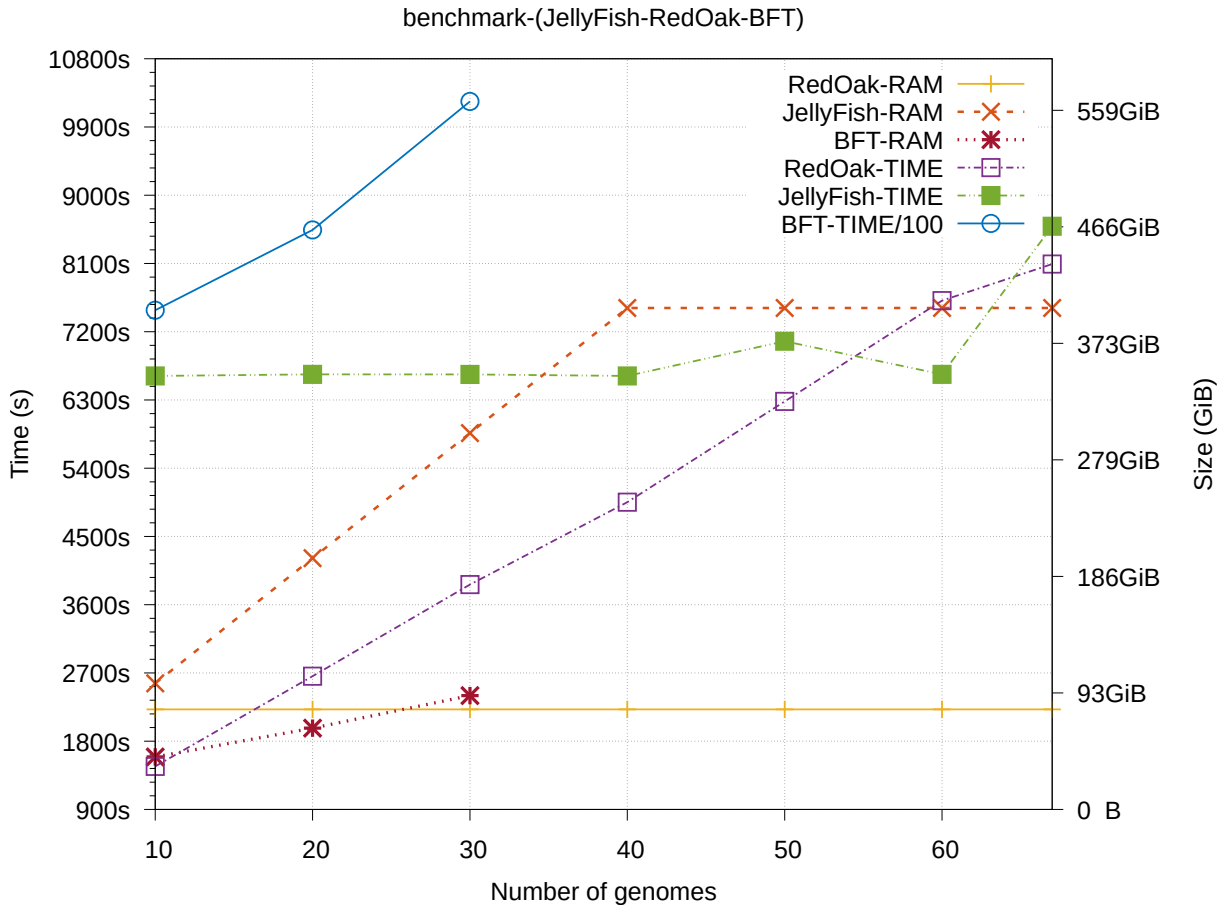


Figure 7.9: Comparaison des performances entre *RedOak*, *Jellyfish* et `BFT` pour l'étape de construction de l'index.

7.3 Résultats biologiques

L'analyse de la variation présence/absence (PAV) de gènes entre les différents génomes est un résultat classique des approches pan-génomiques [Computational Pan-Genomics, 2016, Hu et al., 2018, Zhao et al., 2018].
RedOak possède une fonction d'interrogation de séquences nucléotidiques (y compris de leurs complémentaires inversés) qui peut être utilisée pour analyser rapidement la présence/absence des k-mers de séquences requêtes parmi une vaste collection de génomes.
Par analogie, nous parlerons également de PAV dans le cas des requêtes de *RedOak*. En effet, nous pouvons interroger l'index généré par *RedOak*, en utilisant tous les k-mers contenus dans une séquence de gène donnée, l'index généré avec *RedOak* de l'ensemble des génomes et obtenir une information similaire.

7.3.1 Lecture et interprétation des résultats de requêtes de *RedOak*

Pour chaque génome G, trois scores simples sont calculés : s_G^+, s_G^- et s_G^*, initialisés à 0. Si le k-mer est présent dans le génome, le score s_G^+ est incrémenté. Si le complémentaire inverse du k-mer est présent dans le génome, le score s_G^- est incrémenté et si le k-mer et son complémentaire inverse sont présents dans le génomes, alors le score s_G^* est également incrémenté.
Les scores sont ensuite normalisés sur le nombre de k-mers de la requête, ce qui donne

	Query Time (s)		
Taille de la requête	*RedOak* Query	*Jellyfish* Query	*Jellyfish* Query-Total
270	1	6360	257926
540	3	7919	301113
810	3	11586	374489
1080	7	12996	368945
1350	4	12280	351517
2700	9	12880	391060
5400	20	13397	337673
8100	25	11701	349188
10800	35	10668	262252
13500	44	9779	263889
27000	60	9569	295048

Table 7.2: Comparaison des performances entre *RedOak* et *Jellyfish* pour l'interrogation avec des séquences simulées de différentes longueurs (de 10 000 à 1 000 000) d'un index de 67 génomes assemblés. Les temps indiqués sont les temps d'exécution, correspond au temps utilisateur sont exprimés en secondes.

respectivement les pourcentages de k-mers présents dans le sens de lecture de la requête (le sens $5' - 3'$), dans la séquence complémentaire inversée de la requête (le sens $3' - 5'$) et dans les deux sens de lecture.

Graphiquement, pour une séquence requête q donnée de longueur $|q| = \ell$, *RedOak* va fournir pour chaque génome G une séquence q_G de même longueur composée sur l'alphabet $\{+, -, *, :, .\}$ en utilisant la convention suivante :

- si le k-mer à la position $1 \leq i \leq \ell - k + 1$ de la requête ($q[i..i + k - 1]$) est présent dans le génome G, mais que son complémentaire inverse n'est pas présent alors $q_G[i]$ sera égal à '+' ;
- si le k-mer à la position $1 \leq i \leq \ell - k + 1$ de la requête ($q[i..i + k - 1]$) ainsi que son complémentaire inverse sont présents dans le génome G, alors $q_G[i]$ sera égal à '*' ;
- si le k-mer à la position $1 \leq i \leq \ell - k + 1$ de la requête ($q[i..i + k - 1]$) n'est pas présent dans le génome G, mais que son complémentaire inverse l'est alors $q_G[i]$ sera égal à '-' ;
- si ni le k-mer à la position $1 \leq i \leq \ell - k + 1$ de la requête ($q[i..i + k - 1]$), ni son complémentaire inverse ne sont présents, ou si $\ell - k + 1 < i \leq \ell$, mais qu'il existe un k-mer à la position $\max(i - k + 1, 1) \leq j < i$ qui est présent ou dont le complémentaire inverse est présent, alors $q_G[i]$ sera égal à ':' ;
- dans le dernier cas de figure, alors $q_G[i]$ sera égal à '.'.

Cette convention permet de visualiser la composition en k-mers de la requête pour chaque génome. Bien évidemment, un score s_G^+ (resp. s_G^-) de 100% ne signifie pas que la séquence requête est présente dans le génome G avec certitude, mais peut tout de même largement conforter cette hypothèse. En effet, le score s_G^+ (resp. s_G^-) ne représente que le maximum potentiel d'identité entre la requête (resp. son complémentaire inverse) et n'importe quelle séquence de longueur ℓ du génome G.

Nous allons utiliser un exemple simple d'indexation et de requête de génomes constitué d'une seule séquence de 38 caractères. Pour faciliter la lecture des séquences, les caractères représentant les paires de bases GC donc avec 3 liaisons d'hydrogène seront mis en évidence par des lettres en majuscules et les caractères représentant les paires de bases AT donc avec 2 liaisons d'hydrogène seront mis en évidence par des lettres en minuscule.

Fichier 7.1: Fichier `FichierSequence.fasta` contenant une seule séquence aléatoire de 38 caractères.

```
1 >Sequence
2 GGGGCCCCaaaaCCaaaatttatatataaaaaGGaaaa
```

Fichier 7.2: Fichier `FichierSequenceReverse.fasta` contenant la séquence inversée de la séquence du fichier 7.1 de 38 caractères.

```
1 >Sequence Reverse
2 aaaaGGaaaaatatatatttaaaaCCaaaaCCCCGGGG
```

Fichier 7.3: Fichier `FichierSequenceComplement.fasta` contenant le complément de la séquence du fichier 7.1 de 38 caractères.

```
1 >Sequence Complement
2 CCCCGGGGttttGGttttaaatatatattttCCtttt
```

Fichier 7.4: Fichier `FichierSequenceReverseComplement.fasta` contenant le complémentaire inversé de la séquence du fichier 7.1 de 38 caractères.

```
1 >Sequence ReverseComplement
2 ttttCCttttttatatataaattttGGttttGGGGCCCC
```

Dans un premier temps, *RedOak* sera utilisé pour indexer les 5-mers des 4 fichiers ci-avant (fichiers 7.1 à 7.4). Pour chacun des fichiers, nous les faisons précéder d'un label qui sera affiché en lieu et place du nom du fichier (option `--genome`). Ici, chaque fichier au format `fasta` ne contient qu'une seule séquence, mais bien évidement il est possible d'avoir plusieurs séquences par fichier et d'utiliser plusieurs fichiers par génome.
La séquence requête présentée à *RedOak* sera la séquence exacte du fichier 7.1 (`FichierSequence.fasta`) : `GGGGCCCCaaaaCCaaaatttatatataaaaaGGaaaa`. Ces deux étapes sont réalisées par la ligne de commande ci-dessous, qui exécutera *RedOak* sur deux instances (option `-n 2` de la commande `mpirun`).

```
prompt$ mpirun -n 2 toto/bin/redoak \
> --kmer 5 \
> --genome Sequence:FichierSequence.fasta \
> --genome Sequence\ inversee:FichierSequenceReverse.fasta \
> --genome Sequence\ complementaire:FichierSequenceComplement.fasta \
> --genome Sequence\ complementaire\ inversee:FichierSequenceReverseComplement.
fasta \
> --query Requete:GGGGCCCCaaaaCCaaaatttatatataaaaaGGaaaa
```

L'exécution de cette ligne de commande commence par indexer les fichiers comme illustré ci-après.

```
Total Elapsed Time [Sequence] [======================================] 0.00s
Merging k-mers from Sequence complementaire inversee with the indexed k-mers (this
 may take a while)... [DONE]
Total Elapsed Time [Sequence in...] [================================] 0.00s
Merging k-mers from Sequence complementaire inversee with the indexed k-mers (this
 may take a while)... [DONE]
Total Elapsed Time [Sequence co...] [================================] 0.00s
Merging k-mers from Sequence complementaire inversee with the indexed k-mers (this
 may take a while)... [DONE]
Total Elapsed Time [Sequence co...] [================================] 0.00s
Merging k-mers from Sequence complementaire inversee with the indexed k-mers (this
 may take a while)... [DONE]
```

Ensuite, la requête est effectuée. Tout d'abord, un rappel de la syntaxe utilisée est fourni en reprenant la syntaxe des commentaires *shell*.

```
#The different available values for the query positions are:
#  '+': the kmer starting at this position is shared by the genome
#  '-': the reverse complement of the kmer starting at this position is shared by
the genome
#  '*': both the kmer starting at this position and its reverse complement are
shared by the genome
#  ':': the kmer and its reverse complement are not found in the genome, but there
 is a preceeding k-mer that overlaps this position
#  '.': the kmer and its reverse complement are not found in the genome
#
```

Ensuite la séquence requête est rappelée dans un format similaire au format `fasta` (où l'entête de la séquence est le label fourni avant les « : » de l'argument de l'option `--query` suivi de sa taille), puis les séquences de résultats pour chaque « génome » indexé sont affichés.

```
>Requete [38bp.]
GGGGCCCCaaaaCCaaaatttatatataaaaaGGaaaa
>Sequence [20+, 0-, 14*, 34 5-mers / 38+, 23-, 38 nucl.]
**********************************::::
>Sequence inversee [18+, 0-, 4*, 34 5-mers / 32+, 8-, 38 nucl.]
......+++++++++::+::****::++++++++::::
>Sequence complementaire [0+, 20-, 14*, 34 5-mers / 23+, 38-, 38 nucl.]
****----------**-********--------::::
>Sequence complementaire inversee [0+, 18-, 4*, 34 5-mers / 8+, 32-, 38 nucl.]
..................::-::****::---------::::
```

Nous observons clairement que tous les 5-mers de la requête sont parfaitement retrouvés dans le 1er génome indexé (`Sequence`). En détaillant la sortie, nous avons 20 5-mers qui ont été retrouvés dans le sens de lecture uniquement, aucun 5-mers dont uniquement le complémentaire inversé n'a été retrouvé et 14 5-mers qui ont été retrouvés et dont le complémentaire inversé a également été retrouvé, ce qui signifie que sur les 34 5-mers recherchés, tous ont été retrouvés dans le sens de la lecture, couvrant donc bien les 38 nucléotides de la requête (marqué `38+`).

En observant la réponse obtenue pour le 3ème génome (`Sequence complementaire`), nous n'avons aucun 5-mer qui ne soit retrouvé que dans le sens de la lecture. Il y a 20 5-mers dont les complémentaires inversés ont été retrouvés et 14 5-mers qui ont été retrouvés à la fois dans le sens de la lecture et dont les complémentaires inversés ont également été retrouvés. Ce qui donne le résultat cohérent que sur les 38 nucléotides, tous sont couverts par le complémentaire inversé des 5-mers indexés pour ce génome.

De manière idoine, sur la deuxième séquence (`Sequence inversee`) et la quatrième séquence (`Sequence complementaire inversee`), nous ne retrouvons que certains 5-mers. Ces 5-mers correspondent à la répétition en tandem des motifs `CCaaaaCCaaaa` et `tatatata` de la requête, ce dernier étant identique à son complémentaire inversé.

Enfin, un résumé des informations liées au calcul de l'index et au traitement des requêtes est affiché. Nous y retrouvons le nombre d'instances de *RedOak* exécutées (deux), les informations sur les tailles de k-mers et du préfixe utilisé, le nombre de génomes indexés, le nombre de nucléotides lus ainsi que le nombre de 5-mers distincts indexés, leur répartition en *core*, *shell* (variables) et *cloud* (spécifiques). D'autres informations comme des statistiques et des temps d'utilisation sont disponibles.

```
+-------------------------------------------------------------------+
|                          Informations                             |
+-----------------------------------+-------------------------------+
| Number of running instances       |                             2 |
| - spartacus                       |                             2 |
+-----------------------------------+-------------------------------+
| k-mer size                        |                             5 |
| Prefix size                       |                             4 |
| Number of indexed genomes         |                             4 |
| Number of processed nucleotides   |                           152 |
| Number of indexed k-mers          |                            62 |
| - Number of core k-mers           |                             2 |
| - Number of variable k-mers       |                            52 |
| - Number of specific k-mers       |                             8 |
| Average number of bits per k-mer  |                 3.15894e+06 |
| Average number of bits per nucl.  |                 1.28852e+06 |
+-----------------------------------+-------------------------------+
| Wallclock time                    |                  0.018 sec. |
| - Index construction (/w shell)   |                  0.008 sec. |
| - Running queries (/w shell)      |                  0.000 sec. |
+-------------------+---------------+---------------+---------------+
| By instance       |    Minimum    |    Average    |    Maximum    |
+-------------------+---------------+---------------+---------------+
| CPU time (user)   |    0.016 sec. |    0.016 sec. |    0.016 sec. |
| CPU time (system) |    0.003 sec. |    0.004 sec. |    0.006 sec. |
| Memory            |       11 MB   |       11 MB   |       11 MB   |
+-------------------+---------------+---------------+---------------+
```

7.3.2 Recherche des séquences des gènes de Nipponbare

À titre d'exemple, nous avons indexé les 67 génomes de riz de [Zhao et al., 2018] avec *RedOak* et nous avons recherché toutes les séquences des gènes du génome de Nipponbare.

Cas particulier du gène PSTOL

Notre attention a été particulièrement attirée par le gène PSTOL, gène qui contrôle la tolérance à la carence en phosphore [Gamuyao et al., 2012]. En effet, nous avons pu détecter la présence d'une séquence très fortement similaire au gène PSTOL dans le génome GP104 (6 substitutions sur une séquence requête de 1017pb), alors qu'il est dit absent dans GP104, W1739, W1754, W1777 dans la publication de [Zhao et al., 2018]. Étant donné que c'était la seule différence entre notre outil et la publication de [Zhao et al., 2018], nous nous sommes questionnés sur le fait que ce gène soit réellement présent ou non dans l'accession GP104.

Pour valider notre questionnement nous avons travaillé par rapport à la séquence PSTOL qui correspond au gène Kasalath_00009922_00007544. La séquence trouvée sur NCBI est légèrement différente (*cf.* figure 7.10) aussi l'avons nous intégrée dans notre étude comparative.

Dans les quatre génomes (GP104, W1739, W1754, W1777), nous avons recherché l'existence de séquences protéiques proches de Kasalath_00009922_00007544. Nous avons trouvé exactement les mêmes dans les protéomes des 4 variétés étudiées, qui correspondent parfaitement à la partie N-terminale de PSTOL mais avec un codon stop prématuré (*cf.* figure 7.11).

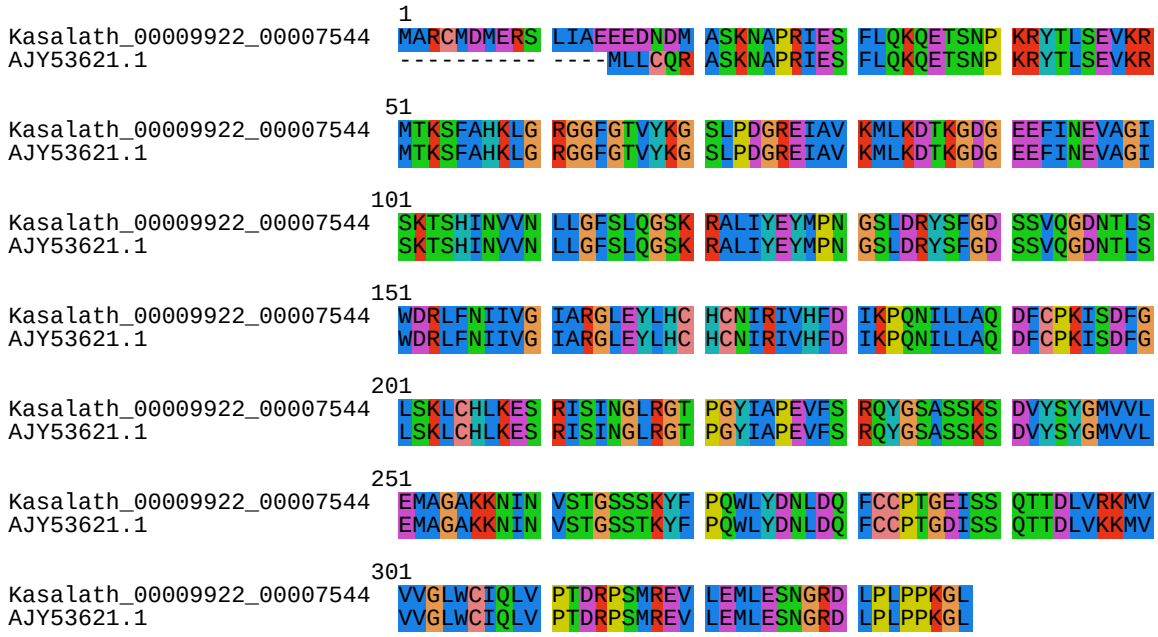


Figure 7.10: Alignement de la séquence `Kasalath_00009922_00007544` codant pour le gène PSTOL avec la séquence `AJY53621.1` correspondante trouvée sur le NCBI.

Nous avons donc récupéré la séquence nucléotidique qui correspond à la séquence `GP104_00004118_000322` et nous l'avons alignée avec la séquence obtenue avec *RedOak*. Il apparaît clairement que la séquence identifiée par *RedOak* correspond bien aux deux exons de la séquence `GP104_00004118_00032251` (*cf.* figure 7.12).

L'alignement de la séquence de `GP104` identifiée par *RedOak* avec la séquence du gène PSTOL du génome de `Kasalath` est quant à lui quasi-parfait. Il n'y a que 6 substitutions entre les deux séquences (aux positions 156, 452, 474, 566, 637 et 810). Toutefois la seconde substitution entraîne l'apparition d'un codon stop prématuré. Ceci explique probablement pourquoi ce gène n'a pas été référencé dans [Zhao et al., 2018] pour le génome `GP104`. Il ne s'agit donc en aucun cas d'un artefact lié à l'utilisation des k-mers par *RedOak*, mais bel et bien d'un phénomène biologique. La séquence recherchée avec *RedOak* apparaît donc bien dans le génome `GP104`, à 6 substitutions près.

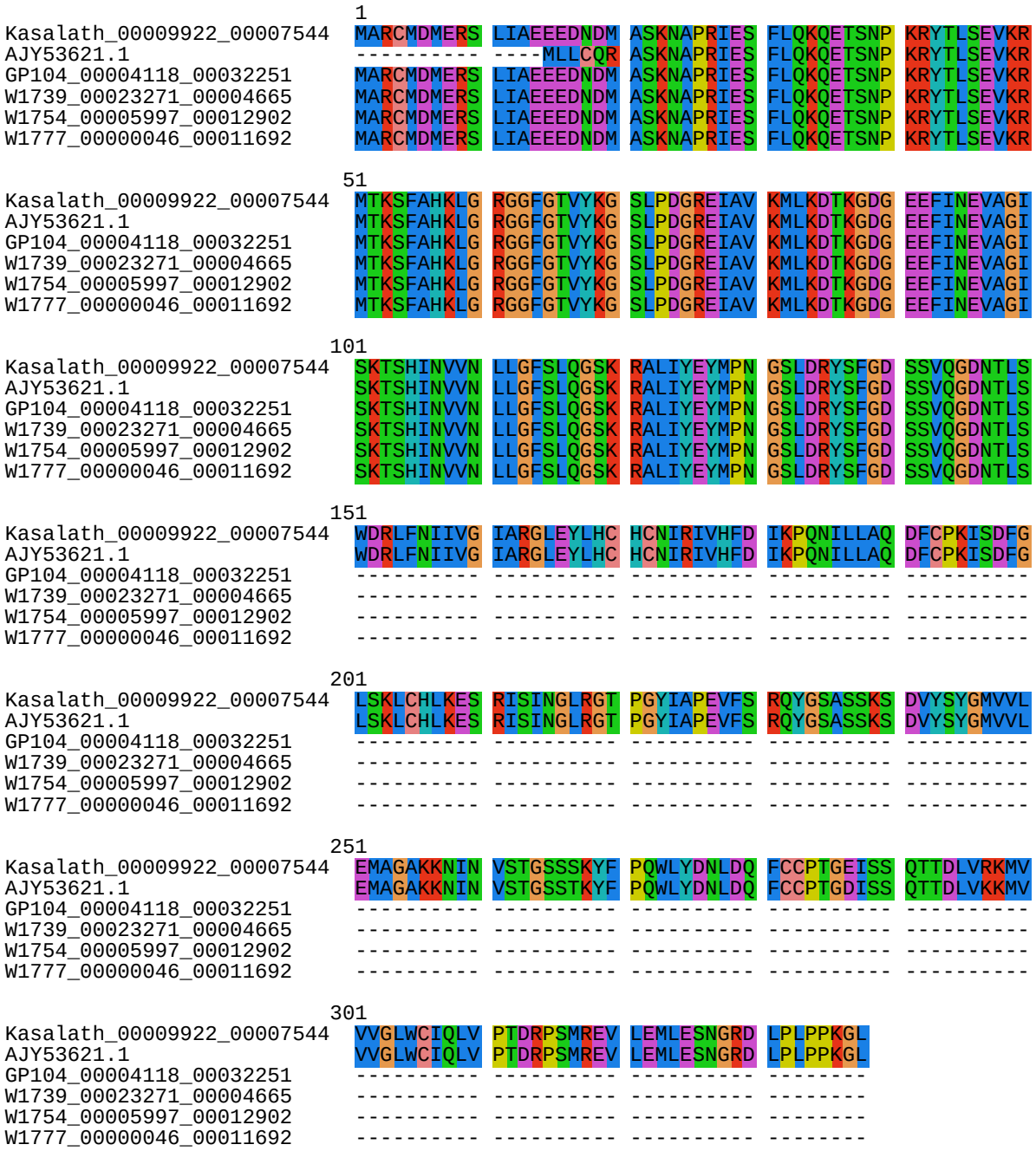


Figure 7.11: Alignement du gène `Kasalath_00009922_00007544` avec des protéines issues des génomes `GP104`, `W1739`, `W1754`, `W1777`, ainsi qu'avec la séquence en provenance du NCBI (`AJY53621.1`).

```
>GP104_00004118_00032251   5   2 exon (s)  32251  -  34150   453 bp, chain +
ATGGCCAGATGCATGGACATGGAAAGGTCGCTTATCGCGGAAGAAGAGGATAACGATATG
GCATCAAAGAATGCACCAAGGATAGAATCTTTCCTACAAAAGCAAGAAACTTCAAACCCA
AAAAGATACACTCTCTCTGAAGTGAAAAGAATGACGAAATCTTTTGCTCACAAGCTTGGC
AGAGGTGGCTTTGGTACTGTTTATAAAGGTAGCCTGCCTGATGGCCGTGAGATAGCCGTC
AAGATGCTAAAGGATACCAAGGGTGATGGGGAGGAATTCATAAATGAGGTTGCTGGCATT
AGTAAAACTTCTCATATCAATGTTGTTAACCTTCTAGGTTTTTCCCTTCAAGGGTCAAAA
AGAGCTCTGATCTATGAGTACATGCCCAATGGTTCACTTGATAGATATTCTTTTGGCGAT
AGCTCTGTCCAAGGAGATAACACCCTGAGCTAG

ALIGNEMENT PAR BLASTN 2 SEQUENCES:
Query = GP104_00004118_00032251
Sbjct = GP104.fa

Score Expect Identities Gaps Strand
111 bits(60)  1e-27  60/60(100%)  0/60(0%)  Plus/Plus

Query  1    ATGGCCAGATGCATGGACATGGAAAGGTCGCTTATCGCGGAAGAAGAGGATAACGATATG  60
            ||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||
Sbjct  1    ATGGCCAGATGCATGGACATGGAAAGGTCGCTTATCGCGGAAGAAGAGGATAACGATATG  60

Score Expect Identities Gaps Strand
728 bits(394)  0.0  394/394(100%)  0/394(0%)  Plus/Plus

Query  60    GGCATCAAAGAATGCACCAAGGATAGAATCTTTCCTACAAAAGCAAGAAACTTCAAACCC  119
             ||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||
Sbjct  1507  GGCATCAAAGAATGCACCAAGGATAGAATCTTTCCTACAAAAGCAAGAAACTTCAAACCC  1566

Query  120   AAAAAGATACACTCTCTCTGAAGTGAAAAGAATGACGAAATCTTTTGCTCACAAGCTTGG  179
             ||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||
Sbjct  1567  AAAAAGATACACTCTCTCTGAAGTGAAAAGAATGACGAAATCTTTTGCTCACAAGCTTGG  1626

Query  180   CAGAGGTGGCTTTGGTACTGTTTATAAAGGTAGCCTGCCTGATGGCCGTGAGATAGCCGT  239
             ||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||
Sbjct  1627  CAGAGGTGGCTTTGGTACTGTTTATAAAGGTAGCCTGCCTGATGGCCGTGAGATAGCCGT  1686

Query  240   CAAGATGCTAAAGGATACCAAGGGTGATGGGGAGGAATTCATAAATGAGGTTGCTGGCAT  299
             ||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||
Sbjct  1687  CAAGATGCTAAAGGATACCAAGGGTGATGGGGAGGAATTCATAAATGAGGTTGCTGGCAT  1746

Query  300   TAGTAAAACTTCTCATATCAATGTTGTTAACCTTCTAGGTTTTTCCCTTCAAGGGTCAAA  359
             ||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||
Sbjct  1747  TAGTAAAACTTCTCATATCAATGTTGTTAACCTTCTAGGTTTTTCCCTTCAAGGGTCAAA  1806

Query  360   AAGAGCTCTGATCTATGAGTACATGCCCAATGGTTCACTTGATAGATATTCTTTTGGCGA  419
             ||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||
Sbjct  1807  AAGAGCTCTGATCTATGAGTACATGCCCAATGGTTCACTTGATAGATATTCTTTTGGCGA  1866

Query  420   TAGCTCTGTCCAAGGAGATAACACCCTGAGCTAG  453
             ||||||||||||||||||||||||||||||||||
Sbjct  1867  TAGCTCTGTCCAAGGAGATAACACCCTGAGCTAG  1900
```

Figure 7.12: Alignement par blastn de deux séquences, de la séquence de GP104-00004118-00032251 identifiée par *RedOak* avec la séquence du gène PSTOL du génome de GP104.

```
>Kasalath_00009922_00007544   1   2 exon (s)   7544  -  10008  1017 bp, chain -
ATGGCCAGATGCATGGACATGGAAAGGTCGCTTATCGCGGAAGAAGAGGATAACGATATG
GCATCAAAGAATGCACCAAGGATAGAATCTTTCCTACAAAAGCAAGAAACTTCAAACCCA
AAAAGATACACTCTCTCTGAAGTGAAAAGAATGACTAAATCTTTTGCTCACAAGCTTGGC
AGAGGTGGCTTTGGTACTGTTTATAAAGGTAGCCTGCCTGATGGCCGTGAGATAGCCGTC
AAGATGCTAAAGGATACCAAGGGTGATGGGGAGGAATTCATAAATGAGGTTGCTGGCATT
AGTAAAACTTCTCATATCAATGTTGTTAACCTTCTAGGTTTTTCCCTTCAAGGGTCAAAA
AGAGCTCTGATCTATGAGTACATGCCCAATGGTTCACTTGATAGATATTCTTTTGGCGAT
AGCTCTGTCCAAGGAGATAACACCCTGAGCTGGGATAGACTGTTCAATATTATTGTCGGG
ATTGCTCGAGGGCTGGAGTATCTCCACTGTCATTGCAACATTCGCATTGTGCATTTTGAT
ATCAAACCTCAAAACATTCTACTGGCTCAAGATTTCTGTCCAAAGATCTCTGATTTTGGC
CTGTCAAAATTGTGCCATCTAAAGGAGAGCAGAATTTCGATCAACGGACTAAGAGGAACA
CCTGGCTACATTGCACCTGAAGTGTTTTCCAGGCAGTATGGATCTGCCAGCAGCAAATCT
GATGTCTACAGCTATGGAATGGTGGTCCTTGAGATGGCTGGTGCAAAGAAAAACATCAAC
GTTAGTACAGGTAGTAGCAGCAAATATTTTCCCCAATGGTTATACGATAATTTGGACCAG
TTTTGTTGCCCCACGGGCGAGATTAGTAGCCAGACCACCGATCTTGTAAGGAAGATGGTT
GTCGTTGGTTTGTGGTGCATACAACTCGTACCTACAGATCGACCGTCCATGAGAGAAGTC
CTTGAGATGTTGGAAAGCAACGGTAGGGACTTACCGTTGCCACCAAAAGGGCTTTGA

ALIGNEMENT PAR BLASTN 2 SEQUENCES:
Query = Kasalath_00009922_00007544
Sbjct = GP104.fa
REMARQUE : Le codon stop est en position 1899 dans la sequence Sbjct, et en 452
dans la Query

Score Expect Identities Gaps Strand
111 bits(60)  3e-27  60/60(100%)  0/60(0%)  Plus/Plus

Query  1    ATGGCCAGATGCATGGACATGGAAAGGTCGCTTATCGCGGAAGAAGAGGATAACGATATG  60
            ||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||
Sbjct  1    ATGGCCAGATGCATGGACATGGAAAGGTCGCTTATCGCGGAAGAAGAGGATAACGATATG  60

Score Expect Identities Gaps Strand
1736 bits(940)  0.0  952/958(99%)  0/958(0%)  Plus/Plus

Query  60    GGCATCAAAGAATGCACCAAGGATAGAATCTTTCCTACAAAAGCAAGAAACTTCAAACCC  119
             ||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||
Sbjct  1507  GGCATCAAAGAATGCACCAAGGATAGAATCTTTCCTACAAAAGCAAGAAACTTCAAACCC  1566

Query  120   AAAAAGATACACTCTCTCTGAAGTGAAAAGAATGACTAAATCTTTTGCTCACAAGCTTGG  179
             |||||||||||||||||||||||||||||||||| ||||||||||||||||||||||||
Sbjct  1567  AAAAAGATACACTCTCTCTGAAGTGAAAAGAATGACGAAATCTTTTGCTCACAAGCTTGG  1626

Query  180   CAGAGGTGGCTTTGGTACTGTTTATAAAGGTAGCCTGCCTGATGGCCGTGAGATAGCCGT  239
             ||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||
Sbjct  1627  CAGAGGTGGCTTTGGTACTGTTTATAAAGGTAGCCTGCCTGATGGCCGTGAGATAGCCGT  1686

Query  240   CAAGATGCTAAAGGATACCAAGGGTGATGGGGAGGAATTCATAAATGAGGTTGCTGGCAT  299
             ||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||
Sbjct  1687  CAAGATGCTAAAGGATACCAAGGGTGATGGGGAGGAATTCATAAATGAGGTTGCTGGCAT  1746

Query  300   TAGTAAAACTTCTCATATCAATGTTGTTAACCTTCTAGGTTTTTCCCTTCAAGGGTCAAA  359
             ||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||
Sbjct  1747  TAGTAAAACTTCTCATATCAATGTTGTTAACCTTCTAGGTTTTTCCCTTCAAGGGTCAAA  1806

Query  360   AAGAGCTCTGATCTATGAGTACATGCCCAATGGTTCACTTGATAGATATTCTTTTGGCGA  419
             ||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||
Sbjct  1807  AAGAGCTCTGATCTATGAGTACATGCCCAATGGTTCACTTGATAGATATTCTTTTGGCGA  1866

Query  420   TAGCTCTGTCCAAGGAGATAACACCCTGAGCTGGGATAGACTGTTCAATATTATTGTCGG  479
             |||||||||||||||||||||||||||||||| |||||||||||||||||||||| |||||
Sbjct  1867  TAGCTCTGTCCAAGGAGATAACACCCTGAGCTAGGATAGACTGTTCAATATTATCGTCGG  1926

Query  480   GATTGCTCGAGGGCTGGAGTATCTCCACTGTCATTGCAACATTCGCATTGTGCATTTTGA  539
             ||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||
Sbjct  1927  GATTGCTCGAGGGCTGGAGTATCTCCACTGTCATTGCAACATTCGCATTGTGCATTTTGA  1986
```

Figure 7.13: Alignement de la séquence de GP104 identifiée par *RedOak* avec la séquence du gène PSTOL du génome de Kasalath.

```
Query  540   TATCAAACCTCAAAACATTCTACTGGCTCAAGATTTCTGTCCAAAGATCTCTGATTTTGG  599
              |||||||||||||||||||||||||| ||||||||||||||||||||||||||||||||
Sbjct  1987  TATCAAACCTCAAAACATTCTACTGGATCAAGATTTCTGTCCAAAGATCTCTGATTTTGG  2046

Query  600   CCTGTCAAAATTGTGCCATCTAAAGGAGAGCAGAATTTCGATCAACGGACTAAGAGGAAC  659
              |||||||||||||||||||||||||||||||||||||||||||||| ||||||||||||
Sbjct  2047  CCTGTCAAAATTGTGCCATCTAAAGGAGAGCAGAATTTCGATCAACGCACTAAGAGGAAC  2106

Query  660   ACCTGGCTACATTGCACCTGAAGTGTTTTCCAGGCAGTATGGATCTGCCAGCAGCAAATC  719
              ||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||
Sbjct  2107  ACCTGGCTACATTGCACCTGAAGTGTTTTCCAGGCAGTATGGATCTGCCAGCAGCAAATC  2166

Query  720   TGATGTCTACAGCTATGGAATGGTGGTCCTTGAGATGGCTGGTGCAAAGAAAAACATCAA  779
              ||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||
Sbjct  2167  TGATGTCTACAGCTATGGAATGGTGGTCCTTGAGATGGCTGGTGCAAAGAAAAACATCAA  2226

Query  780   CGTTAGTACAGGTAGTAGCAGCAAATATTTTCCCCAATGGTTATACGATAATTTGGACCA  839
              |||||||||||||||||||||||||||| ||||||||||||||||||||||||||||||
Sbjct  2227  CGTTAGTACAGGTAGTAGCAGCAAATATTTCCCCCAATGGTTATACGATAATTTGGACCA  2286

Query  840   GTTTTGTTGCCCCACGGGCGAGATTAGTAGCCAGACCACCGATCTTGTAAGGAAGATGGT  899
              ||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||
Sbjct  2287  GTTTTGTTGCCCCACGGGCGAGATTAGTAGCCAGACCACCGATCTTGTAAGGAAGATGGT  2346

Query  900   TGTCGTTGGTTTGTGGTGCATACAACTCGTACCTACAGATCGACCGTCCATGAGAGAAGT  959
              ||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||
Sbjct  2347  TGTCGTTGGTTTGTGGTGCATACAACTCGTACCTACAGATCGACCGTCCATGAGAGAAGT  2406

Query  960   CCTTGAGATGTTGGAAAGCAACGGTAGGGACTTACCGTTGCCACCAAAAGGGCTTTGA  1017
              |||||||||||||||||||||||||||||||||||||||||||||||||||||||||||
Sbjct  2407  CCTTGAGATGTTGGAAAGCAACGGTAGGGACTTACCGTTGCCACCAAAAGGGCTTTGA  2464
```

Figure 7.13: Alignement de la séquence de `GP104` identifiée par *RedOak* avec la séquence du gène PSTOL du génome de `Kasalath`.

7.3.3 *k*-mers et la représentation des extrémités des exons.

Dans cette sous-partie nous nous sommes intéressés aux requêtes qu'il est possible de faire avec *RedOak*. Nous avons donc indexé avec *RedOak* les 67 génomes et recherché avec l'option `-query` la séquence du gène PSTOL. Nous nous intéressons à un phénomène que nous avons pu observer. Ce phénomène se produit lorsque nous utilisons une approche de découpage par *k*-mers ; il se caractérise par une région systématiquement manquante comme mis en avant sur la figure 7.14. Si nous exécutons la requête de la séquence du gène PSTOL sur les mêmes données sans que les introns aient été enlevés, les parties manquantes dans la figure 7.14 sont représentées dans la figure 7.15. Nous expliquons le fait que les régions correspondantes aux extrémités des exons ne sont pas représenté par une approche de découpage en *k*-mers à l'aide du schéma présenté à la figure 7.16. Tous les *k*-mers présents dans la partie génomique qui ont au moins un préfixe appartenant à l'exon1 ET un suffixe appartenant à l'intron1 alors ils ne seront pas retrouvés dans la partie CDS et vont former un trou (se référer à la figure 7.14). Nous avons également essayé de retrouver le gène ERECTA dans notre index, il aurait été éventuellement possible que certains *k*-mers soit partagés entre le gène et un des génomes de riz. Le résultat obtenu est qu'aucun *k*-mer de taille 27 nucléotides n'est retrouvé (voir figure 7.17).

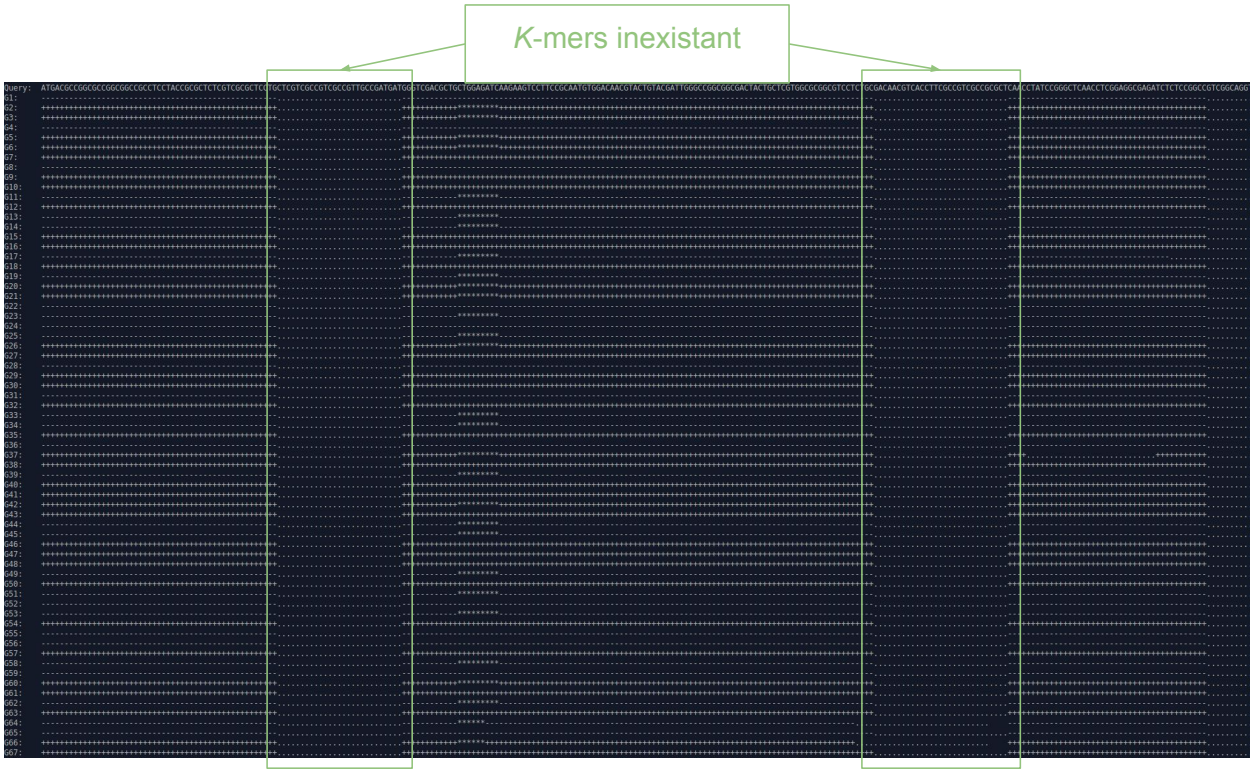


Figure 7.14: Résultat obtenu avec *RedOak* de la recherche du CDS (Coding DNA Sequence) : `">LOC_Os06g10230.1"` qui est une séquence codante dans les 67 génomes de riz proposé par [Zhao et al., 2018].

Synthèse

Dans cette section 7.3 nous avons présenté les résultats de requêtes obtenu avec *RedOak*. Dans un premier temps nous avons créé de toute pièce des fichiers de type fasta contenant qu'une seule séquence puis nous avons utilisé ces fichiers pour illustrer l'utilisation des requêtes de *RedOak*. Nous nous sommes ensuite intéressé à un cas concret, celui du gène PSTOL. Nous avons validé la présence du gène PSTOL à l'aide d'un alignement fait avec le logiciel BLASTN.

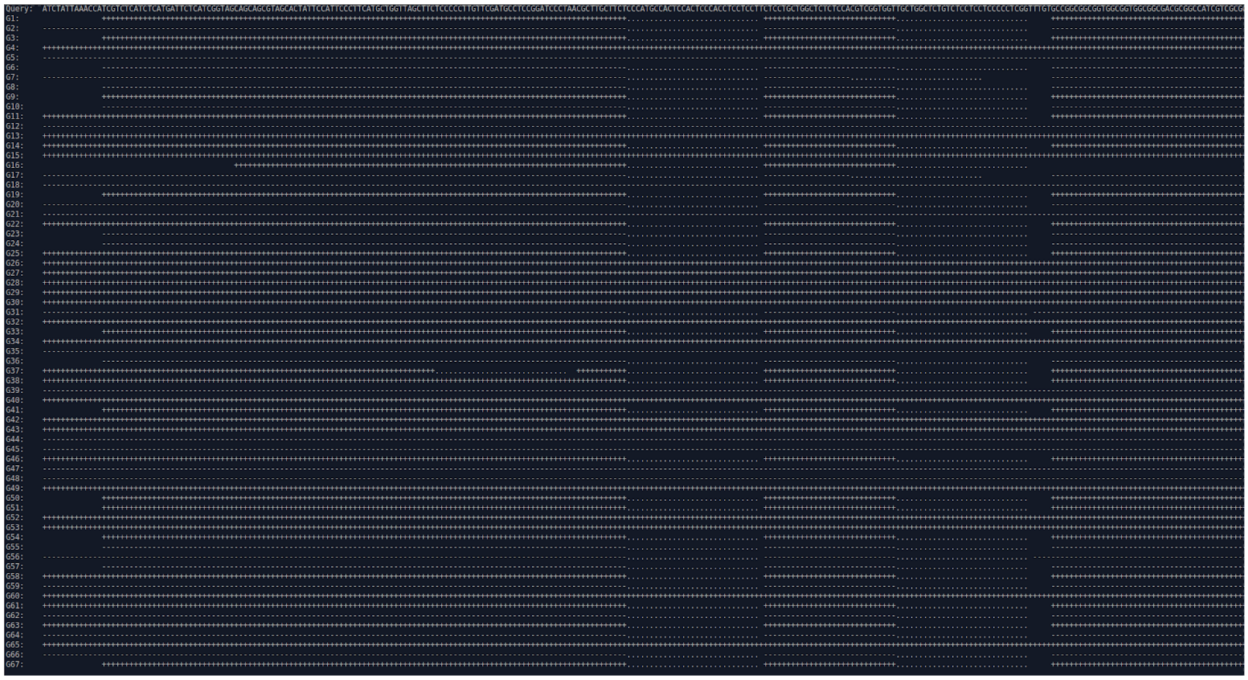


Figure 7.15: Résultat obtenu avec *RedOak* de la recherche de la séquence `">LOC_Os06g35200"` qui est une séquence génomique ne contenant qu'un seul intron dans les 67 génomes de riz proposé par [Zhao et al., 2018].

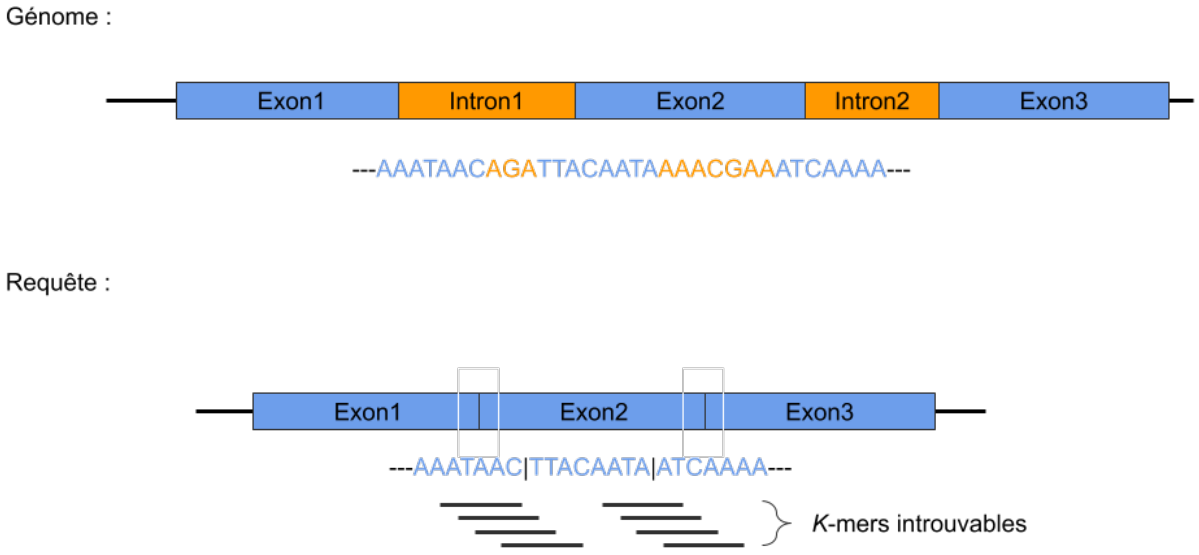


Figure 7.16: Schéma expliquant notre hypothèse sur le fait que les régions proches des exons ne sont pas représentées par le découpage en approche k-mers, tous les k-mers qui chevauchent les jonctions des exons ne seront pas présent dans le génome.

Figure 7.17: Résultat obtenu avec *RedOak* de la recherche de la séquence `"ERECTA"` (qui est une séquence génomique propre à la plante modèle *Arabidopsis thaliana* et qui est absente dans les génomes de riz) dans les 67 génomes de riz proposé par [Zhao et al., 2018].

Finalement nous avons observé qu'une approche par découpage en k-mers ne permet pas de représenter les régions voisines des exons une fois que les introns ont été retirés.

Synthèse

Nous avons vu dans ce chapitre 7 les performances de l'outil *RedOak*, sa capacité à créer un index et l'interroger et qu'il été possible de vérifier rapidement des résultats biologiques comme la présence ou l'absence d'un gène. Nous avons présenté les performances de l'outil des différentes versions, ce qui nous a permis de montrer l'impact de la taille du préfixe en nucléotides, et l'évolution de l'index, qui est le résultat de l'indexation faite par l'outil *RedOak*, en fonction du nombre de k-mers indexés. Il a été introduit ensuite une comparaison avec d'autres outils, nous avons comparé le temps d'exécution pour la création de l'index et le temps d'interrogation de ce dernier. Nous avons également évalué la mémoire nécessaire pour créer et interroger les index créés avec les outils étudiés. Finalement nous nous sommes focalisé sur une requête particulière celle de la séquence du gène PSTOL.

Chapitre 8

Conclusion & Perspectives

De nombreux résultats ont été présentés tout au long de ce manuscrit. Nous avons abordé la notion d'indexation d'une collection de génomes similaires en introduisant dans un premier temps la notion de génome, puis d'indexation, puis de pan-génomes ; cette dernière s'apparentant à l'étude d'une collection de génomes similaires. Une fois les définitions intégrées nous avons abordé les études préliminaires sur les génomes de riz qui nous ont mené à la création de notre structure dont l'implémentation est nommée *RedOak*.

8.1 Synthèse

Les méthodes de pan-génomique sont apparues récemment et ne passent pas à l'échelle des données du riz, elles ne semblent pas être applicables sur des génomes ayant subi de nombreuses recombinaisons. Nous avons donc proposé une solution permettant de pallier le problème d'indexation de milliers de génomes dit mosaïques. Cette solution s'est accompagnée de plusieurs contributions. De nombreux scripts sont alors implémentés pour avoir une idée de la forme du langage de notre ensemble et permettre cette première phase exploratoire.

Nous avons, dans un travail de réflexion sur les aspects combinatoires, créé *LEGO* : un script en python qui utilise la librairie de *Jellyfish* pour découper et compter des k-mers. Cet outil nous a permis rapidement de voir qu'il serait efficace de découper nos génomes en k-mers et permettre une factorisation. L'outil permet de créer des diagrammes de Venn et d'étudier les ensembles. Dans notre cas un ensemble est composé de tous les k-mers d'un génome. Cet outil peu robuste et implementé en python à rapidement été limité pour notre questions principale qui était de savoir quelle est la répartition de nos k-mers en *core*, *shell* et *cloud* k-mers.

Pour approfondir cette étude nous avons implémenté *PICTURES* : un logiciel écrit en C qui prend les comptages de k-mers de *Jellyfish* sauvegardés sur disque. Cet outil plus puissant nous a permis de prendre conscience de la répartition de nos k-mers.

Deux outils majeurs (*gkampi* et *RedOak*) sont directement issus de ces travaux de recherche algorithmiques.
Le premier *gkampi* permet le comptage ET l'indexation de k-mers. Aucune autre approche hormis les *Gk-Arrays* et *PgSA* [Philippe et al., 2011, Kowalski et al., 2015] ne fait de l'indexation de k-mers. De plus la version massivement parallélisée et distribuée des *Gk-Arrays*,

la `libGkArrays-MPI` est plus économe en mémoire et passe nécessairement à l'échelle.
Le second *RedOak* permet l'indexation des k-mers issus des collections de génomes, sans qu'il y ait nécessairement un génome de référence. Cet outil est plus performant que les approches que nous avons implémentées en amont avec des compteurs de k-mers. De surcroît, cet outil passe à l'échelle, là où les autres méthodes testées comme CHICO ou BFT échouent sur nos données. La principale différence entre notre approche et les approches existantes est que *RedOak* est massivement parallélisée et permet une distribution des données sur plusieurs machines.
Les outils *gkampi* et *RedOak* sont initialement des prototypes, qui ont été conçus pour être malgré tout utilisables par la communauté. Nous les avons testés sur différentes architectures tels que des ordinateurs portables, ordinateurs de bureau, réseau artisanal, *clusters* de calculs. Ils ont été également testé sur plusieurs systèmes d'exploitation, macOS et Linux : Catalina 10.15.5, Ubuntu (Kubuntu, Ubuntu, Xubuntu), Debian, Manjaro (20.0 Lysia).

L'ensemble des algorithmes, logiciels ou programmes implémentés ont tous été versionnés sur `git`. Pour accéder à *LEGO* veuillez vous référer à l'URL suivante : `https://gitlab.southgreen.fr/ALICIA/ALICIA/blob/master/Dev/DevPython/lego.py`.
Davantage d'informations sur l'outil PICTURES et son implémentation sont accessibles à l'adresse suivante : `https://gitlab.southgreen.fr/ALICIA/PICTURES`.
La documentation détaillée de l'outil *RedOak* est disponible sur : `https://gitlab.southgreen.fr/GenomeHarvest/RedOak-Silly`.
Des informations complémentaires sur *RedOak* sont présentées dans le résumé de la présentation orale [Agret et al., 2018] effectuée lors du groupe de travail SeqBIM (anciennement SeqBio est un *workshop* scientifique qui a lieu sur plusieurs journées et qui se déroule chaque année. SeqBio est organisé par le groupe COMATEGE commun au GDR BIM [Bioinformatique Moléculaire] et au GDR IM [Informatique Mathématique]).

Les limites des contributions de ma thèse sont la visualisation des résultats produits par *RedOak*. Il aurait été souhaité une interface graphique pour valoriser les interactions Homme-Machine (IHM). L'intégration de *RedOak* dans une interface ou sous forme de brique `Galaxy` est tout à fait possible. Cela nécessite cependant des ressources en ingénierie supplémentaires et sort du cadre de ma thèse.
RedOak dispose de nombreuses fonctionnalités mais ces possibilités ne sont accessibles que pour des personnes maîtrisant le mode console et les lignes de commandes.
Une interface graphique (IG) aurait rendu les fonctionnalités, comme requêter une séquence, accessible à partir d'un simple copier-coller ou à partir d'un fichier qui représente visuellement le résultat. La représentation visuelle aurait pu simplement être une représentation graphique de la sortie texte actuelle. Une belle plus-value aurait été d'implémenter une IG cliquable avec davantage d'informations.

Les performances peuvent également représenter une limitation. Nous avons montré que pour les structures fortement éparses, l'implémentation de *RUBIKS* pourrait pallier cette limitation d'utilisation de la mémoire. Notamment pour indexer une large collection de génomes similaires.
Il serait également intéressant de se poser la question d'une version de *RUBIKS* optimisée sur serveur.
Une étude intéressante aurait été d'indexer séquentiellement un nombre de génomes croissant pour voir les réelles limites de l'outil et de les comparer avec un algorithme naïf basé sur un simple *grep*.

8.2 Perspectives

Nous avons vu dans l'optique d'une évaluation des limites, différents point restant à améliorer. Il serait judicieux d'étendre nos études à d'autres contextes et espèces : faire des études plus ciblées sur les espèces étudiées au sein du projet *GenomeHarvest* comme par exemple le bananier. L'outil *RedOak* est disponible en open source sur le gitlab de la plateforme southgreen et de fait l'ensemble des acteurs du projet *GenomeHarvest* va pouvoir continuer d'utiliser et d'améliorer cet outil.
Il serait également intéressant de mener une étude de loi de distribution du nombre de *core/cloud/shell* k-mers.
Cette étude permettrait de savoir si il existe une taille optimale de k en nucléotides propre à chaque espèces. Un autre point qui me semble brûlant d'étudier et d'approfondir est la *phylokmérie*, concept que nous proposons comme l'étude des liens existants entre espèces sur la base de leur contenus en k-mers. Notre idée est qu'à partir des k-mers il serait possible de retracer les principales étapes de l'évolution des organismes et d'étudier les relations d'appartenance à un même groupe entre les êtres vivants.

Ce travail de thèse offre une multitude de possibilités. Dans un premier temps, l'outil implémenté *RedOak* permet une nouvelle approche par k-mers pour détecter rapidement la présence de certains gènes dans une population donnée, il est donc simple d'utiliser *RedOak* pour créer des PAV à partir de données pangénomique et faire l'analyse des résultats obtenus. Ensuite il est possible de travailler la sortie de *RedOak* pour faciliter les analyses en pangénomique et utiliser le travail d'indexation pour offrir un outil complet d'alignement en génomique. Pour créer un outil complet d'alignement il faut modifier *RedOak* en RedOak-Aligner : cette nouvelle version devra garder la position des k-mers sur les génomes assemblés et il faudrait créer un nouvel exécutable basé sur un script existant `RedOakQueriesFromFile.pl`. Celui-ci, en plus de créer un ensemble de requêtes formatées pour *RedOak* à partir d'un fichier fasta, récupère les positions et propose un alignement de chaque k-mer pour chaque génome. Finalement, il serait pertinent d'explorer cette nouvelle piste de *phylokmérie* en emmêlant combinatoire et statistique. La piste qu'il me semble judicieux d'explorer est d'appliquer un apprentissage profond en utilisant l'algorithme de réseau de neurone pour apprendre sur les millions d'arbres potentiels. Une fois découpé et indexé, chaque préfixe a un ensemble de suffixes correspondant à un ensemble de génomes. Pour chaque préfixe il est possible de faire un arbre de parenté. Nous avons donc éventuellement 4^{11} soit 4 194 304 arbres correspondant à un arbre par préfixe.

8.3 Bilan

Le doctorat est me semble-t-il la préparation par excellence au métier d'enseignant-chercheur. Il permet durant ces quelques années d'échanger avec de nombreux laboratoires de recherches, d'enseigner et de présenter lors de conférences. La thèse met en lumière certains enjeux du monde de la recherche comme les aspects politiques et scientifiques. Outre les connaissances acquises et développées pendant ces années, la formation doctorale permet d'appréhender les qualités à acquérir sur toute une carrière d'enseignant et de chercheur à savoir l'autonomie, la pédagogie et être doué pour l'abstraction.

Annexes

Annexe A

Utilisation de *RedOak*

Dans cette annexe il est présenté l'utilisation détaillée de *RedOak*, un outil implémenté en C++ utilisant la bibliotheque `libGkArrays-MPI`. *RedOak* vient avec une documentation complète, détaillée et générée automatiquement avec *Doxygen* [Heesch, 1997]. *Doxygen* est l'outil standard de référence pour générer de la documentation à partir des codes sources C++ correctement annotés. Cet outil permet également d'annoter d'autres langages de programmation populaires tels que le C, le PHP et le Java.

A.1 Documentation détaillée

Pour faire fonctionner *RedOak*, il est indispensable de lui fournir au choix un génome au format Fasta ou Fastq (compressé ou non compressé) en utilisant l'option `--genome`. Il est également possible d'indexer une liste de génomes au format YAML avec l'option `--list`.
Il y a au moins un argument : un fichier (ou une liste de fichiers séparés par des virgules), éventuellement compressé avec `gzip`. Des informations détaillées et des exemples d'utilisation de ce paramètre figurent à la fin de la section aide générée avec l'option `--help`.
Le code source est suffisamment complet au niveau des balises de *Doxygen* pour pouvoir être compilé et fournir une documentation au format HTML ou PDF complète.

A.1.1 Git

Git est un système de contrôle de version gratuit et open source conçu pour gérer des projets personnels comme professionnels avec rapidité et efficacité. Un logiciel de gestion de versions est un logiciel qui permet de stocker un ensemble de fichiers. Git permet également de travailler en parallèle et en équipe sur un projet commun tout en conservant la chronologie de toutes les modifications qui ont été effectuées dessus.
Le code de *RedOak* a été complètement versionné sur `https://gitlab.southgreen.fr/GenomeHarvest/RedOak`. Sur le dépôt de Git vous trouverez toutes les sources des fichiers binaires dans le répertoire source, et l'ensemble des étapes à suivre pour installer *RedOak* proprement avec les `autotools` se trouve dans le fichier `README`.

A.1.2 Guide d'utilisation de *RedOak*

Prérequis

Les prérequis de l'outil *RedOak*

- Un compilateur moderne pour C++ 11 tel que « g++ » version 4.9 ou supérieure ou « clang » version 3.2 ou supérieure.
- Un système d'exploitation Mac OS X ou Linux de 64 bits.
- OpenMPI
- libGkArrays-MPI
- Doxygen

algorithme A.1: Installer et utiliser RedOak

```
#Pour telecharger et installer RedOak dans le repertoire
#local_install : Veuillez installer d'abord la bibliotheque
#libGkArrays-MPI en suivant les etapes suivantes.
git clone --recursive \
git@gitlab.info-ufr.univ-montp2.fr:doccy/libGkArrays-MPI.git
cd libGkArrays-MPI && ./configure --prefix=$HOME/local_install
make && make install

#Ensuite, installez RedOak a l'aide des commandes suivantes.
git clone git@gitlab.southgreen.fr:GenomeHarvest/RedOak.git
cd RedOak
./configure --with-libGkArraysMPI-prefix=$HOME/local_install \
            --prefix=$HOME/local_install
make && make install

#Testez l'installation a l'aide de la commande suivante.
mpirun -n 10 \
  ${HOME}/local_install/redoak \
    --genome resources/test-0025_reads.fastq \
    --kmer 20

#SGE MakeQsubYaml.sh script:
./resources/makeRedOakQsubYaml.sh

#Pour supprimer RedOak et la bibliotheque libGkArrays-MPI
cd RedOak && make uninstall
cd libGkArrays-MPI && make uninstall
```

Installation

Utilisation : redoak [*options*]

```
RedOak aims to index a large collection of similar genomes

Redoak version 0.1.1 -- Copyright (c) 2017-2019
```

```
Compiled at May 19 2020 23:09:00 with mpic++ for x86_64/linux-gnu

Usage: Redoak [options]

* Where options can be:
--genome, -g [infos:]<files>      Index the genome described by the given
                              argument. The argument is at least a file
                              (or a comma separated list of files),
                              eventually compressed with gzip. Detailled
                              informations and examples about how to use
                              this parameter at the end of this section.
--list, -l <file>                 Use the content of the given (raw formatted)
                              <file> to load genomes.
                              Example:
                                --list my_genomes.lst
--config, -c <file>               Use the content of the given (YAML
                              formatted) <file> to load genomes and
                              initialize settings.
                              Example:
                                --config my_genomes.cfg
--kmer, -k <int>                  Set the value of parameter k to the given
                              value.
                              Example:
                                --kmer 30 or -k 30
--prefix-size, -p <int>           Set the value of parameter k1 to the given
                              value.
                              Example:
                                --prefix-size 12 or -p 12
--query, -Q [<label>:]<sequence>  Search the given <sequence> k-mers in the
                              index and display the results.
                              If some <label> is prepended to the
                              sequence, then this <label> is printed as a
                              comment.
                              Examples:
                               --query
                              'ATAACGAGGGATGCTGGGTAAAATGCAAAGCTAG'
                               --query 'example
                              query:ATAACGAGGGATGCTGGGTAAAATGCAAAGCTAG'
--query-file, -F <sequence_files> For each sequence in the <sequence_files>
                              search the sequence in the index and print
                              the genomes with this sequence.
                              The query file can be either a fasta
                              formatted file (each sequence being a query)
                              or a one line raw query formatted file
                              (following the same syntax as the query
                              '--option'). In the latter case, the only
                              requirement is that the first character of
                              the query file is different from '>' ; One
                              can also add comments starting with '#'
                              until the end of line.
```

```
                              Examples:
                               --query-file 'sequence_files'
--shell, -S                        Launch an shell to interact with the current
                              index. Within MPI (which is the normal
                              behaviour), the processus having rank 0
                              creates input/output/error files to handle
                              IO operations.
--output, -o <filename>            Dump the current index in text format to the
                              given file.
--profile <filename>               Dump a CSV formatted file containing
                              informations and statistics.
--logo                             Print ASCII art logo, then exit.
--verbose, -v                      Verbose output (can be provided many times
                              to increase verbosity).
                              Examples:
                                 --verbose
                                 -v verbose
                                 -vv more verbose
                                 -vvv even more verbose
--quiet, -q                        Run silently.


* Other usage:
--help, -h                         Print usage and exit.
--version, -V                      Print the version informations and exit.

* Passing genomes to the command line with the --genome/-g option:

Some additionnal information can be specified using colons as field
separator.
If some information is provided, the fields are interpreted from left to right
and the rightmost empty fields can be omitted.

Fields from left to right are:
- Genome name
- Genome version
- Occurrence range for which k-mers are integrated in the index and option to
store the sequence headers and lengths

Examples:
* To index a single (compressed fasta) file without any extra information:
--genome gen1.fa.gz
* To index several (compressed or not) files without any extra information:
--genome gen2_chr1.fa,gen2_chr2.fa.gz,gen2_chr3.fa
* To index several files and provide a genome name (notice that the white
spaces must be escaped or the whole argument must be enclosed within
braquets):
--genome My\ Third\ genome:gen3_chr1.fa,gen3_chr2.fa.gz
* To index several (fastq) files setting both a genome name and a genome
version:
```

```
--genome Unassembled\ genome:Version\ 0.1.2:sample1.fq,sample2.fq.gz
* To collect the sequence headers and length (be aware that it consumes a lot
of time and memory):
--genome ::+:sample5.fq.gz,sample6.fq.gz
* To index several files setting a genome name and a minimum number as well as
a maximum number of occurrences for k-mers to be indexed (notice that the
genome version is empty):
--genome Another\ genome::[5,150]:sample3.fq,sample4.fq.gz
* To index several files setting only a minimum number of occurrences for
k-mers to be indexed (notice that both the genome name and genome version are
omitted, as well as the maximum number of occurrences):
--genome ::[5,]:sample5.fq.gz,sample6.fq.gz
* To collect the sequence headers and length and filter the kmers having more
than 10 occurrences:
--genome ::+[,10]:sample5.fq.gz,sample6.fq.gz

It is also possible to provide an index file dumped from the gkampi program.
In such a case, it must be a single file:
--genome My\ Indexed\ Genome:indexed_file.gki

* Passing genomes to the command line with the --list/-l option:

Each line of the listing file is consider as a genome description. The same
syntax as for the --genome option is used.

* Passing genomes and settings to the command line with the --config/-c option:

The specified argument must be a YAML formatted file, such that each genome
entry follows the description below:
Genome: # extra fields within this structure will be simply ignored
- Name: <Genome name> # optionnal
- Version: <Genome version> # optional
- Files: # optional
  - <file 1> # file must exist
    ...
  - <file n> # file must exist
- Chromosomes: # optional
- Index: # optional
  - File: <index_file> # optional
  - Min k-mer occurrence: <value> # optional
  - Max k-mer occurrence: <value> # optional
# At least one file must be specified either in the 'Files'
# or in the 'Index' subsection.

Example:
The following YAML description does the same than passing all the '--genome'
previous examples
Genome: # --genome gen1.fa.gz
- Files:
  - gen1.fa.gz
```

```
Genome: # --genome gen2_chr1.fa,gen2_chr2.fa.gz,gen2_chr3.fa
- Files:
   - gen2_chr1.fa
   - gen2_chr2.fa.gz
   - gen2_chr3.fa

Genome: # --genome My\ Third\ genome:gen3_chr1.fa,gen3_chr2.fa.gz
- Name: My Third genome
- Files:
   - gen3_chr1.fa
   - gen3_chr2.fa.gz

Genome: # --genome Unassembled\ genome:Version\ 0.1.2:sample1.fq,sample2.fq.gz
- Name: Unassembled genome
- Version: Version 0.1.2
- Files:
   - sample1.fq
   - sample2.fq.gz

Genome: # --genome ::+:sample5.fq.gz,sample6.fq.gz
- Files:
   - sample5.fq.gz
   - sample6.fq.gz
- Chromosomes:

Genome: # --genome Another\ genome::[5,150]:sample3.fq,sample4.fq.gz
- Name: Another genome
- Files:
   - sample3.fq
   - sample4.fq.gz
- Index:
   - Min k-mer occurrence: 5
   - Max k-mer occurrence: 150

Genome: # --genome ::[5,]:sample5.fq.gz,sample6.fq.gz
- Files:
   - sample5.fq.gz
   - sample6.fq.gz
- Index:
   - Min k-mer occurrence: 5

Genome: # --genome ::+[,10]:sample5.fq.gz,sample6.fq.gz
- Files:
   - sample5.fq.gz
   - sample6.fq.gz
- Chromosomes:
- Index:
   - Max k-mer occurrence: 10
```

```
Genome: # --genome My\ Indexed\ Genome:indexed_file.gki
- Name: My Indexed Genome
- Index:
   - File: indexed_file.gki

# Notice that the following syntax is will give the same result:
Genome: # --genome My\ Indexed\ Genome:indexed_file.gki
- Name: My Indexed Genome
- Files:
   - indexed_file.gki
...


* Contacts:
Thank you for using Redoak.
You can contact the developer team by e-mail at <no@mail.for.now>.
```

License

Ce logiciel est un programme informatique permettant d'indexer une large collection de génomes similaires.

Ce logiciel est régi par la licence CeCILL soumise au droit français et respectant les principes de diffusion des logiciels libres. Vous pouvez utiliser, modifier et/ou redistribuer ce programme sous les conditions de la licence CeCILL telle que diffusée par le CEA, le CNRS et l'INRIA sur le site `http://www.cecill.info`.

En contrepartie de l'accessibilité au code source et des droits de copie, de modification et de redistribution accordés par cette licence, il n'est offert aux utilisateurs qu'une garantie limitée. Pour les mêmes raisons, seule une responsabilité restreinte pèse sur l'auteur du programme, le titulaire des droits patrimoniaux et les concédants successifs.

À cet égard l'attention de l'utilisateur est attirée sur les risques associés au chargement, à l'utilisation, à la modification et/ou au développement et à la reproduction du logiciel par l'utilisateur étant donné sa spécificité de logiciel libre, qui peut le rendre complexe à manipuler et qui le réserve donc à des développeurs et des professionnels avertis possédant des connaissances informatiques approfondies. Les utilisateurs sont donc invités à charger et tester l'adéquation du logiciel à leurs besoins dans des conditions permettant d'assurer la sécurité de leurs systèmes et ou de leurs données et, plus généralement, à l'utiliser et l'exploiter dans les mêmes conditions de sécurité.

Le fait que vous puissiez accéder à cet en-tête signifie que vous avez pris connaissance de la licence CeCILL, et que vous en avez accepté les termes.

This software is a computer program whose purpose is to index a large collection of similar genomes.

Auteurs/Authors

— Clément AGRET <clement.agret@lirmm.fr>
— Annie CHATEAU <annie.chateau@lirmm.fr>
— Alban MANCHERON <alban.mancheron@lirmm.fr>
— Manuel RUIZ <manuel.ruiz@cirad.fr>
— Gautier SARAH <gautier.sarah@inra.fr>

Programmeurs/Programmers

— Clément AGRET <clement.agret@lirmm.fr>
— Alban MANCHERON <alban.mancheron@lirmm.fr>
— Gautier SARAH <gautier.sarah@inra.fr>

Contact

— Clément AGRET <clement.agret@lirmm.fr>

A.2 Capture d'écran

A.2.1 Output Query

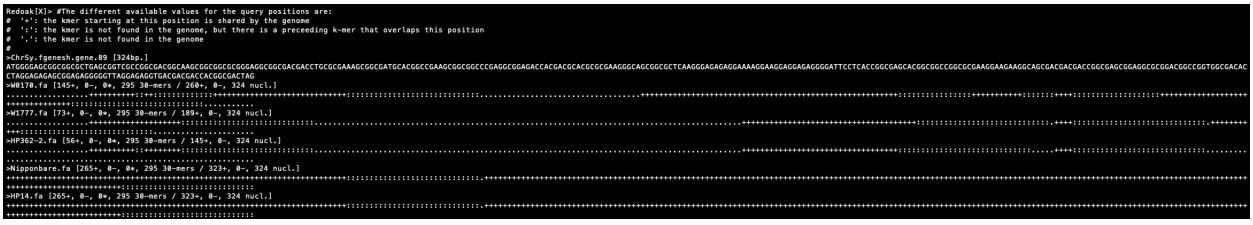


Figure A.1: .

Bibliographie

[Agret et al., 2018] Agret, C., Chateau, A., Sarah, G., Droc, G., Ruiz, M., and Mancheron, A. (2018).
Development of indexing compressed structure for analyzing a collection of similar genomes : application to rice .
Cette référence a été citée 1 fois à la page 116.

[Baier et al., 2016] Baier, U., Beller, T., and Ohlebusch, E. (2016).
Graphical pan-genome analysis with compressed suffix trees and the Burrows-Wheeler transform.
Bioinformatics (Oxford, England), 32(4) :497–504.
Cette référence a été citée 1 fois à la page 40.

[Beller and Ohlebusch, 2016] Beller, T. and Ohlebusch, E. (2016).
A representation of a compressed de bruijn graph for pan-genome analysis that enables search.
Algorithms Mol Biol, 11 :20.
Cette référence a été citée 1 fois à la page 40.

[Benoit et al., 2014] Benoit, G., Lavenier, D., Lemaitre, C., Rizk, G., and Bloocoo, G. R. (2014).
Bloocoo, a memory efficient read corrector.
Technical report.
Cette référence a été citée 1 fois à la page 18.

[Bentley et al., 2008] Bentley, D. R., Balasubramanian, S., Swerdlow, H. P., Smith, G. P., Milton, J., Brown, C. G., Hall, K. P., Evers, D. J., Barnes, C. L., Bignell, H. R., Boutell, J. M., Bryant, J., Carter, R. J., Keira Cheetham, R., Cox, A. J., Ellis, D. J., Flatbush, M. R., Gormley, N. A., Humphray, S. J., Irving, L. J., Karbelashvili, M. S., Kirk, S. M., Li, H., Liu, X., Maisinger, K. S., Murray, L. J., Obradovic, B., Ost, T., Parkinson, M. L., Pratt, M. R., Rasolonjatovo, I. M., Reed, M. T., Rigatti, R., Rodighiero, C., Ross, M. T., Sabot, A., Sankar, S. V., Scally, A., Schroth, G. P., Smith, M. E., Smith, V. P., Spiridou, A., Torrance, P. E., Tzonev, S. S., Vermaas, E. H., Walter, K., Wu, X., Zhang, L., Alam, M. D., Anastasi, C., Aniebo, I. C., Bailey, D. M., Bancarz, I. R., Banerjee, S., Barbour, S. G., Baybayan, P. A., Benoit, V. A., Benson, K. F., Bevis, C., Black, P. J., Boodhun, A., Brennan, J. S., Bridgham, J. A., Brown, R. C., Brown, A. A., Buermann, D. H., Bundu, A. A., Burrows, J. C., Carter, N. P., Castillo, N., Catenazzi, M. C. E., Chang, S., Neil Cooley, R., Crake, N. R., Dada, O. O., Diakoumakos, K. D., Dominguez-Fernandez, B., Earnshaw, D. J., Egbujor, U. C., Elmore, D. W., Etchin, S. S., Ewan, M. R., Fedurco, M., Fraser, L. J., Fuentes Fajardo, K. V., Scott Furey, W., George, D., Gietzen, K. J., Goddard, C. P., Golda, G. S., Granieri, P. A., Green, D. E., Gustafson, D. L., Hansen, N. F., Harnish, K., Haudenschild, C. D., Heyer, N. I., Hims, M. M., Ho, J. T., Horgan, A. M., Hoschler, K., Hurwitz, S., Ivanov, D. V., Johnson, M. Q., James, T., Huw Jones, T. A., Kang, G. D., Kerelska, T. H., Kersey, A. D., Khrebtukova, I., Kindwall, A. P., Kingsbury, Z., Kokko-Gonzales, P. I., Kumar, A.,

Laurent, M. A., Lawley, C. T., Lee, S. E., Lee, X., Liao, A. K., Loch, J. A., Lok, M., Luo, S., Mammen, R. M., Martin, J. W., McCauley, P. G., McNitt, P., Mehta, P., Moon, K. W., Mullens, J. W., Newington, T., Ning, Z., Ling Ng, B., Novo, S. M., O'Neill, M. J., Osborne, M. A., Osnowski, A., Ostadan, O., Paraschos, L. L., Pickering, L., Pike, A. C., Pike, A. C., Chris Pinkard, D., Pliskin, D. P., Podhasky, J., Quijano, V. J., Raczy, C., Rae, V. H., Rawlings, S. R., Chiva Rodriguez, A., Roe, P. M., Rogers, J., Rogert Bacigalupo, M. C., Romanov, N., Romieu, A., Roth, R. K., Rourke, N. J., Ruediger, S. T., Rusman, E., Sanches-Kuiper, R. M., Schenker, M. R., Seoane, J. M., Shaw, R. J., Shiver, M. K., Short, S. W., Sizto, N. L., Sluis, J. P., Smith, M. A., Ernest Sohna Sohna, J., Spence, E. J., Stevens, K., Sutton, N., Szajkowski, L., Tregidgo, C. L., Turcatti, G., Vandevondele, S., Verhovsky, Y., Virk, S. M., Wakelin, S., Walcott, G. C., Wang, J., Worsley, G. J., Yan, J., Yau, L., Zuerlein, M., Rogers, J., Mullikin, J. C., Hurles, M. E., McCooke, N. J., West, J. S., Oaks, F. L., Lundberg, P. L., Klenerman, D., Durbin, R., and Smith, A. J. (2008).
Accurate whole human genome sequencing using reversible terminator chemistry.
Nature, 456(7218) :53–59.
Cette référence a été citée 1 fois à la page 11.

[Břinda et al., 2015] Břinda, K., Sykulski, M., and Kucherov, G. (2015).
Spaced seeds improve k-mer-based metagenomic classification.
31(22) :3584–3592.
Cette référence a été citée 1 fois à la page 73.

[Chapman et al., 2011] Chapman, J. A., Ho, I., Sunkara, S., Luo, S., Schroth, G. P., and Rokhsar, D. S. (2011).
Meraculous : De Novo Genome Assembly with Short Paired-End Reads.
PLoS ONE, 6(8) :e23501.
Cette référence a été citée 1 fois à la page 17.

[Chong and Xu, 2014] Chong, K. and Xu, Z. (2014).
Investment in plant research and development bears fruit in China.
Cette référence a été citée 1 fois à la page 15.

[Citation : Luo et al., 2012] Citation : Luo, C., Tsementzi, D., Kyrpides, N., Read, T., and Konstantinidis, K. T. (2012).
Direct Comparisons of Illumina vs. Roche 454 Sequencing Technologies on the Same Microbial Community DNA Sample.
PLoS ONE, 7(2) :30087.
Cette référence a été citée 1 fois à la page 11.

[Collins et al., 2004] Collins, F. S., Lander, E. S., Rogers, J., and Waterson, R. H. (2004).
Finishing the euchromatic sequence of the human genome.
Nature, 431(7011) :931–945.
Cette référence a été citée 1 fois à la page 11.

[Computational Pan-Genomics, 2016] Computational Pan-Genomics, C. (2016).
Computational pan-genomics : status, promises and challenges.
Brief Bioinform.
Cette référence a été citée 1 fois à la page 100.

[Conway et al., 2017] Conway, J. R., Lex, A., and Gehlenborg, N. (2017).
UpSetR : An R package for the visualization of intersecting sets and their properties.
Bioinformatics, 33(18) :2938–2940.
Cette référence a été citée 1 fois à la page 53.

[Croucher et al., 2015] Croucher, N. J., Finkelstein, J. A., Pelton, S. I., Parkhill, J., Bentley, S. D., Lipsitch, M., and Hanage, W. P. (2015).
Population genomic datasets describing the post-vaccine evolutionary epidemiology of Streptococcus pneumoniae.
Scientific Data, 2(1) :1–9.
Cette référence a été citée 1 fois à la page 39.

[Danek et al., 2014] Danek, A., Deorowicz, S., and Grabowski, S. (2014).
Indexes of large genome collections on a pc.
PLOS ONE, 9(10) :1–12.
Cette référence a été citée 1 fois à la page 40.

[Deorowicz et al., 2013a] Deorowicz, S., Danek, A., and Grabowski, S. (2013a).
Genome compression : a novel approach for large collections.
Bioinformatics, 29(20) :2572–2578.
Cette référence a été citée 1 fois à la page 40.

[Deorowicz et al., 2013b] Deorowicz, S., Debudaj-Grabysz, A., and Grabowski, S. (2013b).
Disk-based k-mer counting on a PC.
BMC Bioinformatics, 14(1).
Cette référence a été citée 2 fois aux pages 18 et 20.

[Deorowicz et al., 2014] Deorowicz, S., Kokot, M., Grabowski, S., and Debudaj-Grabysz, A. (2014).
KMC 2 : Fast and resource-frugal k-mer counting.
Bioinformatics, 31(10) :1569–1576.
Cette référence a été citée 1 fois à la page 68.

[D'Hont et al., 2012] D'Hont, A., Denoeud, F., Aury, J. M., Baurens, F. C., Carreel, F., Garsmeur, O., Noel, B., Bocs, S., Droc, G., Rouard, M., Da Silva, C., Jabbari, K., Cardi, C., Poulain, J., Souquet, M., Labadie, K., Jourda, C., Lengelle, J., Rodier-Goud, M., Alberti, A., Bernard, M., Correa, M., Ayyampalayam, S., McKain, M. R., Leebens-Mack, J., Burgess, D., Freeling, M., Mbeguie, A. M. D., Chabannes, M., Wicker, T., Panaud, O., Barbosa, J., Hribova, E., Heslop-Harrison, P., Habas, R., Rivallan, R., Francois, P., Poiron, C., Kilian, A., Burthia, D., Jenny, C., Bakry, F., Brown, S., Guignon, V., Kema, G., Dita, M., Waalwijk, C., Joseph, S., Dievart, A., Jaillon, O., Leclercq, J., Argout, X., Lyons, E., Almeida, A., Jeridi, M., Dolezel, J., Roux, N., Risterucci, A. M., Weissenbach, J., Ruiz, M., Glaszmann, J. C., Quetier, F., Yahiaoui, N., and Wincker, P. (2012).
The banana (musa acuminata) genome and the evolution of monocotyledonous plants.
Nature, 488(7410) :213–7.
Cette référence a été citée 1 fois à la page xii.

[Dilthey et al., 2015] Dilthey, A., Cox, C., Iqbal, Z., Nelson, M. R., and McVean, G. (2015).
Improved genome inference in the MHC using a population reference graph.
Nature Genetics, 47(6) :682–688.
Cette référence a été citée 1 fois à la page 38.

[Ding et al., 2017] Ding, W., Baumdicker, F., and Neher, R. A. (2017).
panX : pan-genome analysis and exploration.
Nucleic Acids Research, 46(1) :e5–e5.
Cette référence a été citée 1 fois à la page 39.

[Erbert et al., 2017] Erbert, M., Rechner, S., and Müller-Hannemann, M. (2017).
Gerbil : a fast and memory-efficient k-mer counter with gpu-support.

Algorithms for Molecular Biology, 12(1) :9.
Cette référence a été citée 1 fois à la page 68.

[Ernst and Rahmann, 2013] Ernst, C. and Rahmann, S. (2013).
PanCake : A Data Structure for Pangenomes.
In Beißbarth, T., Kollmar, M., Leha, A., Morgenstern, B., Schultz, A.-K., Waack, S., and Wingender, E., editors, *German Conference on Bioinformatics 2013*, volume 34 of *OpenAccess Series in Informatics (OASIcs)*, pages 35–45, Dagstuhl, Germany. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.
Cette référence a été citée 2 fois aux pages 38 et 40.

[Ferragina and Manzini, 2001] Ferragina, P. and Manzini, G. (2001).
An experimental study of a compressed index.
Information Sciences, 135(1) :13–28.
Cette référence a été citée 2 fois aux pages 31 et 39.

[Finkel and Bentley, 1974] Finkel, R. A. and Bentley, J. L. (1974).
Quad trees a data structure for retrieval on composite keys.
Acta Informatica, 4(1) :1–9.
Cette référence a été citée 1 fois à la page 55.

[Gagie et al., 2017] Gagie, T., Navarro, G., and Prezza, N. (2017).
Optimal-Time Text Indexing in BWT-runs Bounded Space.
Cette référence a été citée 1 fois à la page 31.

[Gamuyao et al., 2012] Gamuyao, R., Chin, J. H., Pariasca-Tanaka, J., Pesaresi, P., Catausan, S., Dalid, C., Slamet-Loedin, I., Tecson-Mendoza, E. M., Wissuwa, M., and Heuer, S. (2012).
The protein kinase pstol2 from traditional rice confers tolerance of phosphorus deficiency.
Nature, 488(7412) :535–9.
Cette référence a été citée 1 fois à la page 104.

[Garsmeur et al., 2018] Garsmeur, O., Droc, G., Antonise, R., Grimwood, J., Potier, B., Aitken, K., Jenkins, J., Martin, G., Charron, C., Hervouet, C., Costet, L., Yahiaoui, N., Healey, A., Sims, D., Cherukuri, Y., Sreedasyam, A., Kilian, A., Chan, A., Van Sluys, M. A., Swaminathan, K., Town, C., Berges, H., Simmons, B., Glaszmann, J. C., van der Vossen, E., Henry, R., Schmutz, J., and D'Hont, A. (2018).
A mosaic monoploid reference sequence for the highly complex genome of sugarcane.
Nat Commun, 9(1) :2638.
Cette référence a été citée 1 fois à la page xii.

[Giordano et al., 2018] Giordano, F., Stammnitz, M. R., Murchison, E. P., and Ning, Z. (2018).
scanpav : a pipeline for extracting presence-absence variations in genome pairs.
Bioinformatics, 34(17) :3022–3024.
Cette référence a été citée 1 fois à la page 36.

[Gog et al., 2014] Gog, S., Beller, T., Moffat, A., and Petri, M. (2014).
From theory to practice : Plug and play with succinct data structures.
In *13th International Symposium on Experimental Algorithms, (SEA 2014)*, pages 326–337.
Cette référence a été citée 4 fois aux pages 25, 26, 27 et 48.

[Golicz et al., 2016] Golicz, A. A., Batley, J., and Edwards, D. (2016).
Towards plant pangenomics.
Plant Biotechnol J, 14(4) :1099–105.
Cette référence a été citée 1 fois à la page 37.

[Gusfield, 1997] Gusfield, D. (1997).
Algorithms on Strings, Trees, and Sequences : Computer Science and Computational Biology.
Cambridge University Press.
Cette référence a été citée 1 fois à la page 28.

[Hazelhurst and Lipták, 2011] Hazelhurst, S. and Lipták, Z. (2011).
KABOOM ! A new suffix array based algorithm for clustering expression data.
Bioinformatics, 27(24) :3348–3355.
Cette référence a été citée 1 fois à la page 18.

[Heesch, 1997] Heesch, D. (1997).
Doxygen.
Cette référence a été citée 1 fois à la page 121.

[Herbig et al., 2012] Herbig, A., Jager, G., Battke, F., and Nieselt, K. (2012).
GenomeRing : alignment visualization based on SuperGenome coordinates.
Bioinformatics, 28(12) :i7–i15.
Cette référence a été citée 2 fois aux pages 37 et 38.

[Holley et al., 2016] Holley, G., Wittler, R., and Stoye, J. (2016).
Bloom Filter Trie : an alignment-free and reference-free data structure for pan-genome storage.
Algorithms for Molecular Biology, 11(1) :3.
Cette référence a été citée 3 fois aux pages 19, 40 et 97.

[Hu et al., 2018] Hu, Z., Wang, W., Wu, Z., Sun, C., Li, M., Lu, J., Fu, B., Shi, J., Xu, J., Ruan, J., Wei, C., and Li, Z. (2018).
Novel sequences, structural variations and gene presence variations of asian cultivated rice.
Sci Data, 5 :180079.
Cette référence a été citée 1 fois à la page 100.

[Huang et al., 2013] Huang, L., Popic, V., and Batzoglou, S. (2013).
Short read alignment with populations of genomes.
Bioinformatics, 29(13) :i361–70.
Cette référence a été citée 1 fois à la page 38.

[Huffman, 1952] Huffman, D. A. (1952).
A method for the construction of minimum-redundancy codes.
Proceedings of the IRE, 40(9) :1098–1101.
Cette référence a été citée 1 fois à la page 25.

[IUPAC IUBMB JCMB, 1970] IUPAC IUBMB JCMB (1970).
Abbreviations and Symbols for Nucleic Acids, Polynucleotides and their Constituants.
IUPAC : International Union of Pure and Applied Chemistry,
IUBMB : International Union of Biochemistry and Molecular Biology,
JCBM : Joint Commission on Biochemical Nomenclature.
Cette référence a été citée 1 fois à la page 6.

[IUPAC IUBMB JCMB, 1983] IUPAC IUBMB JCMB (1983).
Abbreviations and Symbols for the Description of Conformations of Polynucleotide chains.
IUPAC : International Union of Pure and Applied Chemistry,
IUBMB : International Union of Biochemistry and Molecular Biology,
JCBM : Joint Commission on Biochemical Nomenclature.
Cette référence a été citée 1 fois à la page 19.

[Izawa et al., 2009] Izawa, T., Konishi, S., Shomura, A., and Yano, M. (2009).

DNA changes tell us about rice domestication.
Cette référence a été citée 1 fois à la page 15.

[Jandrasits et al., 2018a] Jandrasits, C., Dabrowski, P. W., Fuchs, S., and Renard, B. Y. (2018a).
seq-seq-pan : building a computational pan-genome data structure on whole genome alignment.
BMC Genomics, 19(1) :47.
Cette référence a été citée 1 fois à la page 38.

[Jandrasits et al., 2018b] Jandrasits, C., Dabrowski, P. W., Fuchs, S., and Renard, B. Y. (2018b).
seq-seq-pan : building a computational pan-genome data structure on whole genome alignment.
BMC Genomics, 19(1) :47.
Cette référence a été citée 1 fois à la page 40.

[Kelley et al., 2010] Kelley, D. R., Schatz, M. C., and Salzberg, S. L. (2010).
Quake : Quality-aware detection and correction of sequencing errors.
Genome Biology, 11(11) :R116.
Cette référence a été citée 1 fois à la page 18.

[Khiste and Ilie, 2017] Khiste, N. and Ilie, L. (2017).
HISEA : HIerarchical SEed Aligner for PacBio data.
BMC Bioinformatics, 18(1) :564.
Cette référence a été citée 1 fois à la page 18.

[Koonin, 2005] Koonin, E. V. (2005).
Orthologs, Paralogs, and Evolutionary Genomics.
Annual Review of Genetics, 39(1) :309–338.
Cette référence a été citée 1 fois à la page 8.

[Koonin and Wolf, 2008] Koonin, E. V. and Wolf, Y. I. (2008).
Genomics of bacteria and archaea : the emerging dynamic view of the prokaryotic world.
Nucleic Acids Res, 36(21) :6688–719.
Cette référence a été citée 1 fois à la page 37.

[Kovach et al., 2009] Kovach, M. J., Calingacion, M. N., Fitzgerald, M. A., and McCouch, S. R. (2009).
The origin and evolution of fragrance in rice (oryza sativa l.).
Proceedings of the National Academy of Sciences, 106(34) :14444–14449.
Cette référence a été citée 1 fois à la page 15.

[Kowalski et al., 2015] Kowalski, T., Grabowski, S., and Deorowicz, S. (2015).
Indexing arbitrary-length k-mers in sequencing reads.
PLOS ONE, 10(7) :1–16.
Cette référence a été citée 3 fois aux pages 24, 31 et 115.

[Lander and Waterman, 1988] Lander, E. S. and Waterman, M. S. (1988).
Genomic mapping by fingerprinting random clones : A mathematical analysis.
Genomics, 2(3) :231 – 239.
Cette référence a été citée 1 fois à la page 13.

[Li et al., 2014] Li, J. Y., Wang, J., and Zeigler, R. S. (2014).
The 3,000 rice genomes project : new opportunities and challenges for future rice research.
Gigascience, 3 :8.
Cette référence a été citée 3 fois aux pages xii, 40 et 97.

[Li et al., 2008] Li, R., Li, Y., Kristiansen, K., and Wang, J. (2008).
SOAP : short oligonucleotide alignment program.
Bioinformatics, 24(5) :713–714.
Cette référence a été citée 1 fois à la page 18.

[Lindner et al., 2016] Lindner, M. S., Strauch, B., Schulze, J. M., Tausch, S. H., Dabrowski, P. W., Nitsche, A., and Renard, B. Y. (2016).
HiLive : real-time mapping of illumina reads while sequencing.
Bioinformatics, 33(6) :917–319.
Cette référence a été citée 1 fois à la page 18.

[Luscombe et al., 2001] Luscombe, N., Greenbaum, D., and Gerstein, M. (2001).
What is bioinformatics ? a proposed definition and overview of the field.
PubMed.
Cette référence a été citée 2 fois aux pages 8 et 9.

[Ma and Bennetzen, 2004] Ma, J. and Bennetzen, J. (2004).
Ma j, bennetzen jl. rapid recent growth and divergence of rice nuclear genomes. proc natl acad sci usa 101 : 12404-12410.
Proceedings of the National Academy of Sciences of the United States of America, 101 :12404–10.
Cette référence a été citée 1 fois à la page 15.

[Manber and Myers, 1993a] Manber, U. and Myers, G. (1993a).
Suffix arrays : A new method for on-line string searches.
SIAM Journal on Computing, 22(5) :935–948.
Cette référence a été citée 1 fois à la page 30.

[Manber and Myers, 1993b] Manber, U. and Myers, G. (1993b).
Suffix arrays : A new method for on-line string searches.
SIAM Journal on Computing, 22(5) :935–948.
Cette référence a été citée 1 fois à la page 39.

[Mancheron et al., 2020] Mancheron, A., Agret, C., Buffard, M., Chateau, A., Philippe, N., and Salson, M. (2020).
Browse the ocean of NGS data without drowning with Gkampi.
In prep.
Cette référence a été citée 1 fois à la page 61.

[Manjunath et al., 2017] Manjunath, M., Zhang, Y., Yeo, S., Sobh, O., Russell, N., Followell, C., Bushell, C., Ravaioli, U., and Song, J. (2017).
ClusterEnG : An interactive educational web resource for clustering big data.
bioRxiv, page 120915.
Cette référence a été citée 1 fois à la page 18.

[Marcais and Kingsford, 2011] Marcais, G. and Kingsford, C. (2011).
A fast, lock-free approach for efficient parallel counting of occurrences of k-mers.
Bioinformatics, 27(6) :764–770.
Cette référence a été citée 5 fois aux pages 18, 20, 50, 68 et 97.

[Marcus et al., 2014] Marcus, S., Lee, H., and Schatz, M. C. (2014).
SplitMEM : a graphical algorithm for pan-genome analysis with suffix skips.
Bioinformatics, 30(24) :3476–3483.
Cette référence a été citée 1 fois à la page 40.

[Marschall et al., 2018] Marschall, T., Marz, M., Abeel, T., Dijkstra, L., Dutilh, B. E., Ghaffaari, A., Kersey, P., Kloosterman, W. P., Mäkinen, V., Novak, A. M., Paten,

B., Porubsky, D., Rivals, E., Alkan, C., Baaijens, J. A., De Bakker, P. I., Boeva, V., Bonnal, R. J., Chiaromonte, F., Chikhi, R., Ciccarelli, F. D., Cijvat, R., Datema, E., Van Duijn, C. M., Eichler, E. E., Ernst, C., Eskin, E., Garrison, E., El-Kebir, M., Klau, G. W., Korbel, J. O., Lameijer, E. W., Langmead, B., Martin, M., Medvedev, P., Mu, J. C., Neerincx, P., Ouwens, K., Peterlongo, P., Pisanti, N., Rahmann, S., Raphael, B., Reinert, K., de Ridder, D., de Ridder, J., Schlesner, M., Schulz-Trieglaff, O., Sanders, A. D., Sheikhizadeh, S., Shneider, C., Smit, S., Valenzuela, D., Wang, J., Wessels, L., Zhang, Y., Guryev, V., Vandin, F., Ye, K., and Schönhuth, A. (2018).
Computational pan-genomics : Status, promises and challenges.
Briefings in Bioinformatics, 19(1) :118–135.
Cette référence a été citée 2 fois aux pages 37 et 38.

[Matsumoto et al., 2005] Matsumoto, T., Wu, J., Kanamori, H., Katayose, Y., Fujisawa, M., Namiki, N., Mizuno, H., Yamamoto, K., Antonio, B. A., Baba, T., Sakata, K., Nagamura, Y., Aoki, H., Arikawa, K., Arita, K., Bito, T., Chiden, Y., Fujitsuka, N., Fukunaka, R., Hamada, M., Harada, C., Hayashi, A., Hijishita, S., Honda, M., Hosokawa, S., Ichikawa, Y., Idonuma, A., Iijima, M., Ikeda, M., Ikeno, M., Ito, K., Ito, S., Ito, T., Ito, Y., Ito, Y., Iwabuchi, A., Kamiya, K., Karasawa, W., Kurita, K., Katagiri, S., Kikuta, A., Kobayashi, H., Kobayashi, N., MacHita, K., Maehara, T., Masukawa, M., Mizubayashi, T., Mukai, Y., Nagasaki, H., Nagata, Y., Naito, S., Nakashima, M., Nakama, Y., Nakamichi, Y., Nakamura, M., Meguro, A., Negishi, M., Ohta, I., Ohta, T., Okamoto, M., Ono, N., Saji, S., Sakaguchi, M., Sakai, K., Shibata, M., Shimokawa, T., Song, J., Takazaki, Y., Terasawa, K., Tsugane, M., Tsuji, K., Ueda, S., Waki, K., Yamagata, H., Yamamoto, M., Yamamoto, S., Yamane, H., Yoshiki, S., Yoshihara, R., Yukawa, K., Zhong, H., Yano, M., Sasaki, T., Yuan, Q., Ouyang, S., Liu, J., Jones, K. M., Gansberger, K., Moffat, K., Hill, J., Bera, J., Fadrosh, D., Jin, S., Johri, S., Kim, M., Overton, L., Reardon, M., Tsitrin, T., Vuong, H., Weaver, B., Ciecko, A., Tallon, L., Jackson, J., Pai, G., Van Aken, S., Utterback, T., Reidmuller, S., Feldblyum, T., Hsiao, J., Zismann, V., Iobst, S., De Vazeille, A. R., Buell, C. R., Ying, K., Li, Y., Lu, T., Huang, Y., Zhao, Q., Feng, Q., Zhang, L., Zhu, J., Weng, Q., Mu, J., Lu, Y., Fan, D., Liu, Y., Guan, J., Zhang, Y., Yu, S., Liu, X., Zhang, Y., Hong, G., Han, B., Choisne, N., Demange, N., Orjeda, G., Samain, S., Cattolico, L., Pelletier, E., Couloux, A., Segurens, B., Wincker, P., D'Hont, A., Scarpelli, C., Weissenbach, J., Salanoubat, M., Quetier, F., Yu, Y., Kim, H. R., Rambo, T., Currie, J., Collura, K., Luo, M., Yang, T. J., Ammiraju, J. S., Engler, F., Soderlund, C., Wing, R. A., Palmer, L. E., De La Bastide, M., Spiegel, L., Nascimento, L., Zutavern, T., O'Shaughnessy, A., Dike, S., Dedhia, N., Preston, R., Balija, V., McCombie, W. R., Chow, T. Y., Chen, H. H., Chung, M. C., Chen, C. S., Shaw, J. F., Wu, H. P., Hsiao, K. J., Chao, Y. T., Chu, M. K., Cheng, C. H., Hour, A. L., Lee, P. F., Lin, S. J., Lin, Y. C., Liou, J. Y., Liu, S. M., Hsing, Y. I., Raghuvanshi, S., Mohanty, A., Bharti, A. K., Gaur, A., Gupta, V., Kumar, D., Ravi, V., Vij, S., Kapur, A., Khurana, P., Khurana, P., Khurana, J. P., Tyagi, A. K., Gaikwad, K., Singh, A., Dalal, V., Srivastava, S., Dixit, A., Pal, A. K., Ghazi, I. A., Yadav, M., Pandit, A., Bhargava, A., Sureshbabu, K., Batra, K., Sharma, T. R., Mohapatra, T., Singh, N. K., Messing, J., Nelson, A. B., Fuks, G., Kavchok, S., Keizer, G., Llaca, E. L. V., Song, R., Tanyolac, B., Young, S., Ho, K., Hahn, J. H., Sangsakoo, G., Vanavichit, A., De Mattos, L. A. T., Zimmer, P. D., Malone, G., Dellagostin, O., De Oliveira, A. C., Bevan, M., Bancroft, I., Minx, P., Cordum, H., Wilson, R., Cheng, Z., Jin, W., Jiang, J., Leong, S. A., Iwama, H., Gojobori, T., Itoh, T., Niimura, Y., Fujii, Y., Habara, T., Sakai, H., Sato, Y., Wilson, G., Kumar, K., McCouch, S., Juretic, N., Hoen, D., Wright, S., Bruskiewich, R., Bureau, T., Miyao, A., Hirochika, H., Nishikawa, T., Kadowaki,

K. I., Sugiura, M., and Burr, B. (2005).
The map-based sequence of the rice genome.
Nature, 436(7052) :793–800.
Cette référence a été citée 1 fois à la page 14.

[Metzker, 2010] Metzker, M. L. (2010).
Sequencing technologies — the next generation.
Nature Reviews Genetics, 11(1) :31–46.
Cette référence a été citée 1 fois à la page 18.

[Minkin et al., 2016] Minkin, I., Pham, S., and Medvedev, P. (2016).
TwoPaCo : An efficient algorithm to build the compacted de Bruijn graph from many complete genomes.
pages 1–9.
Cette référence a été citée 1 fois à la page 40.

[Myint et al., 2012] Myint, K., Arikit, S., Wanchana, S., Yoshihashi, T., Choowongkomon, K., and Vanavichit, A. (2012).
A pcr-based marker for a locus conferring the aroma in myanmar rice (oryza sativa l.).
TAG. Theoretical and applied genetics. Theoretische und angewandte Genetik, 125 :887–96.
Cette référence a été citée 1 fois à la page 15.

[Noé and Kucherov, 2005] Noé, L. and Kucherov, G. (2005).
YASS : enhancing the sensitivity of DNA similarity search.
33(suppl_2) :W540–W543.
Cette référence a été citée 1 fois à la page 73.

[O'Leary et al., 2016] O'Leary, N. A., Wright, M. W., Brister, J. R., Ciufo, S., Haddad, D., McVeigh, R., Rajput, B., Robbertse, B., Smith-White, B., Ako-Adjei, D., Astashyn, A., Badretdin, A., Bao, Y., Blinkova, O., Brover, V., Chetvernin, V., Choi, J., Cox, E., Ermolaeva, O., Farrell, C. M., Goldfarb, T., Gupta, T., Haft, D., Hatcher, E., Hlavina, W., Joardar, V. S., Kodali, V. K., Li, W., Maglott, D., Masterson, P., McGarvey, K. M., Murphy, M. R., O'Neill, K., Pujar, S., Rangwala, S. H., Rausch, D., Riddick, L. D., Schoch, C., Shkeda, A., Storz, S. S., Sun, H., Thibaud-Nissen, F., Tolstoy, I., Tully, R. E., Vatsan, A. R., Wallin, C., Webb, D., Wu, W., Landrum, M. J., Kimchi, A., Tatusova, T., DiCuccio, M., Kitts, P., Murphy, T. D., and Pruitt, K. D. (2016).
Reference sequence (refseq) database at ncbi : current status, taxonomic expansion, and functional annotation.
Nucleic Acids Res, 44(D1) :D733–45.
Cette référence a été citée 1 fois à la page 36.

[Organick, 1975] Organick, E. I. (1975).
Editor's Overview.
ACM Computing Surveys (CSUR), 7(1) :1.
Cette référence a été citée 1 fois à la page 19.

[Park et al., 2009] Park, G., Hwang, H.-K., Nicodème, P., and Szpankowski, W. (2009).
Profiles of Tries.
SIAM Journal on Computing, 38(5) :1821–1880.
Cette référence a été citée 1 fois à la page 66.

[Peter et al., 2018] Peter, J., De Chiara, M., Friedrich, A., Yue, J. X., Pflieger, D., Bergstrom, A., Sigwalt, A., Barre, B., Freel, K., Llored, A., Cruaud, C., Labadie, K., Aury, J. M., Istace, B., Lebrigand, K., Barbry, P., Engelen, S., Lemainque, A., Wincker, P., Liti, G., and Schacherer, J. (2018).

Genome evolution across 1,011 saccharomyces cerevisiae isolates.
Nature, 556(7701) :339–344.
Cette référence a été citée 1 fois à la page 36.

[Philippe et al., 2011] Philippe, N., Salson, M., Lecroq, T., Léonard, M., Commes, T., and Rivals, E. (2011).
Querying large read collections in main memory : a versatile data structure.
BMC Bioinformatics, 12(1) :242.
Cette référence a été citée 4 fois aux pages 22, 24, 59 et 115.

[Pickett et al.,] Pickett, B. D., Miller, J. B., and Ridge, P. G.
Cette référence a été citée 1 fois à la page 18.

[Rizk et al., 2013] Rizk, G., Lavenier, D., and Chikhi, R. (2013).
DSK : k-mer counting with very low memory usage.
Bioinformatics, 29(5) :652–653.
Cette référence a été citée 1 fois à la page 19.

[Rosone and Sciortino, 2013] Rosone, G. and Sciortino, M. (2013).
The Burrows-Wheeler Transform between Data Compression and Combinatorics on Words, pages 353–364.
Springer Berlin Heidelberg, Berlin, Heidelberg.
Cette référence a été citée 2 fois aux pages 31 et 39.

[Roy et al., 2014] Roy, R. S., Bhattacharya, D., and Schliep, A. (2014).
Turtle : Identifying frequent k -mers with cache-efficient algorithms.
Bioinformatics, 30(14) :1950–1957.
Cette référence a été citée 1 fois à la page 19.

[Sacomoto et al., 2012] Sacomoto, G. A. T., Kielbassa, J., Chikhi, R., Uricaru, R., Antoniou, P., Sagot, M.-F., Peterlongo, P., and Lacroix, V. (2012).
KISSPLICE : de-novo calling alternative splicing events from RNA-seq data.
BMC bioinformatics, 13 Suppl 6(6) :S5.
Cette référence a été citée 1 fois à la page 18.

[Sanger et al., 1977] Sanger, F., Air, G. M., Barrell, B. G., Brown, N. L., Coulson, A. R., Fiddes, J. C., Hutchison, C. A., Slocombe, P. M., and Smith, M. (1977).
Nucleotide sequence of bacteriophage ϕx174 DNA.
Nature, 265(5596) :687–695.
Cette référence a été citée 1 fois à la page 11.

[Schatz et al., 2014] Schatz, M. C., Maron, L. G., Stein, J. C., Hernandez Wences, A., Gurtowski, J., Biggers, E., Lee, H., Kramer, M., Antoniou, E., Ghiban, E., Wright, M. H., Chia, J. M., Ware, D., McCouch, S. R., and McCombie, W. R. (2014).
Whole genome de novo assemblies of three divergent strains of rice, oryza sativa, document novel gene space of aus and indica.
Genome Biol, 15(11) :506.
Cette référence a été citée 1 fois à la page 36.

[Schneeberger et al., 2009] Schneeberger, K., Hagmann, J., Ossowski, S., Warthmann, N., Gesing, S., Kohlbacher, O., and Weigel, D. (2009).
Simultaneous alignment of short reads against multiple genomes.
Genome biology, 10(9) :R98.
Cette référence a été citée 1 fois à la page 38.

[Sevim et al., 2019] Sevim, V., Lee, J., Egan, R., Clum, A., Hundley, H., Lee, J., Everroad, R. C., Detweiler, A. M., Bebout, B. M., Pett-Ridge, J., Göker, M., Murray, A. E.,

Lindemann, S. R., Klenk, H. P., O'Malley, R., Zane, M., Cheng, J. F., Copeland, A., Daum, C., Singer, E., and Woyke, T. (2019).
Shotgun metagenome data of a defined mock community using Oxford Nanopore, PacBio and Illumina technologies.
Scientific data, 6(1) :285.
Cette référence a été citée 1 fois à la page 12.

[Shannon, 1948] Shannon, C. E. (1948).
A Mathematical Theory of Communication.
The Bell System Technical Journal, 27 :379–423, 623–656.
Cette référence a été citée 2 fois aux pages 25 et 49.

[Sherman et al., 2018] Sherman, R. M., Forman, J., Antonescu, V., Puiu, D., Daya, M., Rafaels, N., Boorgula, M. P., Chavan, S., Vergara, C., Ortega, V. E., Levin, A. M., Eng, C., Yazdanbakhsh, M., Wilson, J. G., Marrugo, J., Lange, L. A., Williams, L. K., Watson, H., Ware, L. B., Olopade, C. O., Olopade, O., Oliveira, R. R., Ober, C., Nicolae, D. L., Meyers, D. A., Mayorga, A., Knight-Madden, J., Hartert, T., Hansel, N. N., Foreman, M. G., Ford, J. G., Faruque, M. U., Dunston, G. M., Caraballo, L., Burchard, E. G., Bleecker, E. R., Araujo, M. I., Herrera-Paz, E. F., Campbell, M., Foster, C., Taub, M. A., Beaty, T. H., Ruczinski, I., Mathias, R. A., Barnes, K. C., and Salzberg, S. L. (2018).
Assembly of a pan-genome from deep sequencing of 910 humans of african descent.
Nat Genet.
Cette référence a été citée 1 fois à la page 36.

[Sichtig et al., 2019] Sichtig, H., Minogue, T., Yan, Y., Stefan, C., Hall, A., Tallon, L., Sadzewicz, L., Nadendla, S., Klimke, W., Hatcher, E., Shumway, M., Aldea, D. L., Allen, J., Koehler, J., Slezak, T., Lovell, S., Schoepp, R., and Scherf, U. (2019).
Fda-argos is a database with public quality-controlled reference genomes for diagnostic use and regulatory science.
Nat Commun, 10(1) :3313.
Cette référence a été citée 1 fois à la page 36.

[Sijmons et al., 1991] Sijmons, P. C., Grundler, F. M., von Mende, N., Burrows, P. R., and Wyss, U. (1991).
Arabidopsis thaliana as a new model host for plant-parasitic nematodes.
The Plant Journal, 1(2) :245–254.
Cette référence a été citée 1 fois à la page 15.

[Simpson et al., 2009] Simpson, J. T., Wong, K., Jackman, S. D., Schein, J. E., Jones, S. J., and Birol, I. (2009).
ABySS : A parallel assembler for short read sequence data.
Genome Research, 19(6) :1117–1123.
Cette référence a été citée 1 fois à la page 17.

[Sirén, 2009] Sirén, J. (2009).
Compressed suffix arrays for massive data.
In *Proceedings of the 16th International Symposium on String Processing and Information Retrieval*, SPIRE '09, pages 63–74, Berlin, Heidelberg. Springer-Verlag.
Cette référence a été citée 2 fois aux pages 31 et 40.

[Sirén, 2016] Sirén, J. (2016).
Burrows-wheeler transform for terabases.
In *2016 Data Compression Conference (DCC)*, pages 211–220.
Cette référence a été citée 1 fois à la page 40.

[Snipen and Ussery, 2010] Snipen, L. and Ussery, D. W. (2010).
Standard operating procedure for computing pangenome trees.
Standards in Genomic Sciences, 2(1) :135–141.
Cette référence a été citée 1 fois à la page 37.

[Stein et al., 2018] Stein, J. C., Yu, Y., Copetti, D., Zwickl, D. J., Zhang, L., Zhang, C., Chougule, K., Gao, D., Iwata, A., Goicoechea, J. L., Wei, S., Wang, J., Liao, Y., Wang, M., Jacquemin, J., Becker, C., Kudrna, D., Zhang, J., Londono, C. E. M., Song, X., Lee, S., Sanchez, P., Zuccolo, A., Ammiraju, J. S. S., Talag, J., Danowitz, A., Rivera, L. F., Gschwend, A. R., Noutsos, C., Wu, C. C., Kao, S. M., Zeng, J. W., Wei, F. J., Zhao, Q., Feng, Q., El Baidouri, M., Carpentier, M. C., Lasserre, E., Cooke, R., Rosa Farias, D. D., da Maia, L. C., Dos Santos, R. S., Nyberg, K. G., McNally, K. L., Mauleon, R., Alexandrov, N., Schmutz, J., Flowers, D., Fan, C., Weigel, D., Jena, K. K., Wicker, T., Chen, M., Han, B., Henry, R., Hsing, Y. C., Kurata, N., de Oliveira, A. C., Panaud, O., Jackson, S. A., Machado, C. A., Sanderson, M. J., Long, M., Ware, D., and Wing, R. A. (2018).
Genomes of 13 domesticated and wild rice relatives highlight genetic conservation, turnover and innovation across the genus oryza.
Nat Genet, 50(2) :285–296.
Cette référence a été citée 1 fois à la page 36.

[Tettelin et al., 2005] Tettelin, H., Masignani, V., Cieslewicz, M. J., Donati, C., Medini, D., Ward, N. L., Angiuoli, S. V., Crabtree, J., Jones, A. L., Durkin, A. S., DeBoy, R. T., Davidsen, T. M., Mora, M., Scarselli, M., Margarit Y Ros, I., Peterson, J. D., Hauser, C. R., Sundaram, J. P., Nelson, W. C., Madupu, R., Brinkac, L. M., Dodson, R. J., Rosovitz, M. J., Sullivan, S. A., Daugherty, S. C., Haft, D. H., Selengut, J., Gwinn, M. L., Zhou, L., Zafar, N., Khouri, H., Radune, D., Dimitrov, G., Watkins, K., O'Connor, K. J., Smith, S., Utterback, T. R., White, O., Rubens, C. E., Grandi, G., Madoff, L. C., Kasper, D. L., Telford, J. L., Wessels, M. R., Rappuoli, R., and Fraser, C. M. (2005).
Genome analysis of multiple pathogenic isolates of Streptococcus agalactiae : Implications for the microbial "pan-genome".
Proceedings of the National Academy of Sciences of the United States of America, 102(39) :13950–13955.
Cette référence a été citée 1 fois à la page 36.

[The 1000 Genomes Project Consortium, 2015] The 1000 Genomes Project Consortium (2015).
A global reference for human genetic variation.
Nature, 526(7571) :68–74.
Auton, A., Brooks, L. D., Durbin, R. M., Garrison, E. P., Kang, H. M., Korbel, J. O., Marchini, J. L., McCarthy, S., McVean, G. A., and Abecasis, G. R.
Cette référence a été citée 1 fois à la page 36.

[The 1001 Genomes Consortium, 2016] The 1001 Genomes Consortium (2016).
1,135 genomes reveal the global pattern of polymorphism in *Arabidopsis thaliana*.
Cell, 166(2) :481–491.
Alonso-Blanco, C., Andrade, J., Becker, C., Bemm, F., Bergelson, J., Borgwardt, K. M., Cao, J., Chae, E., Dezwaan, T. M., Ding, W., Ecker, J. R., Exposito-Alonso, M., Farlow, A., Fitz, J., Gan, X., Grimm, D. G., Hancock, A. M., Henz, S. R., Holm, S., Horton, M., Jarsulic, M., Kerstetter, R. A., Korte, A., Korte, P., Lanz, C., Lee, C.-R., Meng, D., Michael, T. P., Mott, R., Muliyati, N. W., Nägele, T., Nagler, M., Nizhynska, V., Nordborg, M., Novikova, P. Y., Picó, F.-X., Platzer, A., Rabanal, F. A., Rodriguez, A.,

Rowan, B. A., Salomé, P. A., Schmid, K. J., Schmitz, R. J., Seren, Ü., Sperone, F. G., Sudkamp, M., Svardal, H., Tanzer, M. M., Todd, D., Volchenboum, S. L., Wang, C., Wang, G., Wang, X., Weckwerth, W., Weigel, D., and Zhou, X.
Cette référence a été citée 1 fois à la page 36.

[Thomas, 1971] Thomas, C. A. (1971).
The genetic organization of chromosomes.
Annual Review of Genetics, 5(1) :237–256.
PMID : 16097657.
Cette référence a été citée 1 fois à la page 8.

[Valenzuela, 2016] Valenzuela, D. (2016).
Chico : A compressed hybrid index for repetitive collections.
In Goldberg, A. V. and Kulikov, A. S., editors, *Experimental Algorithms*, pages 326–338, Cham. Springer International Publishing.
Cette référence a été citée 3 fois aux pages 31, 40 et 97.

[Weiner, 1973] Weiner, P. (1973).
Linear pattern matching algorithms.
In *Proceedings of the 14th Annual Symposium on Switching and Automata Theory (Swat 1973)*, SWAT '73, pages 1–11, Washington, DC, USA. IEEE Computer Society.
Cette référence a été citée 1 fois à la page 29.

[Wu et al., 2014] Wu, G. A., Prochnik, S., Jenkins, J., Salse, J., Hellsten, U., Murat, F., Perrier, X., Ruiz, M., Scalabrin, S., Terol, J., Takita, M. A., Labadie, K., Poulain, J., Couloux, A., Jabbari, K., Cattonaro, F., Del Fabbro, C., Pinosio, S., Zuccolo, A., Chapman, J., Grimwood, J., Tadeo, F. R., Estornell, L. H., Munoz-Sanz, J. V., Ibanez, V., Herrero-Ortega, A., Aleza, P., Perez-Perez, J., Ramon, D., Brunel, D., Luro, F., Chen, C., Farmerie, W. G., Desany, B., Kodira, C., Mohiuddin, M., Harkins, T., Fredrikson, K., Burns, P., Lomsadze, A., Borodovsky, M., Reforgiato, G., Freitas-Astua, J., Quetier, F., Navarro, L., Roose, M., Wincker, P., Schmutz, J., Morgante, M., Machado, M. A., Talon, M., Jaillon, O., Ollitrault, P., Gmitter, F., and Rokhsar, D. (2014).
Sequencing of diverse mandarin, pummelo and orange genomes reveals complex history of admixture during citrus domestication.
Nat Biotechnol, 32(7) :656–62.
Cette référence a été citée 1 fois à la page xii.

[Yu et al., 2019] Yu, J., Golicz, A. A., Lu, K., Dossa, K., Zhang, Y., Chen, J., Wang, L., You, J., Fan, D., Edwards, D., and Zhang, X. (2019).
Insight into the evolution and functional characteristics of the pan-genome assembly from sesame landraces and modern cultivars.
Plant Biotechnol J, 17(5) :881–892.
Cette référence a été citée 1 fois à la page 37.

[Zhang et al., 2014] Zhang, Q., Pell, J., Canino-Koning, R., Howe, A. C., and Brown, C. T. (2014).
These Are Not the K-mers You Are Looking For : Efficient Online K-mer Counting Using a Probabilistic Data Structure.
PLoS ONE, 9(7) :e101271.
Cette référence a été citée 1 fois à la page 20.

[Zhao et al., 2018] Zhao, Q., Feng, Q., Lu, H., Li, Y., Wang, A., Tian, Q., Zhan, Q., Lu, Y., Zhang, L., Huang, T., Wang, Y., Fan, D., Zhao, Y., Wang, Z., Zhou, C., Chen, J., Zhu, C., Li, W., Weng, Q., Xu, Q., Wang, Z. X., Wei, X., Han, B., and Huang, X. (2018).
Pan-genome analysis highlights the extent of genomic variation in cultivated and wild rice.

Nat Genet, 50(2) :278–284.
Cette référence a été citée 16 fois aux pages 15, 48, 49, 51, 96, 97, 98, 100, 104, 105, 110, 111, 112, 145, 146 et 147.

[Ziv and Lempel, 1978] Ziv, J. and Lempel, A. (1978).
Compression of individual sequences via variable-rate coding.
IEEE Transactions on Information Theory, 24(5) :530–536.
Cette référence a été citée 2 fois aux pages 25 et 31.

Liste des tableaux

— *Corps du document* —

— *Annexes* —

— *Pages annexées* —

Table des figures

— Corps du document —

— *Annexes* —

— *Pages annexées* —

Liste des codes et algorithmes

— *Corps du document* —

— *Annexes* —

— *Pages annexées* —

Table des matières

— Pages liminaires —

— Corps du document —

Étude des différentes approches d'indexation : du génome au pan-génome

Application aux génomes de riz

Clément Agret

Résumé

Le sujet de ma thèse concerne l'étude des structures d'indexation et les méthodes de compression de données pour palier au problème d'indexation d'une collection de génomes similaires.
Le but ultime est d'appliquer ces méthodes à l'indexation des génomes du riz afin de faciliter l'analyse de l'ensemble des études et activités du projet *GenomeHarvest*. Comme par exemple l'impact de leurs variations structurelles sur les taux de recombinaison, les études des fréquences allèliques, les études GWAS, etc.
L'indexation de génomes complets est une étape importante dans l'exploration et la compréhension des données d'organismes vivants. Un index devrait fournir une réponse rapide aux questions suivantes : - Combien de fois un motif donné apparaît dans une collection de génomes ?
- Quelles sont les positions et les génomes porteurs d'un motif donné ?
- Quelle est la longueur du motif à la position i pour un génome donné ?
C'est ce que nous avons réalisé grâce à cette thèse, nous avons réfléchi, proposés un algorithme. Une fois l'algorithme trouvé, nous avons implémenté cet algorithme sous forme d'un logiciel qui permet l'indexation d'une large collection de génome et son requêtage.

Mots-clés : Mots clés

Different indexing approaches study: From Genome to Pan-genome

Application to rice genome

Abstract

The subject of my thesis concerns the study of indexing structures and data compression methods to find a solution to the problem of indexing a collection of similar genomes.
The ultimate goal is to apply these methods to the indexing of rice genomes in order to facilitate the analysis of all the studies and activities of the GenomeHarvest project. As for example the impact of their structural variations on the rates of recombination, studies of allelic frequencies, GWAS studies, etc.
The indexing of whole genomes is an important step in the exploration and understanding of data from living organisms. An index should provide a quick answer to the following questions: - How often does a given motif appear in a collection of genomes?
- What are the positions and genomes carrying a given motif?
- What is the length of the motif at position i for a given genome?
This is what we achieved thanks to this thesis, we thought and finally proposed an algorithm. When the algorithm was found, we implemented this one into a software that allows the indexing of a large genome collection and its querying.

Keywords: keywords

Classification acm

Catégories et descripteurs de sujets : E.1 [**Data**]: DATA STRUCTURES; F.2.2 [**Theory of Computation**]: ANALYSIS OF ALGORITHMS AND PROBLEM COMPLEXITY—*Nonnumerical Algorithms and Problems*; G.2.1 [**Mathematics of Computing**]: DISCRETE MATHEMATICS—*Combinatorics*; G.3 [**Mathematics of Computing**]: PROBABILITY AND STATISTICS; H.1.1 [**Information Systems**]: MODELS AND PRINCIPLES—*Systems and Information Theory*; H.3.1 [**Information Systems**]: INFORMATION STORAGE AND RETRIEVAL—*Content Analysis and Indexing*; H.3.3 [**Information Systems**]: INFORMATION STORAGE AND RETRIEVAL—*Information Search and Retrieval*; H.3.4 [**Information Systems**]: INFORMATION STORAGE AND RETRIEVAL—*Systems and Software*; I.5 [**Computing Methodologies**]: PATTERN RECOGNITION.

Termes généraux : Algorithms, Experimentation, Measurement, Performance, Standardization, Theory.

Discipline : BIDAP: Biologie Intégrative, Diversité et Amélioration des Plantes
Spécialité : Génétique et amélioration des plantes

LIRMM (équipe MAB)/CIRAD (équipe ID)
TA A-108/03, Avenue Agropolis, 34398 Montpellier
Cedex 5, France