Dynamic Pan-genome Graphs

Khodor Hannoush

COLLEGE MATHS, TELECOMS
DOCTORAL INFORMATIQUE, SIGNAL
BRETAGNE SYSTEMES, ELECTRONIQUE

Université de Rennes

THÈSE DE DOCTORAT DE

L'UNIVERSITÉ DE RENNES

ÉCOLE DOCTORALE N° 601
Mathématiques, Télécommunications, Informatique, Signal, Systèmes, Électronique
Spécialité : *Informatique*

Par

Khodor Hannoush

Dynamic Pan-genome Graphs

Thèse présentée et soutenue à Rennes, le 13/12/2024
Unité de recherche : Équipe GenScale, Univ Rennes, Inria, CNRS, IRISA

Rapporteurs avant soutenance :

Guillaume Blin	Professeur des universités, Université de Bordeaux
Segolene Caboche	Ingénieur de recherche, Université de Lille

Composition du Jury :

Présidente :	Claire Lemaitre	Directrice de recherche, Univ Rennes, INRIA, Rennes
Examinateurs :	Guillaume Blin	Professeur des universités, Université de Bordeaux
	Segolene Caboche	Ingénieur de recherche - HdR, Université de Lille
	Raluca Uricaru	Maitresse de conférence, Université de Bordeaux
	Claire Lemaitre	Directrice de recherche, Univ Rennes, INRIA, Rennes
Dir. de thèse :	Pierre Peterlongo	Directeur de recherche, Univ Rennes, INRIA, Rennes
Co-encadrante. de thèse :	Camille Marchet	Chargée de recherche CNRS, CRIStAL, Lille

Whenever I set myself to the task of learning, I realize how little I know and the more I learn, the more I realize how ignorant I am.

Abu Abdullah Muhammad bin Idris ash-Shafii

to my father Salim and my brother Mahmoud who left me so early
to my mother, my brothers and my sisters
to my life partner Sabah
to all innocent people around the world

Acknowledgement

The Prophet **Muhammad**, peace and blessings be upon him, said, **"Whoever does not thank people will not thank Allah"**. So I would like to thank all the people who made the work of this thesis possible.

First and foremost, I want to thank my PhD supervisors **Pierre Peterlongo** and **Camille Marchet**, who guided me along this fruitful journey. I believe that I have never met amazing mentors like them. Your invaluable advice will remain part of my personality. I am really grateful for your support during the difficult times I went through. Without your guidance and patience, it would have been an impossible goal.

I would like to thank my doctoral advisory committee members **Christine Gaspin** and **Karel Brinda** for the scientific discussions we had during the annual advisory meetings. Your suggestions were extremely useful. Your support as well will never be forgotten.

I want to thank Guillaume Blin and Ségolène Caboche for accepting to be reviewers of my thesis. I would also like to thank Claire Lemaitre and Raluca Uricaru for being part of my PhD defense jury. I am grateful for the time and energy you allocated to this thesis.

I want to thank all the members of the Symbiose team and the members of the Bonsai team. I really enjoyed the moments I spent with you during these three years.

I would like to thank my office mates Nicolas Guillaudeux, Victor Epain, Arya Kaul, and Meven Mognol for the pleasurable moments we spent together. I also want to thank Francesca Brunetti for the fun times we shared and for her support.

Special thanks to Kevin De Silva, Teo Lemane, Florent Leray, and Lucas Robidou for their technical support. I also want to thank Marie Le Roïc from the Genscale team for managing all the administrative requirements during these three years.

I would like to thank all my colleagues in the ITN Alpaca project for the valuable workshops we held together. I also want to thank all the members of this training network for managing such a big and great project. Special thanks to the people of Marie Skłodowska-Curie Actions for funding this project, part of which is this thesis.

I would like to take the opportunity to thank my internship supervisor **Fariza Tahi**, who introduced me to the field of computational biology. I am really grateful for the

significant efforts you made to bring me to France to complete my studies. I also thank Louis Becquey, Eric Angel, and Blaise Hanczar for their guidance during my internship.

I would like to thank all my professors at the American University of Beirut (AUB) for introducing me to the domain of computer science. Special thanks to Professors Wassim El Hajj and Haidar Safa for their support and guidance during my journey at AUB.

Last but not least, I want to thank my mother, Hayat. Words cannot express the efforts and sacrifices you have made for me. I want to thank all my brothers, especially Ali, and my sisters for their support. I also thank my life partner, Sabah Hamieh, for her support.

Finally, I want to thank all my friends in France and in Lebanon, especially Othman Chamaa, for their support and encouragement.

Résumé en français

Introduction

L'ADN, molécule fondamentale de l'information génétique, est composé de quatre nucléotides, formant une structure double brin. Grâce aux techniques de séquençage, il est possible de lire et d'analyser ces séquences pour mieux comprendre l'évolution et le fonctionnement des organismes vivants. Les avancées en séquençage ont entraîné une augmentation massive des données génomiques, nécessitant de nouvelles méthodes pour stocker et analyser ces informations efficacement.

Le concept de pan-génome a été introduit pour mieux représenter l'ensemble des variations génétiques au sein d'une espèce. La première définition du pan-génome se concentre sur les familles de gènes, divisant le génome en une partie centrale (gènes partagés par tous les individus) et une partie accessoire (gènes spécifiques à certains sous-groupes). La deuxième définition, dans laquelle s'inscrit cette thèse, considère le pan-génome comme l'union des séquences d'ADN des différentes souches d'une espèce. Cette approche permet d'analyser non seulement la présence ou l'absence de gènes, mais aussi les variations au niveau des séquences telles que les SNPs et les insertions/délétions.

Une des manières de représenter ces séquences est le graphe de de Bruijn, qui permet de modéliser les relations entre les k-mères (sous-séquences d'ADN de tailles k) au sein du pan-génome. Cette structure de données est centrale pour traiter efficacement les grandes quantités de séquences générées par les technologies de séquençage. La thèse se concentre sur le développement de méthodes efficaces pour la mise à jour de ces graphes, afin d'intégrer les nouvelles données sans reconstruction complète, tout en optimisant les ressources utilisées.

État de l'art

Les progrès des technologies de séquençage génomique ont produit une explosion de données biologiques, nécessitant des outils efficaces pour les stocker et les analyser. Pour traiter ces volumes de séquences, les graphes de de Bruijn sont devenus indispensables,

car ils permettent de représenter les données de séquençage de manière compacte et exploitable. Cette approche réduit la redondance des séquences et améliore considérablement l'efficacité des méthodes bioinformatiques.

Un graphe de de Bruijn est une structure où les nœuds représentent des k-mères (sous-séquences de longueur k) et les arêtes correspondent à des chevauchements de longueur $k-1$ entre ces k-mères. Cette représentation permet d'exprimer des relations entre de nombreuses séquences, tout en évitant les duplications. Le graphe de de Bruijn compacté est défini en allant plus loin et en fusionnant les chemins maximaux non ramifiés en unitigs, réduisant ainsi la taille globale du graphe et la mémoire nécessaire à son stockage.

Diverses méthodes ont été développées pour construire efficacement des graphes de de Bruijn et leurs variantes compactées. Des outils comme BOSS [14], Minia [20] et FDBG [27] permettent de construire des graphes de de Bruijn. D'autres, tels que BCALM [22], Cuttlefish [53] et GGCAT [1], se concentrent sur la construction de graphes de de Bruijn compactés.

Cependant, l'une des limites de ces méthodes réside dans leur nature statique. Lorsque de nouvelles séquences génomiques doivent être intégrées, il est nécessaire de reconstruire complètement le graphe, une opération coûteuse qui implique des calculs redondants. Les méthodes dynamiques telles que DynamicBOSS [4], Bifrost [43] et BuffBOSS [2] permettent les mises à jour de graphes, mais ils effectuent toujours un travail redondant ou nécessitent une reconstruction complète après plusieurs modifications, ce qui a un impact sur leur efficacité globale.

Enfin, il devient crucial de développer des méthodes qui ciblent spécifiquement les régions du cDBG qui nécessitent une mise à jour, évitant ainsi la reconstruction complète du graphe et réduisant les calculs redondants. Ces approches optimiseraient la gestion des mises à jour en ne modifiant que les parties pertinentes, ce qui optimise le temps de calcul. Cela permettrait une gestion plus souple et plus rapide des séquences génomiques, répondant ainsi aux besoins croissants générés par l'augmentation des données de séquençage.

La mise à jour d'un graphe de de Bruijn compacté

La nécessité de mettre à jour les graphes de de Bruijn compactés (cDBG) lors de l'ajout de nouvelles séquences, telles que des génomes entiers, a incité le développement de nouvelles méthodes capables de gérer cette tâche de manière efficace. Cdbgtricks, le travail de cette thèse, est une nouvelle méthode, conçue pour éviter la nécessité d'une reconstruc-

tion complète du graphe à chaque mise à jour. Traditionnellement, l'ajout de nouveaux génomes à un cDBG implique la reconstruction du graphe entier, ce qui représente une surcharge de calcul. Cdbgtricks simplifie ce problème en se concentrant sur l'identification de l'ensemble des nouveaux k-mères et en identifiant uniquement les changements spécifiques à apporter au graphe existant, ce qui permet de maintenir l'intégrité du graphe tout en minimisant les recalculs inutiles.

Cdbgtricks est basé sur le traitement des k-mères existants dans le graphe et des k-mères nouvellement ajoutés comme des ensembles distincts. Cette stratégie permet d'isoler les changements et d'optimiser la gestion des ajouts. Cdbgtricks utilise des fonctions de hachage parfaites minimales (MPHF) et une approche de partitionnement des k-mères pour cibler uniquement les parties du graphe affectées par les nouveaux ajouts. En partitionnant les k-mères en plusieurs compartiments, où chaque compartiment est indexé indépendamment par une MPHF dédiée, il est possible de réindexer uniquement les compartiments affectés lorsque de nouveaux k-mères sont ajoutés, évitant ainsi une refonte complète de la structure.

Cdbgtricks est capable d'identifier les changements dans le graphe. En effet, lorsque de nouveaux k-mères sont ajoutés, certains nœuds du graphe, appelés unitigs, peuvent être affectés, soit en se séparant, soit en fusionnant avec d'autres. Cdbgtricks est capable d'identifier précisément ces changements grâce à son index et de mettre à jour le graphe en conséquence.

Cdbgtricks est également capable de mettre à jour l'index du graphe qui pourrait être utilisé pour le requêter. Le code source ouvert de Cdbgtricks est disponible, et les expériences sont reproductibles (voir `https://github.com/khodor14/Cdbgtricks`).

Requête sur un graphe de de Bruijn compacté

de La recherche dans un graphe de de Bruijn compacté (cDBG) est utilisée dans diverses applications biologiques. Le type de requête à effectuer dépend largement de l'application biologique en question. Certaines méthodes répondent aux appartenances des k-mères de la requête, d'autres effectuent un mapping de la requête sur le graphe. Il existe différentes méthodes dans l'état actuel de la technique, telles que BLight [68], SSHash [79], Bifrost et GGCAT [1], pour n'en citer que quelques-unes.

La contribution de cette thèse a été étendue pour permettre l'interrogation d'un graphe de de Bruijn compacté. La fonctionnalité d'interrogation a été ajoutée à Cdbgtricks.

Cdbgtricks offre deux types de requêtes. La première permet de rapporter la présence ou l'absence de la requête, et la seconde rapporte l'emplacement des k-mères de la requête dans le graphe.

Les expériences comparatives menées avec Cdbgtricks ont montré que, bien qu'il offre des performances inférieures aux outils de pointe tels que Bifrost [43], GGCAT [1] et SSHash [79] en termes de rapidité d'exécution, il garantit néanmoins des temps de réponse raisonnables.

Conclusion et perspectives

La thèse apporte une contribution dans le domaine de la mise à jour et de l'interrogation des graphes de de Bruijn compactés, en optimisant ces processus pour répondre aux besoins croissants de l'analyse génomique. La méthode développée permet d'intégrer de nouvelles séquences sans reconstruction complète des graphes, tout en optimisant le temps de calcul. Ces avancées répondent aux défis posés par l'explosion des données de séquençage.

À court terme, plusieurs axes de recherche sont envisagés. Tout d'abord, le développement de méthodes pour stocker et mettre à jour efficacement les références auxquelles appartiennent les k-mères (couleurs). Deuxièmement, l'estimation de la distance entre les séquences et le graphe ou entre deux graphes. Cela nous permettra de mieux évaluer la similarité des nouvelles séquences avec le graphe existant ou entre deux graphes. Un autre objectif est d'optimiser le partitionnement des k-mères de manière à ce que les k-mers consécutifs soient trouvés dans les mêmes compartiment afin de minimiser les ressources utilisées. Enfin, le développement d'une fonction de hachage parfaite minimale (MPHF) est essentiel pour permettre des mises à jour rapides de la MPHF sans reconstruction.

Les perspectives à long terme comprennent la mise en œuvre d'une méthode de mise à jour d'un graphe de de Bruijn compact construit sur un alphabet plus large que l'alphabet classique. Finalement, une autre direction consiste à développer un algorithme de partitionnement d'un graphe de de Bruijn compacté afin de réduire les besoins en mémoire lors des mises à jour ou des requêtes, en permettant le traitement indépendant des sous-graphes et en facilitant le calcul en parallèle.

Table of Contents

Chapter 1

Introduction

This chapter establishes the biological context and computational view of genomic data that will be explored in the following chapters.

1.1 Biological background

This section aims to provide some biological background related to this thesis. Section 1.1.1 will provide some biological background that will help define the problem tackled in this thesis. Section 1.1.2 will provide the methods used to access these data from the organisims. Finally, section 1.1.3 provides the datasets used in the experiments of this thesis and some statistics about the used samples.

1.1.1 Preliminaries

Deoxyribonucleic acid (DNA) forms the genetic basis of life. It is a molecule that encodes the genetic instructions essential for the development, functioning, growth, and reproduction of all known living organisms. The structure of the DNA was introduced by Watson and Crick [96], but it was discovered in 1869 by Friedrich Miescher [28]. DNA is a macro-molecule composed of two strands of polynucleotides coiled around each other to form a double helix (figure 1.1). Each strand is a chain of nucleotides. A nucleotide is an organic molecule composed of a nitrogenous base, a pentose sugar and a phosphate group. There are four nucleotides in the DNA: the adenine (A), the cytosine (C), the guanine (G) and the thymine (T). The nucleotides are bounded to one another through the phosphodiester linkage between the sugar of the first and the phosphate of the next. The two strands are bounded together according to the base pairing rules (A with T and C with G) through a hydrogen bound.

The DNA sequence encodes the genetic information of an organism. There are two types of organisms: the prokaryotes and the eukaryotes. The prokaryotes such as bacteria and

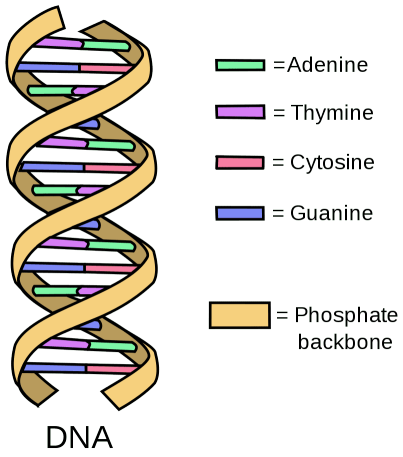

Figure 1.1 – Representation of a DNA molecule. The figure is taken from `https://commons.wikimedia.org/wiki/File:DNA_Visual_Representation_of_Double-Helix.png`

archaea store their DNA in the cytoplasm. On the other hand, the eukaryotes such as animals and plants store most of their DNA in the cell nucleus. The DNA can be linear or circular depending on the organism, and it is organized in the chromosomes. Each chromosome has a set of genes which are sequences arranged one after another. A gene is a nucleotide sequence within DNA that undergoes transcription to generate a functional ribonucleic acid. The complete set of the DNA of an organism constitutes its genome which is a sequence of the bases A, C, G and T.

An organism is made up of cells. The number of cells differs from one organism to another. Each cell contains a copy of the genome of the organism. A genome may contain one or more copies of the chromosomes. A genome that contains one copy is called a haploid genome, while a genome that contains two copies of the chromosomes is called a diploid genome. A genome may contain more than two copies of the chromosomes and is called in such a case a polyploidy genome.

1.1.2 Accessing the DNA sequences

To analyse the genetic content, biologists need to be able to read the DNA. This is done through a mechanism called DNA sequencing. Sequencing is the process of frag-

menting the DNA, and then obtaining the textual sequences of these fragments using a machine called the sequencer. The sequencing methods have evolved over time. Here we will discuss briefly the three generations of sequencing.

First generation sequencers In 1977 Frederick Sanger developed the chain terminated method which was named the *Sanger sequencing* [89]. The process is called sequencing by synthesis, involves synthesizing DNA from a template using a primer and denatured nucleotides and a chain-terminating dideoxynucleotides (ddNTPs). These ddNTPs terminate the elongation of the DNA strand, creating a collection of fragments of various lengths, which are then separated by gel electrophoresis. Each nucleotide addition is marked along the gel, allowing the sequence to be reconstructed by identifying the bands corresponding to each nucleotide. Sanger method was among other methods used in sequencing the first human mitochondrial DNA and the complete human genome.
However, sequencing methods do not always correctly capture all nucleotides. Sequences with some incorrectly captured nucleotides are referred to as erroneous sequences. The percentage of incorrectly sequenced nucleotides is called the error rate. Despite achieving an error rate ranging from 0.001% to 1% [95], and being relied upon for sequencing regions that are difficult to read with modern technologies, the Sanger method is limited by low throughput and high costs.

Next generation sequencing The introduction of second-generation sequencing technologies in the early 2000s, known as Next Generation Sequencing (NGS) [92] or High Throughput Sequencing (HTS), marked a significant leap forward in genomic research. These methods enabled millions of parallel sequencing reactions, drastically increasing the speed and efficiency of DNA sequencing while reducing costs exponentially compared to Sanger method. Illumina and Ion Torrent are two prominent examples of NGS platforms that work on the principle of sequencing by synthesis (SBS), where DNA fragments are amplified and sequenced on solid surfaces using fluorescent or pH-based detection methods. These technologies produce short reads, typically ranging from 100 to 400 nucleotides, with error rates between 0.1% to 1%, enabling stakeholders to sequence vast quantities of genetic material quickly and accurately.
The main limitation of NGS is the short size of reads which is 150 nucleotides and it rarely exceeds 250 nucleotides. Genomes often contain repetitive DNA sequences. One of the drawbacks of short read is difficulty to accurately align the reads to the repeated

regions. This difficulty was tackled by introducing paired-end sequencing which allows sequencing both ends of a DNA fragment.

Third generation sequencing Third Generation Sequencing (TGS) [90] technologies introduced to the field of genomic sequencing both the Single Molecule Real Time (SMRT) and Nanopore sequencing. These technologies enable the sequencing of longer DNA fragments without the need for polymerase chain reaction (PCR) amplification, resulting in fewer but substantially longer reads compared to previous technologies. Two TGS technologies were proposed : Pacific Biosciences (PacBio) and Oxford Nanopore Technologies (ONT). The average read length of PacBio is 10 thousand bases and can reach up to 60 thousand bases, while the average read length of ONT is 10 thousand bases and can exceed 4 million bases.

Initially, both technologies started with high error rates of up to 13%, however some advancements have drastically reduced this rate to less than 1%. Despite challenges such as initial error rates and variability in base-calling accuracy, TGS technologies continue to evolve rapidly, leading to a drastic increase in the genomic databases while decreasing the sequencing cost.

1.1.3 Genomes

Genome and genetic information

A genome is the complete set of nucleotides (A, C, G, and T in DNA) that forms all the chromosomes of an individual or a species. It contains the instructions for building and maintaining an organism. Within the genome, specific sequences of DNA called genes encode the information necessary to produce proteins, which perform most of the essential functions in a cell.

Genomes used in this thesis

In this thesis, we used two datasets to explore genomic diversity. The first dataset includes 15,806 *Escherichia coli* (*E. coli*) genomes. *E. coli* has a relatively small circular genome, averaging 4 million bases in size. Despite being smaller in size, the *E. coli* genomes exhibit significant diversity, with notable variations across strains. A figure of *E. coli* is shown in Figure 1.2. Currently, the National Center for Biotechnology Information

(NCBI) database hosts 285, 114 *E. coli* genome samples [1].

The second dataset consists of 100 human genomes. In contrast to *E. coli*, the human genome is substantially larger, spanning approximately 3 billion bases. However, human genomes are strikingly similar, with over 99.9% of the sequence being identical across individuals. The NCBI database currently contains 1,582 human genome samples [2]. These datasets provide contrasting perspectives on genomic variation, with E. coli highlighting extensive diversity within a species and the human genome demonstrating remarkable uniformity across individuals.

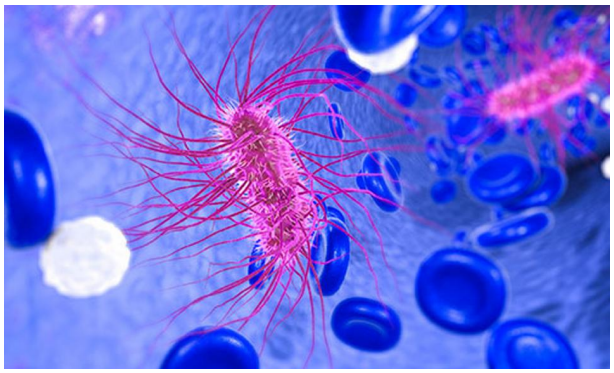

Figure 1.2 – *E. coli* bacteria. The figure is taken from `https://www.fda.gov/news-events/public-health-focus/e-coli-and-foodborne-illness`

1.1.4 Pan-genomes

The term *pan-genome* was defined first by Medini et al. in 2005 [71]. The prefix *pan* is derived from a greek word which means "whole" or "everything". In the study of Medini et al [71], a pan-genome is defined as the global gene repertoire and is composed of two parts: the core genome and the accessory genome.

The core genome is the set of genes that are shared by all strains, a sub type of a biological species [40]. While housekeeping genes, that encode essential functions for the survival of all the species, are a part of the core genome, but not all core genes are categorized as housekeeping genes. For example, the core genome of the *Streptococcus agalactiae* encodes its pathogenicity through genes involved in neonatal infection [71].

On the other hand, the accessory genome is the set of genes that are not found in all the

1. https://www.ncbi.nlm.nih.gov/search/all/?term=Escherichia+coli
2. https://www.ncbi.nlm.nih.gov/search/all/?term=human

strains. Accessory genes encode functions that are not essential for the survival of the species. They account for the diversity of the species, and they enable them to adapt to various environmental changes.
In their studies of the *Streptococcus agalactiae* pan-genome, Medini et al. modeled the growth of the pan-genome as a function of the number of species analyzed. The aim is to determine the number of strains that characterize the core and accessory genomes of the species. The Heap's law was used to model the growth of the pan-genome. More specifically, the Heap's law in the context of pan-genome growth is given as:

$$N = kn^{-\alpha} \tag{1.1}$$

where N is the number of gene families and k and α are determined experimentally. The value of α characterizes the openness of the pan-genome. An open pan-genome (figure 1.3.b) is characterized by the continuous discovery of new gene families as new genomes are analyzed (figure 1.3.c). In Heap's law, an open pan-genome is determined by $\alpha > 1$. An open pan-genome implies that the species is highly adaptable to environmental changes. In contrast, a closed pan-genome (figure 1.3.a) is one in which nearly all genes have been discovered after sequencing a relatively small number of genomes (figure 1.3.c).

Another definition of pan-genome has aroused in recent years to relate the pan-genome to the DNA sequences of the genome rather than the gene families. In this context, a pan-genome is defined as the union of the sequences of the strains [10]. This sequence-based approach provides a more detailed understanding of genomic variation, capturing not just the presence or absence of genes, but also sequence-level differences, such as single-nucleotide polymorphisms (SNPs), insertions, deletions, and structural variations. In recent years, several data structures have been proposed to construct and analyze pan-genomes at the sequence level [78, 6, 43]. The work of this thesis lies within the scope of sequence based pan-genomics. Sequence-based pan-genomics faces challenges related to the large volume of data, the computational complexity of constructing and analyzing pan-genomes, and the need for efficient indexing and querying methods.

1.2 Some computational view of genomic data

In this section, we will go over some computational background and how DNA sequences are stored in the computer before they are processed or stored in data structures.

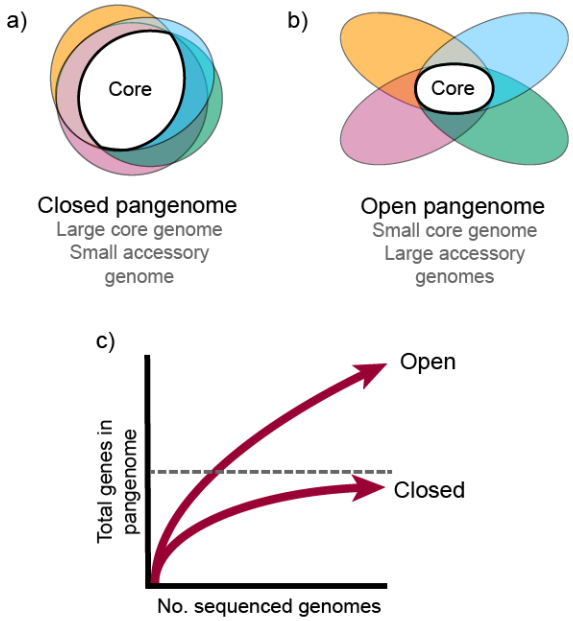

Figure 1.3 – **(a)** closed pangenome. **(b)** an open pangenome. **(c)** the growth of the size of an open and a closed pangenome. The figure is taken from `https://en.wikipedia.org/wiki/Pan-genome`.

1.2.1 Strings

In the theory of computation, an alphabet is a finite set of symbols denoted by σ. An alphabet is used to construct a language. For example $\sigma = \{A, B, ..., Y, Z, a, b, ..., y, z\}$ is the alphabet of the English language. A string s is a sequence of characters drawn from an alphabet e.g., $s =$"genome". A string s has a length denoted by $|s|$ which is equal to the number of characters in the string. Denote by $s[i]$ the character of s at the position i where $0 \leq i < |s|$ e.g. $s[0] = g$ and $s[5] = e$ when $s =$"genome". A substring of s is a string that starts at i and ends at j where $0 \leq i \leq j < |s|$. For example the string *eno* is a substring of the string *genome*, it starts at $i = 1$ and ends at $j = 3$. A prefix of a string s is a substring of s that starts at $i = 0$ and ends at $0 \leq j < |s|$ e.g., *gen* is a prefix of *genome*. A suffix of a string s is a substring of s that starts at $0 \leq i < |s|$ and ends at $j = |s| - 1$ e.g., *nome* is a suffix of *genome*. The concatenation of two strings x and y is done by appending the characters of y to the right of x, and is denoted by $concat(x, y) = x \odot y$ where $\odot$ is the concatenation operator. The reverse of a string s is string s' which constructed by spelling s from right to left, i.e., $s' = s[|s| - 1] \odot s[|s| - 2] \odot ... \odot s[1] \odot s[0]$.

1.2.2 DNA as a text

DNA sequences are accessed through the sequencers. The sequencers provide the DNA as signals which are converted to text through a process called base calling. DNA sequences are stored in a computer in a text format. From a computational view, a DNA sequence is a string. The alphabet for the DNA language is $\sigma = \{A, C, G, T\}$ which are the first letters of the four nucleotides names Adenine, Cytosine, Guanine and Thymine. All the properties of a regular string that were introduced in section 1.2.1 hold for the DNA strings. As introduced in section 1.1.1, a DNA sequence is double stranded. Each strand is the reverse complement of the other. The reverse complement of a DNA string s is the complement of the reversed string of s. To find the complement of each character in a DNA string, we use the base-pairing rules: Adenine (A) pairs with Thymine (T), and Cytosine (C) pairs with Guanine (G). For example, to find the reverse complement of $s =$ ACGGT, first we need to reverse s so we get TGGCA. Finally, we complement each base in the reversed string to get ACCGT.

DNA sequences can be stored in different file formats. The common file formats used in computational biology analysis are Fasta and Fastq. A Fasta file can store the

DNA reads or the sequences of the whole genome. It provides for every sequence some additional information represented as a string in the header of this sequence which begins by a special character " > ". On the other hand a Fastq file provides the reads with some differences in the header of the sequence. The header begins by a special character "@", and it contains as well the quality of every character which simply represents the confidence in each base of the sequence. Figure 1.4 shows an example of a Fasta and a Fastq files.

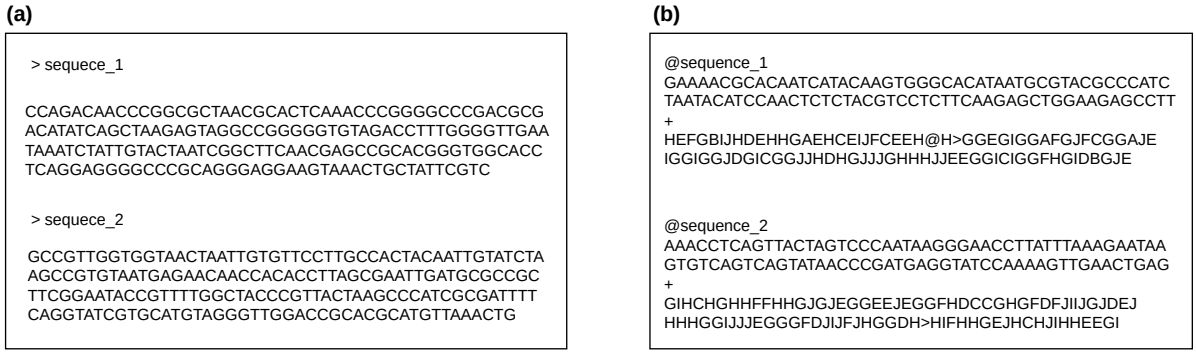

Figure 1.4 – **(a)** an example of a Fasta file with two sequences. **(b)** an example of a Fastq file with two sequences.

1.2.3 Notions of Graph

A graph is a data structure that consists of a set of nodes (also called vertices) and a set of edges (or links) that connect pairs of nodes. A graph is a pair of two sets $G = (V, E)$ where V is an unordered set of vertices and $E \subseteq \{(u, v) \mid u,\ v \in V \text{ and } u \neq v\}$ is an unordered set of edges. A graph can be directed or undirected. In a directed graph, each edge $e = (u, v) \in E$ where $u,\ v\ \in V$ links the two vertices u and v in an asymmetrical way, i.e., v can be directly reached from u while u cannot be directly reached from v. However, in an undirected graph an edge $e = (u, v) \in E$ links u and v in a symmetrical way i.e., v can be directly reached from u and u can be directly reached from v (figure 1.5). A path p in a graph is an ordered set of nodes $p = \{u_1, u_2, .., u_{|p|}\}$ where every two consecutive nodes are connected by an edge $e = (u_i, u_{i+1}) \in E$ for $1 \leq i < |p|$, and $|p|$ is the number of nodes in p. A node u is said to be an in-neighbor (or a predecessor) of a node v if $(u, v) \in E$. Conversely, a node u is said to be an out-neighbor (or a successor) of a

node v if $(v, u) \in E$. Every node in a graph has an in-degree and an out-degree. The in-degree (respectively out-degree) of a node u is the number of in-neighbor (respectively out-neighbor) nodes of u. The degree of a node is the sum of the in-degree and the out-degree. Directed graphs are used for example to store a collection of genomes in a de Bruijn graph that will be introduced in chapter in 2.

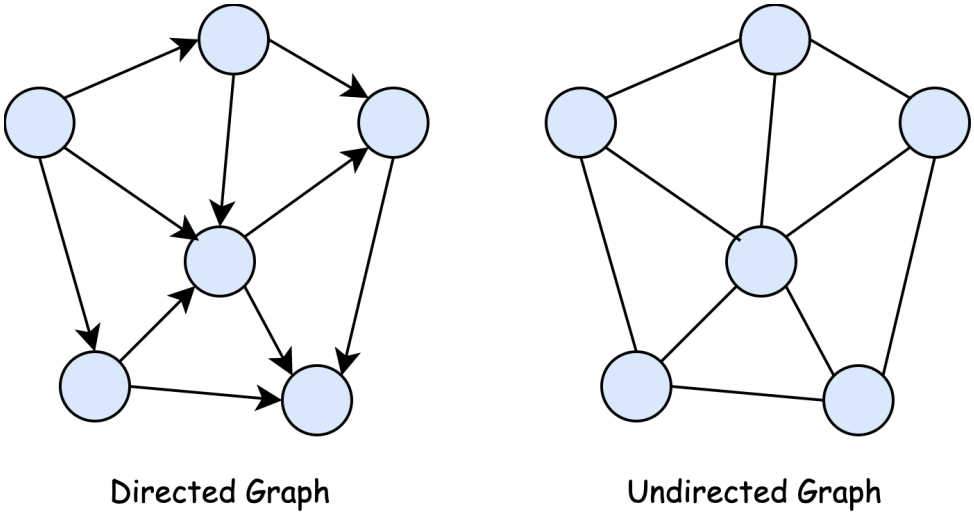

Figure 1.5 – Directed versus undirected graph. The figure is taken from `https://cs226fa21.github.io/notes/26-graph/step05.html`

1.3 Thesis objective

The advancement in sequencing technology have decreased the sequencing cost (Figure 1.6), and provided reads with longer lengths. This advancement made the sequencing of millions of samples affordable and have lead in turn to an exponential increase of the size of genomic databases (Figure 1.7). The National Center for Biotechnology Information (NCBI) has moved the sequence read archive (SRA) database to the cloud. This database stores now more than 55 petabytes of data.
Given this exponential growth, there is a high need for developing methods that efficiently store and analyse large datasets. The research community have responded to this need by providing dozens of methods able to efficiently store large datasets of genomic data such as Bifrost [43], GGCAT [1] and Fulgor [36] to name a few. Yet, this growth which points to a high speed generation of DNA sequences open the need for not only to provide

efficient storage mechanisms, but also to allow the addition (mutability) of new sequences to structures. In other words, there is a need to allow for the addition of new sequenced data to a de Bruijn graph data structure (see section 2.2 in chapter 2), that stores DNA sequences, in a memory- and time-efficient manner.
Pan-genomes are essential for understanding genetic variability across strains or individuals. Updating pan-genomes as new sequences are generated requires methods capable of handling the high similarity within species efficiently. The genomic similarity suggests that adding new sequences often affects only a small portion of the existing data structure, making a complete rebuild unnecessary in many cases.
This thesis aims at developing novel methods to update in an efficient way a data structure for storing DNA sequences called the de Bruijn graph that will be introduced in chapter 2.

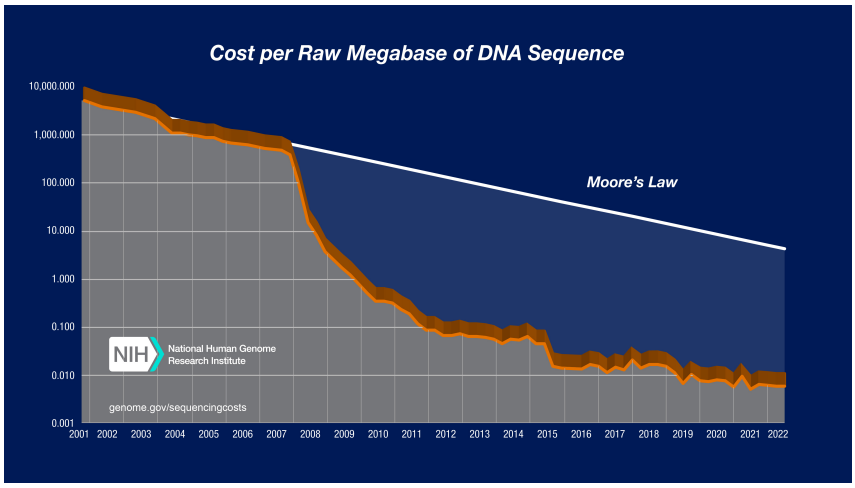

Figure 1.6 – Evolution of the sequencing cost. The figure is taken from `https://www.genome.gov/about-genomics/fact-sheets/DNA-Sequencing-Costs-Data`

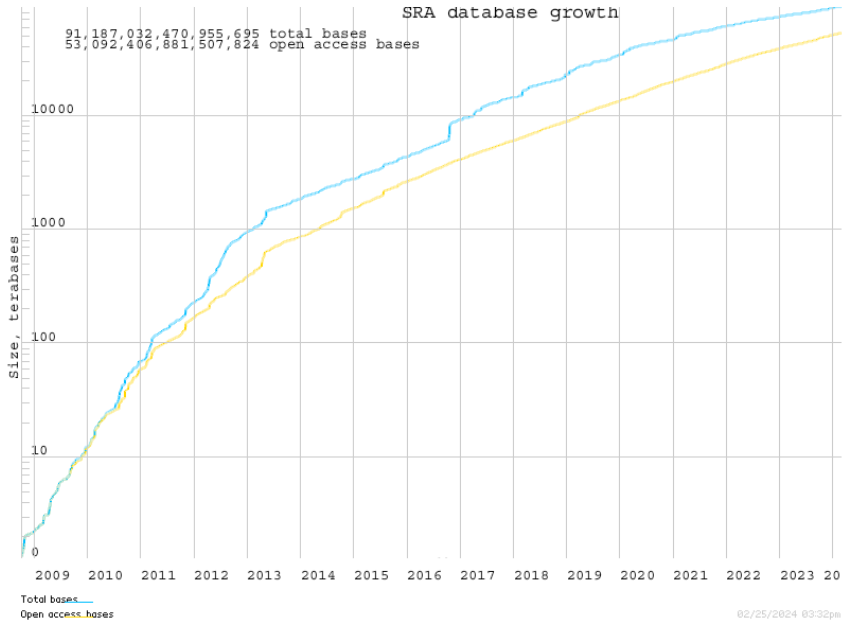

Figure 1.7 – The number of available bases in the sequence read archive database. The figure is taken from `https://www.ncbi.nlm.nih.gov/sra/docs/sragrowth/`

Chapter 2

State of the Art

This chapter reviews the state of the art in genomic data analysis and storage, focusing on the use of de Bruijn graphs to handle the vast amounts of sequencing data generated by modern technologies. It explores various approaches for constructing and updating a de Bruijn graphs, highlighting the limitations of static methods and the need for more efficient strategies to integrate new genomic sequences without full reconstruction of the graph.

2.1 Fundamental concepts and data structures

To grasp the developed methods that deals with the de Bruijn graph, it is essential to first introduce the fundamental data structures such as hash functions, hash tables and Bloom filters used in these methods. These concepts are crucial to efficiently store and retrieve genomic data to and from the data structures that uses them.

2.1.1 Hash functions

A hash function f is a mathematical function that associates to a key x, an integer value $v = f(x)$. There are three types of hashing: universal, perfect and minimal perfect hashing. A universal hashing aims to map N keys to the range of hash values $[0; R]$ in a uniform way where $R > N$ is the largest hash value that can be assigned to a key, meaning each key has an equal probability of being assigned to any hash value within the range. However, this uniform distribution does not prevent collisions, where different keys may get the same hash value. To resolve collision, various techniques can be used such as chaining or open addressing [24]. In the case of chaining, the keys that get the same hash value are stored together in a vector, linked list or a binary search tree. In open addressing, if a key x gets a hash value v that already assigned to another key, then x gets the first unused hash value according to the used open addressing technique. Although

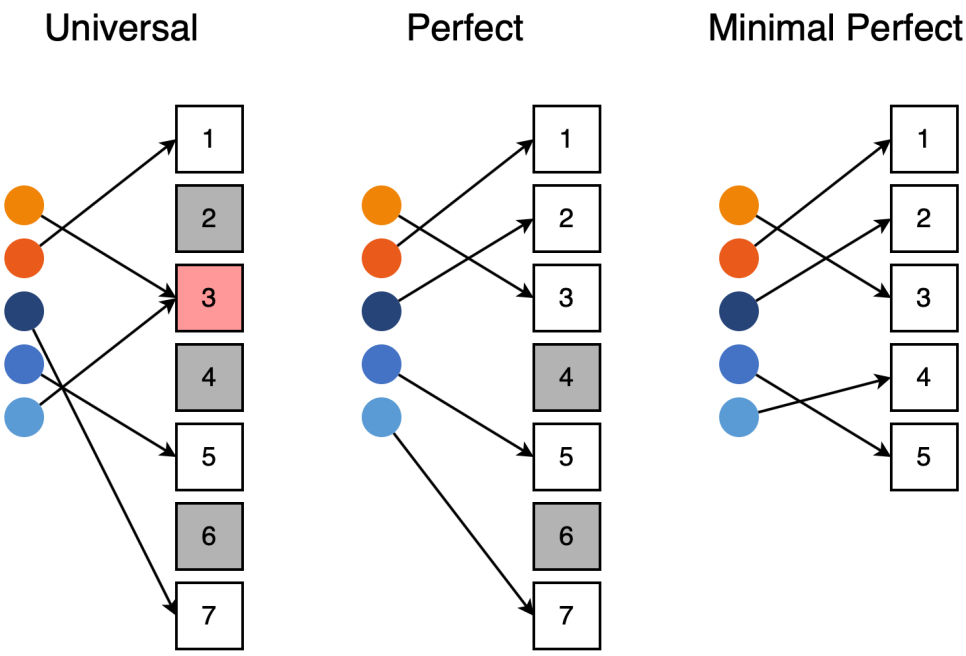

Figure 2.1 – **Difference between universal, perfect and minimal perfect hashing** The number of keys is $N = 5$. The range for the universal and perfect hashing is $R = 7$. The range for the minimal perfect hashing is $R = N = 5$.The figure is taken from `https://randorithms.com/2019/09/12/MPH-functions.html`

these two techniques resolve the collision problem, they reduce the access time. Another way to resolve collision is by using perfect hashing. In perfect hashing every key gets a unique hash value in the range $[0; R]$ where $R > N$. However, perfect hashing results in empty slots in the range $[0; R]$, i.e. at least one value in this range is not assigned to a given key. To handle this issue, minimal perfect hash functions (MPHF) can be used. Minimal perfect hash functions guarantee that collisions do not occur and that the size of the set of values is the same as the size of the set of keys, i.e., $R = N$. An illustration of these hashing techniques is presented in figure 2.1.

2.1.2 Hash Tables

A hash table is a dictionary which is also known to be one of "key-value" data structure. The hash table maps a value to every key. A hash function is used to achieve this mapping.

2.1.3 Bloom filters

A Bloom filter [13] is a space- and time-efficient data structure used to approximate the membership of a query in a dataset. One or more hash functions are used to compute a Bloom filter from a set of keys. A Bloom filter B is composed of a binary array A of size m which is an array that stores 0s or 1s. Initially, the values in a Bloom filter are set to 0. Inserting a key x into the Bloom filter is done as follows:

$$insert(x) : A[f_i(x)] = 1 \ for \ all \ i = 1, ..., r \tag{2.1}$$

where r is the number of hash functions and f_i is the i-th hash function.
To test whether or not an element x was inserted into the Bloom filter, we proceed as follows:

$$MayContain(x) = A[f_1(x)] \wedge A[f_2(x)] \wedge ... \wedge A[f_r(x)] \tag{2.2}$$

where $\wedge$ denotes the logical and operator. If $MayContain$ is evaluated to 0 then x does not belong to the dataset from the Bloom filter is constructed, whereas if it is evaluated to 1 then x may or may not belong to the dataset. When the $MayContain(x)$ evaluates to 1 but x is not in the dataset, then a false positive is generated (figure 2.2). This happens when the values in the Bloom filter corresponding to x where set to 1 for other k-mers different than x.
Given a set S of n elements to be inserted into the Bloom filter, then the probability of erroneously saying that an element x belongs to S is:

$$p = (1 - e^{-\frac{rn}{m}})^r \tag{2.3}$$

This suggests that the probability of false positives decreases if the size of the Bloom filter m increases.
That optimal number of hash functions that minimizes the probability of false positives is given as:

$$r = \frac{m}{n} \ln 2 \tag{2.4}$$

where ln is the natural logarithmic function.
Given a desired probability p of false positives, the size of the Bloom filter is then given

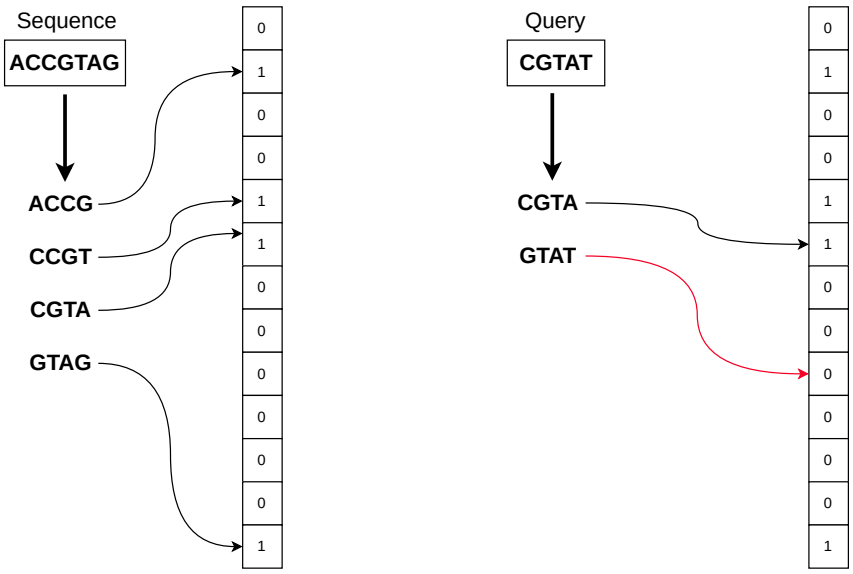

Figure 2.2 – Inserting and querying squences to and from a Bloom filter. The figure is adapted from `https://digitalbunker.dev/understanding-the-inner-workings-of-Bloom-filters/`

as:

$$m = -n\frac{\ln p}{(\ln 2)^2} \tag{2.5}$$

For example, if we want to insert one million elements to a Bloom filter with a desired probability of false positives $p = 0.01$, then the optimal size of the Bloom filter is $m = 9,585,059$ and the optimal number of hash functions is $r = 7$.

A blocked Bloom filter (BBF) [83] is an array of Bloom filters. An additional hash function h is used to determine in which Bloom filter to insert or locate an element.

2.1.4 General definitions

Definition 1 ***k-mer****: A k-mer is a string of k characters drawn from an alphabet* Σ *(figure 2.3).*

Definition 2 ***Canonical k-mer****: The canonical k-mer of a k-mer x is the lexicographically smallest k-mer between x and its reverse complement. For example the canonical k-mer of ACGGT is ACCGT.*

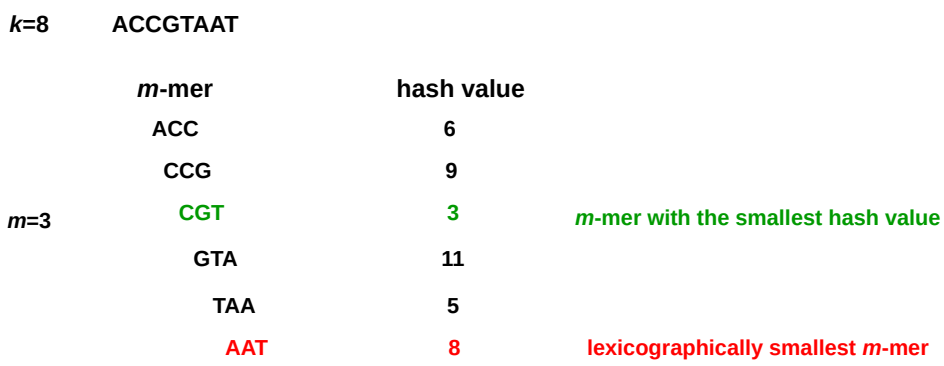

Figure 2.3 – An example of a k-mer of length 8, a lexicographic based minimizer (red) and a hash based minimizer where (green) $m = 3$.

Definition 3 ***Minimizer****: A minimizer of a string s is a substring q of fixed length m where $m < |s|$ and q is the smallest m-mer of s with respect to some order such as the lexicographic order or the hash based order (figure 2.3). .*

Definition 4 ***Left minimizer****: The left minimizer of a k-mer u is the minimizer of size $0 < l < k - 1$ of the $(k - 1)$-mer prefix of u.*

Definition 5 ***Right minimizer****: The right minimizer of a k-mer u is the minimizer of size $0 < l < k - 1$ of the $(k - 1)$-mer suffix of u.*

Definition 6 ***Super-k-mer****: A super-k-mer is a sequence whose k-mers share the same minimizer.*

2.2 The de Bruijn Graph

The de Bruijn Graph is a directed graph representing overlaps between sequences of symbols. It was named after Nicolaas Govert de Bruijn who introduced it in his paper on the combinatorial problems [16]. The original proposition was used for the binary sequences. It has been widely used in diverse fields including, but not limited to, information theory [18] and network design [52]. The de Bruijn graph is one of the fundamental data structures that play crucial roles in computational biology. It is tremendously used in various applications, including but not limited to genome assembly [72, 91], read error

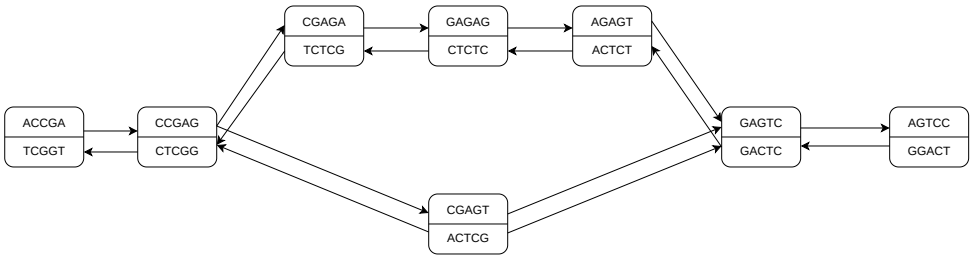

Figure 2.4 – A node centric de Bruijn graph. Nodes represent k-mers of length 5 and edges connect consecutive nodes if they share an overlap of length 4. The reverse complement of complement of every k-mer is shown in the node. The edges connects nodes in the forward and the backward directions.

correction [59, 46], read alignment [64, 5] and read abundance queries [69]. There are two approaches of defining a de Bruijn graph: the node centric approach and the edge centric approach. In both approaches, the information stored in the graph are drawn from an alphabet σ of m symbols.

2.2.1 Node centric de Bruijn Graph

In a node centric approach, the nodes represents words of length k called k-mers, while the edges represent an overlap of length $k-1$ between two k-mers. The nodes represent as well the reverse complement of the nodes. An edge between two nodes u and v indicates that the last $k-1$ symbols in u are the same as the first $k-1$ symbols in v. An illustration of a node centric de Bruijn graph is provided in figure 2.4.

2.2.2 Edge centric

In an edge centric approach, the edges are words of length $k+1$, while nodes are words of length k. An edge e connects two nodes u and v if the symbols in u are the first k symbols in e, and the symbols in v are the last k symbols in v (figure 2.5).

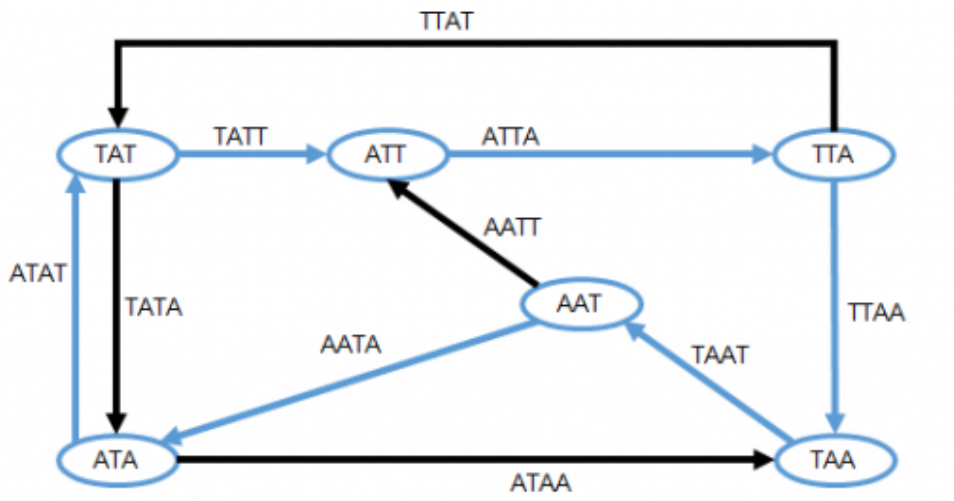

Figure 2.5 – An edge centric de Bruijn graph. Edges represent k-mer of length 4, and nodes represent $(k-1)$-mers. The figure is taken from `https://medvedevgroup.com/2020/08/21/what-do-eulerian-and-hamiltonian-cycles-have-to-do-with-genome-assembly/`.

2.3 Methods to construct the de Bruijn graph

2.3.1 The de Bruijn graph as a set of k-mers

The de Bruijn graph offers the storage of k-mers and the navigation from one k-mer to another. A set of k-mers can be considered as a de Bruijn graph since the nodes can be retrieved from these k-mers.

k-mer counting methods

Counting the frequency of k-mers is a crucial step in building a de Bruijn graph. It is the starting point of several tools as we will discuss later. The number of occurrences of every k-mer in the dataset, also referred as its frequency, cannot be known before processing the dataset. Thus, the aim of counting methods is to determine the frequency of all the k-mers in the dataset. This process is particularly useful for filtering out low-frequent k-mers from the dataset that is composed of sequenced reads, because they are considered to contain sequencing errors. Sequencing errors often result in the generation of low-frequency k-mers due to issues like incorrect base calls during sequencing or am-

plification.
Several k-mer counting methods were proposed such as Jellyfish [67], KMC [29, 30, 55] and DSK [85]. They differ by their choices of handling the datasets on disk or on the RAM and by their optimization techniques. Although it is not designed only for counting k-mers, kmtricks [56] can provide k-mer counting functionalities.

Exact representation of a set of k-mers

CBL [70] is data structure to represent a set of k-mers. It extends the work of Conway and Bromage [23] by adding smallest cyclic rotation of k-mers along with a dynamic bit vectors. CBL is an exact representation of k-mers, i.e., it does not generate neither false positives nor false negative. The dynamic bit vector allows the additions and deletion of k-mers from the structure which makes CBL support k-mer set operations (intersection, union, and difference).

2.3.2 Full representation of dBG (nodes and edges)

Minia [20] a method that builds a de Bruijn graph from a set of short reads. Minia starts first with a k-mer counting step using DSK [85] and filters out low-abundance k-mers which considered to contain sequencing errors. The remaining k-mers are stored in a Bloom filter, which can generate false positives. To afford for an exact representation, Minia stores some other k-mers, that do not belong to the read set, in a separate hash table. These k-mers are reffered as critical false positives. They are the connection of true k-mers, but but they are considered as false positives. To navigate from one k-mer x to another, the Bloom filter is checked for the successors of x. If the Bloom filter indicates that some of these successors are presents, the hash table is checked to validate whether or not they are true or false positives. Minia was implemented to assemble genomes from short reads throught the construction of contigoues sequences by traversing the de Bruijn graph. It reduced the memory footprint with respect to other assemblers. Nevertheless, it requires double access for each k-mer, one to the Bloom filter and another one to the hash table. Note that Minia is not able to check false positives k-mers that do not have true successors or predecessors in the graph.

BOSS [14] provides an implementation of the Burrows Wheleer Transform [17] of the de Bruijn graph. Instead of stroring each k-mer separately which is a memory-intensive

task, BOSS stores the last character of every k-mer, a sequence of k characters. To access a k-mer, a backward traversal on the data structure is performed using Rank and Select. Rank(c,i) returns the number of characters c up to the position i, while Select(c,i) determines the position of the character c whose rank is i. While BOSS provides a memory-efficient representation of a de Bruijn graph as it uses 4 $bits/k-mer$, it needs to perform k access to the memory to locate a k-mer.

FDBG [27] is an implementation of Belazzougui et al. [11] data structure to construct a de Bruijn graph. The idea in FDBG is to decrease the number of access to the data structure compared to BOSS structure while preserving exact representation. FDBG first computes a minimal perfect hash function f over the set of distinct k-mers N of the input sequence. The predecessors (respectively successors) are stored in a bit array IN (respectively OUT). A subsample S of N is stored as full text. To verify whether or not a k-mer x is present in the graph, the hash value of x is computed thanks to $f(x)$, then a backward (respectively forward) traversal is performed using IN (respectively OUT) to reach a k-mer $y \in S$ which is finally compared to x. If a k-mer x does not belong to the graph, but the hash function provided a valid hash value, then additional verification needs to be performed. In such a case several accesses are performed to make sure that x does not belong to the graph.

2.3.3 Memory storage optimization of dBG

On compacting the dBG

From a de Bruijn graph, we can define two important components: The path and the maximal non-branching path. A maximal non-branching path is a path $p = \{f, u_1, u_2, ..., u_{|p|}, l\}$ where every node has exactly one in-neighbor and one out-neighbor except for f that could have zero or at least two in-neighbors and l that could have zero or at least two out-neighbors. The nodes along every maximal non-branching path can be compacted to form a single node called a unitig. An illustration of a de Bruijn graph and its compacted version is provided in figure 2.6. The de Bruijn graph compaction is an important step in genome assembly. It has been used in several de Bruijn graph based genome assemblers [75, 58, 20]. Many efforts were made to compact the de Bruijn graph [51, 22, 43, 54]. DBGFM [22] proposed an algorithm called Bcalm to compact maximal non-branching paths in low memory. Later, this work was optimized to achieve a parallel and external memory compaction algorithm [19].

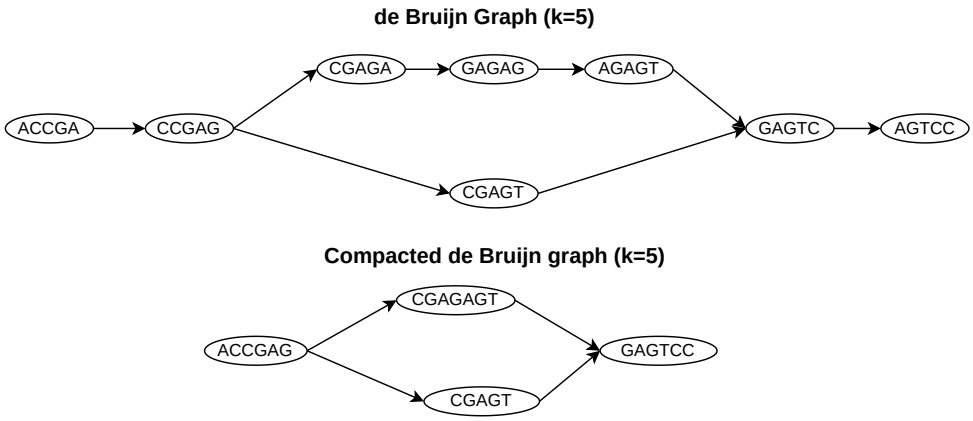

Figure 2.6 – A de Bruijn graph and its compacted version.

Compacted de Bruijn graph

Bcalm [22] and **Bcalm2** [19] are two methods to efficiently construct a compacted de Bruijn graph from reads or whole genomes. The key idea in Bcalm is to achieve parallel compaction of k-mers. To do this the k-mers are partitionned into buckets based on their left (respectively right) minimizer. First the left minimizer min_l and the right minimizer min_r of every k-mer are determined. A k-mer is then attributed to the bucket that corresponds to its left minimizer min_l and to another bucket that corresponds to its right minimizer min_r if $min_l \neq min_r$. These buckets are written to disk to be processed later. The compaction algorithm is then performed independently and in parallel over the k-mers of each bucket. This results in compacted strings inside each bucket. If in a bucket B_m associated to a minimizer m, the minimizer of the $(k-1)$-mer prefix or the minimizer of the $(k-1)$-mer suffix of a string $u \in B_m$ are different than m, then u is marked as a potential candidate to be merged with another string v in another bucket. The candidate strings are then merged during the reunification step. Bcalm2 intends to write (respectively read) to (respectively from) disk to compact the k-mers inside the buckets and to produce the final unitigs.

Bifrost [43] is a tool for constructing a compacted de Bruijn graph from a set of reads or assembled genomes. It builds first a blocked Bloom filter (BBF) from the k-mers of the input sequences. Bifrost goes again over the input sequences, and it uses the BBF to compute the unitigs of the graph (see section 2.3.3). As stated in section 2.1, the Bloom filters generate false positives, so the computed unitigs contains false positive k-mers. To

resolve this issue, Bifrost associates a counter to every k-mer during the construction of the BBF. Finally, it goes over the constructed unitigs to remove those k-mers with counters less than two. Doing this requires doing several passes over the input. It also requires to split or merge some unitigs where these false positives k-mers appear.

Cuttlefish [53] and its extension **Cuttlefish2** [54] are tools for constructing a compacted de Bruijn graph. The key difference is that Cuttlefish can construct the graph from only assembled genomes, while Cuttlefish2 can process a set of reads or a set of assembled genomes. Cuttlefish first starts with a k-mer counting step using KMC3 [55]. The key novelty in Cuttlefish with respect to the literature is the use of a deterministic finite state automata (DFA) [44] to compute the branching of every $(k-1)$-mer. A DFA is a theoretical computational model used in computer science to represent and analyze the behavior of systems with a finite number of states. Finally, Cuttlefish uses this DFA to compute the unitigs of the graph. Cuttlefish partitions the k-mers into disk buckets. These buckets are loaded in such a way the total memory usage does not exceed S, where S is determined by Cuttlefish. Thus, Cuttlefish requires high input-output bounds. KMC3 tends to use more memory when increasing the value of k which in turn affects Cuttlefish memory usage.

GGCAT [1] is a tool for constructing a compacted de Bruijn graph from a set of reads or a set of assembled genomes. GGCAT starts first with a k-mer counting step. It discards the k-mers whose frequencies are below a user defined threshold. The remaining k-mers are partitioned into buckets based on their minimizers. GGCAT then computes the compacted strings inside each bucket, and it determines alongside which compacted string is a potential candidate to be merged with strings from other buckets. Finally, it merges these compacted strings during the reuniting step. To limit its memory usage, GGCAT uses more disk and larger input-output.

2.3.4 Colored and compacted de Bruijn graph

A variant of the de Bruijn graph is the colored de Bruijn graph. It was introduced for the first time in the Cortex variant genotyper [49]. The graph can be built from a set of reads or a set of assembled genomes. Coloring the de Bruijn graph means that each k-mer in the graph is associated to the identifiers of the samples in which it appears (figure 2.7). It has been used in several applications such as pangenomics and gene prediction [93, 66,

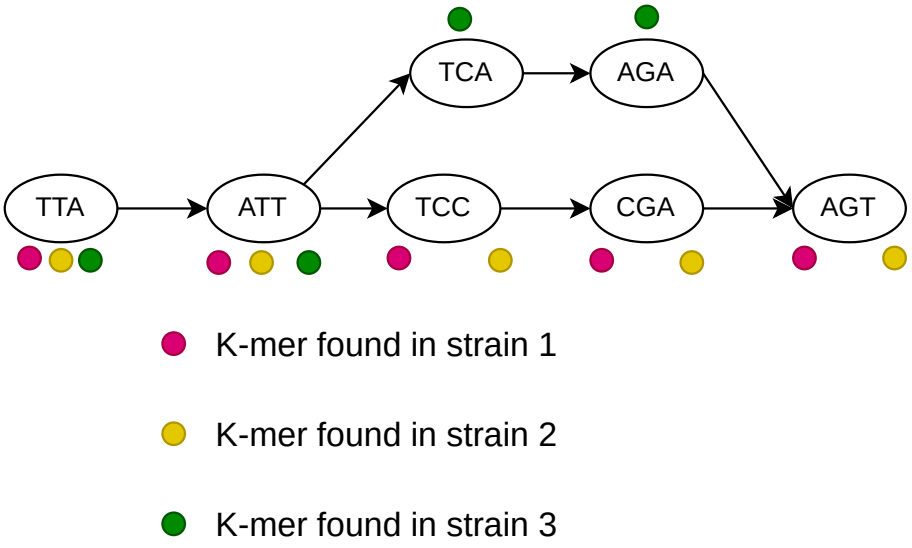

Figure 2.7 – A colored de Bruijn graph. Colors represent the identifier of the references from which the k-mers were sampled. The figure is adapted from [32].

64, 5, 45]. One can store the colors in a two-dimensional array C, also referred as the set of colors, where $C[i][j] = 1$ if k-mer i appears in the reference j, and $C[i][j] = 0$ otherwise. Doing so for all the k-mers in the graph is resource intensive. In recent years, efforts have been made to optimize memory storage of the colors. Mantis [78] compressed this color matrix by attributing a color class to a subset of the k-mers that belong to the same set of references. A color class is a binary array A where $A[i] = 1$ if the k-mers associated to A belong to the reference i, and $A[i] = 0$ otherwise. GGCAT [1] used the same approach by associating one color class for each set of monochromatic unitigs, those unitigs whose k-mers belong to the same set of genomes. An extension of the Fulgor tool [36] proposed to take advantage of the similarity between color classes to compress more the storage of colors.

2.4 Methods to update the de Bruijn graph

The advent and affordability of sequencing technologies have made the generation of billions of read sequences both time and cost-efficient. For example the cost of sequencing a human genome has dropped from around 300 million dollars in the 2000s to less than

one thousand dollars nowadays, with the time required decreasing from several months to just a few days, or even hours, depending on the platform used. While the question of efficiently storing and compressing large genomic datasets have been tackled in recent years, there is a growing interest in dynamic, space- and time-efficient data structures for handling these datasets, as the full reconstruction of such structures can be avoided for every sequenced sample. Having such data structures is a challenging research question since dynamicity and compressibility are contradictory. Yet, several efforts have been made to provide dynamic de Bruijn graph data structures. Updating a compacted de Bruijn graph aims to add new sequences to the graph or delete some sequences from the graph without full reconstruction.

2.4.1 Dynamic de Bruijn graph data structures

A handful of methods were proposed to update the de Bruijn graph. FDBG [27] supports the additions and deletions of nodes and edges. It builds an MPHF from the $(k-1)$-mers of the input along with a hash table to support the additions. FDBG [27] stores explicitly the new k-mers in the hash table which causes a high memory usage for large additions. DynamicBOSS [4] is based on the de Bruijn graph data structure of Bowe et al. [14], whereas BufBOSS [2] is based on the Wheeler graph [37]. The key difference between DynamicBOSS and BufBOSS is that DynamicBOSS adopts a dynamic bit vector to support the update operations, while BufBOSS provides a buffering scheme to add or delete nodes and edges. The relience of DynamicBOSS on dynamic bit vectors slows down its performance. DynamicBOSS builds also another graph G' from the new sequences and merges it with the input graph G. As for BufBOSS, it stores the added or deleted nodes in a buffer. The size of the buffer is limited to support a few number of additions or deletions. If the number of additions or deletions cannot be supported by the buffer, BufBOSS reconstructs the graph. DynamicMantis [7] uses Bentley J. and Saxe J. algorithm [12] to convert the static Mantis [78] into dynamic one. The limitation of this approach is that the conversion from a static to a dynamic structure can introduce increased computational complexity and memory overhead, potentially affecting performance and scalability for large datasets.

2.4.2 Dynamic Compacted de Bruijn graph data structures

While several methods were proposed to update the de Bruijn graph, Bifrost was the only one that updates the compacted de Bruijn graph. Given a compacted de Bruijn graph G and a set of new sequences S to be added to G, Bifrost builds another compacted de Bruijn graph G' from the sequences in S using its construction algorithm. Finally, it merges succinctly the two graphs. When contructing a new graph from the input sequences, Bifrost constructs unitigs from the shared k-mers between the initial graph and the input sequences which points to the fact that repetitive computations are performed. The reliance on constructing a graph from the new sequences introduces computational overhead, potentially limiting the scalability and efficiency of Bifrost, particularly when dealing with large graphs, highlighting the need for a more efficient updating mechanism.

2.5 Indexing the de Bruijn graph

One may need to search for a sequence in a compacted de Bruijn graph. As such, we need to index the k-mers of the graph to perform pattern matching in constant time. There are two main approaches for indexing the compacted de Bruijn graph: probabilistic and exact. The probabilistic approach to indexing the compacted de Bruijn graph uses data structures that approximate membership queries to save space. Bloom trees extend this concept by organizing multiple Bloom filters in a hierarchical structure to improve query efficiency. Probabilistic methods are particularly valuable when working with extremely large datasets where exact methods would be impractically resource-intensive.

In contrast, the exact approach to indexing ensures that all queries return precise results without any false positives or negatives. The exact approaches are categorized into full-text-based and hash-based methods. Full-text-based methods involve indexing the entire sequence data which often resulting in large index sizes. On the other hand, hash-based methods use hash functions to map sequences to unique hash values, enabling efficient and accurate retrieval. These methods balance the need for precision and efficiency. Here we will concentrate on hash based methods.

Pufferfish [6] is a tool for indexing a compacted de Bruijn graph. The tool computes a minimal perfect hash function f from the canonical k-mer of the graph. It also stores the positions of all the k-mers of the unitigs. Based on the hash value computed by f and

the corresponding position of a k-mer x, Pufferfish is able to validate if x is present in the graph by comparing it to the k-mer stored at this position or to its reverse complement. While this ensures a fast query, it consumes high memory especially in the case of large graphs. To address this issue, Pufferfish proposes a sparse indexing scheme. Instead of storing the positions of all the k-mers, they store the positions of the first and last k-mer of every unitig and the first and last k-mer of every reference sequence. This scheme is provided at the cost of query time. If a k-mer is searched in the index, and its position is not stored, Pufferfish keep searching the nearby k-mer whose position is stored.

BLight [68] constructs an index of a compacted de Bruijn graph. Super-k-mers are computed from the set of unitigs. All k-mers that share the same minimizers are placed in the same bucket. For each bucket, an MPHF is used to index the k-mers of the bucket. To find a k-mer, first its bucket is retrieved using its minimizer. A rank is calculated to find which super-k-mer it may belong to. Finally the k-mers of this super-k-mer are checked to test the existence of the query k-mer in the index. The tool allows for a sub-sampling strategy at the cost of higher computation time. When subsampling is performed, the number of super-k-mers to be checked increases.

Bifrost [43] provides also the possibility to index a compacted de Bruijn graph to allow the query of reads or the addition of new sequences to the graph. It stores the unitigs in a hash table. It indexes the minimizers in the graph. It associates to every minimizer all its occurrences in the graph. Given a k-mer query x, it gets compared to all the k-mers that share the same minimizer of x. This design is memory efficient since there are less minimizers than k-mers in the graph, but it is not time efficient because it is linear with respect to the number of occurrence of the minimizer of the query.

SSHash [79] is an indexing data structure of a collection of samples. It builds an index from a compacted de Bruijn graph. The data structure partitions the k-mers into buckets where each bucket has its own minimizer. Consecutive k-mers sharing the same minimizer are grouped together to form super-k-mers. The strings input are grouped together in $2N$ bits where N is the total number of bases in the input. An additional array of offsets is used to indicate the starting and ending of each unitig in the binary array to avoid querying alien k-mers, those k-mers that do not belong to the data structure. An array of sizes is maintained to indicate the size of each bucket. The hash values of minimizers are computed using an MPHF over the minimizers. Large buckets are partitioned to avoid high computation time of locating a k-mer in the index. To query a k-mer, its bucket is

retrieved from its minimizer. Additional checks are computed to test if the k-mer might be in a partition or in a small bucket. The actual location of the k-mer super-k-mer is retrieved and then all the k-mers of the super-k-mer are compared to this k-mer.

2.6 Conclusion

The exponential growth in genomic databases highlights the need of efficient data structures for handling petabases of data. de Bruijn graphs (dBGs) handle large datasets efficiently by compactly representing sequences as overlapping k-mers, significantly reducing memory usage while maintaining scalability. Several methods for constructing the de Bruijn graph were proposed. The characteritics of paths in the graph, the need to assemble genomes and the motivation to optimize the memory consumption of de Bruijn graph have led to the development of several compaction algorithms such as BCALM [19], PanTools [93],TwoPaCo [73], deGSM [39], Cuttlefish [54], Bifrost [43] and GGCAT [1]. They differ by their methods of counting k-mers, the intermediate data structures to store the data, the reliance on disk or memory, and their concurrency techniques.
The growth of genomic databases shed light on the need of dynamic data structures since the construction from scratch can be avoided for every added sequence. Several efforts were made on the initial proposed de Bruijn graph data structures. FDBG [27], DynamicBOSS [4] and BufBOSS [2] provided the ability to add or delete k-mer to or from the graph. However, their reliance on reconstruction of the graph after a few number of updates slows down their performance. Additionally, their reliance on hash tables or buffers to store the added or deleted sequences increase their memory usage. There is also a need to provide dynamic compacted de Bruijn graph. Bifrost is the only method that supports mutability operation. It builds another graph from the new sequences, and it succinctly merges it with the input graph even though there are few number of k-mer to be added. Hence, it highlights the need to develop methods that tackle the update more efficiently. The interest is not only to store the genomic data, but also to provide the ability to retrieve these data through efficient query methods. Several methods to index the compacted de Bruijn graph were developed such as Pufferfish [6], BLight [68], Bifrost [43] and SSHash [79]. Out of these methods, only Bifrost supports the mutability of the index while adding the sequences to the graph. However, the query time is linear with respect to the number of occurrence of the minimizer of a k-mer query, which means that repet-

itive access to the data structure need to be performed, thus increasing the query time. All in all, these limitations in the literature highlight the need to efficiently update the compacted de Bruijn graph and its index. The method in question should be able to identify which regions in the graph and which parts of the index should be updated to avoid full reconstruction while keeping the query time constant.

Chapter 3

Strategies to update a compacted de Bruijn graph

This chapter presents the main contribution of this thesis: a strategy to update a compacted de Bruijn graph and its index, implemented in a tool called Cdbgtricks [41]. This approach is situated within the context of pan-genomic analysis introduced in Chapter 1, where managing large sets of DNA sequences poses significant challenges. Specifically, the ability to efficiently update a compacted graph is crucial for handling the dynamic nature of pan-genomes, which continually evolve with the addition of new data.

3.1 Using k-mer set operations to update a compacted de Bruijn graph

Updating a compacted de Bruijn graph involves adding new sequences to the graph. Instead of reconstructing the graph from scratch and performing repetitive operations, we need to add sequences to a constructed graph. This process is crucial because in many biological applications, new data is constantly being generated, whether from new genome assemblies or sequencing projects. By integrating these new sequences, we ensure that the de Bruijn graph remains up-to-date and reflective of the available dataset.
Here, we show how we reduce the problem of updating the graph to the simpler task of adding a set of new k-mers. By focusing on k-mers, we can efficiently integrate new sequences without reconstructing the entire graph from scratch, thus saving computational time.

3.1.1 The graph and the new sequences as sets of k-mers

We have seen in the previous chapter that the de compacted de Bruijn graph is obtained by compacting the k-mers of each of the maximal non-branching path. Similarly

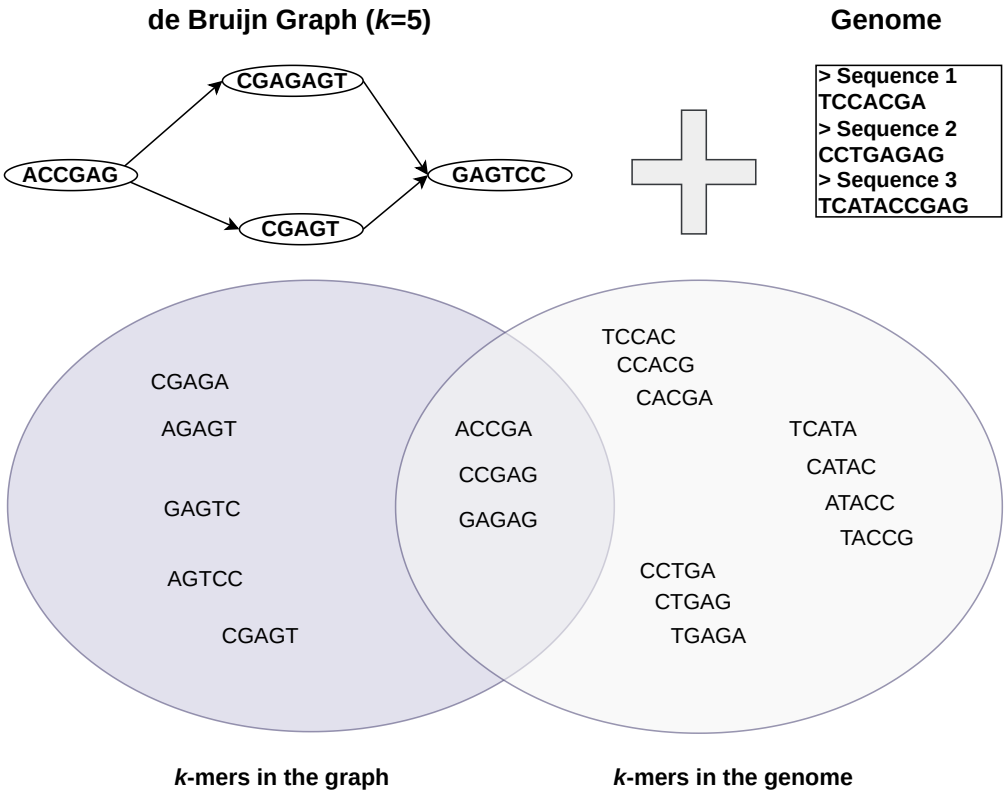

Figure 3.1 – **Venn diagram**. Representing the compacted de Bruijn graph (top left) and the genome to be added (top right) as two sets of k-mers.

the sequences that we want to add to the graph can be converted to a set of distinct k-mers using k-mer counting methods [85, 55].

3.1.2 Efficient set operations for updating a compacted de Bruijn graph

In a compacted de Bruijn graph G, the k-mers are unique. The sequences S that should be added to G share a subset of their set of distinct k-mers with the graph. Let's call K_G the set of k-mers in G and K_S the set of distinct k-mers in S (figure 3.1). Adding the k-mers of K_S to K_G can be reduced to adding the set of k-mers $N = K_S - K_G$, the set of new k-mers that are in S but not in G. Now adding the k-mers of S can be formulated as:

$$K_G \cup K_S = K_G \cup (K_S - K_G) = K_G \cup N \tag{3.1}$$

Although we could have enumerated the k-mers of G and the k-mers of S to find N, we decided to find N using kmtricks [56] that provides set operations over sets of k-mers out of which is the set difference in time- and memory-efficient manner. To determine the set N using kmtricks, we proceed as follows:

1. Construct one matrix K_G of k-mers for the graph and one matrix K_S of k-mers for the new sequences.
2. Calculate the difference between these two matrices $N = K_S - K_G$

3.2 Possible operations when adding a single k-mer

A key operation to update the compacted de Bruijn graph is to add the set of new k-mers N. When adding a k-mer x in N to the graph, one of the following cases may happen:

1. **Add x as a new unitig**. Neither $pref(x)$ nor $suff(x)$ appears in any unitig of G, where $pref(x)$ is the prefix of x of size $k-1$, and $suff(x)$ is the suffix of x of size $k-1$. In this case, the existing unitigs of G are not modified and x is added as a new single unitig in G (Fig 3.2.a).
2. **Right extension of a unitig**. If $pref(x)$ equals $suff(u)$ for a unitig u of G, and u has no out-neighbor, then u is extended with the last character of x ($u = u \odot x[k-1]$) (Fig 3.2.b).
3. **Left extension of a unitig**. If $suff(x)$ equals $pref(u)$ for a unitig u of G and u has no in-neighbor, then the first character of x is added to the left of u ($u = x[0] \odot u$) (Fig 3.2.c).
4. **Merge two unitigs**. If the addition of x leads to the right extension of a unitig u_1 **and** to the left extension of a unitig u_2, then, after the extensions, $suff(u_1) = pref(u_2)$. In this case the two unitigs u_1 and u_2 are merged into a unique unitig $u = u_1 \odot u_2(k, |u_2|)$ (Fig 3.2.d).
5. **Splitting a unitig**. If $pref(x)$ exists in a unitig u of G, not being the suffix nor the prefix of u, then u is split into two unitigs u_1 and u_2 where $suff(u_1) = pref(u_2) = pref(x)$ and x is added as a single unitig. Respectively, if $suff(x)$

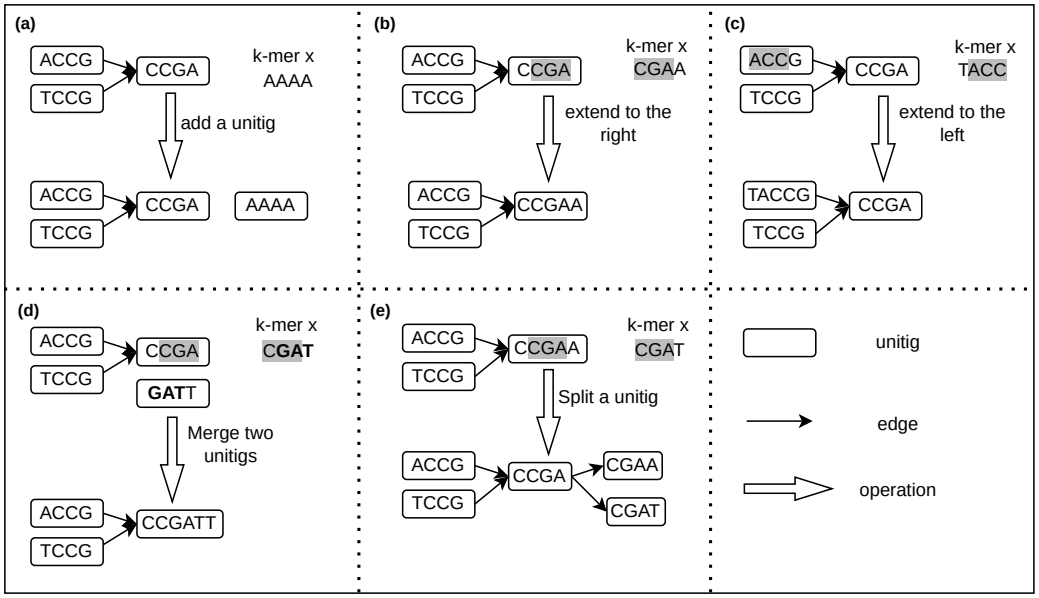

Figure 3.2 – **Possible operations when adding a k-mer to a compacted de Bruijn graph with $k = 4$. (a)** Adding the k-mer as a new unitig. **(b)** Extending a unitig to the right. **(c)** Extending a unitig to the left. **(d)** Merging two unitigs. **(e)** Split a unitig into two unitigs. Gray and bold sequences represent overlap between the added k-mer and some unitigs of the graph.

exists in a unitig u of G, not being the suffix nor the prefix of u, then u is split into two unitigs u_1 and u_2 where $suff(u_1) = pref(u_2) = suff(x)$ and x is added as a single unitig (Fig 3.2.e).

3.3 Graph Indexation

Determining whether or not $pref(x)$ or $suff(x)$ of a k-mer x belong to the graph, we need to rely on the pattern matching of these substrings in the unitigs of G. In order to achieve these operations in a fast way, we need to index the graph.

3.3.1 Indexing k-mers to query $(k-1)$-mers

Despite the fact that we query $(k-1)$-mers, we index the k-mers of the graph. A $(k-1)$-mer x may have eight occurrences because it can be the suffixes of four possible k-mers and the prefix of four other prefixes. Indexing from one to eight couples (unitig id, offset) per indexed element is not efficient as it requires a structure of undefined and

variable size. This leads to heavy data-structures and cache-misses on construction and query times. To cope with this issue, we chose to index k-mers instead of $(k-1)$-mers. Indeed, each k-mer of a compacted de Bruign graph occurs exactly at one position. Given this indexing scheme in which k-mers are indexed, when querying a $(k-1)$-mer x', the eight possible k-mers containing this $(k-1)$-mer (four k-mers in which x' is the prefix, and four k-mers in which x' is the suffix) are queried. If a match is found, the offset of the $(k-1)$-mer is deduced depending on the case (either x' is the prefix or the suffix of a queried k-mer for which a match is found).

There are three possible considerations to index the k-mers of the compacted de Bruijn graph:

1. Index the k-mers as they appear in the unitigs.
2. Index the k-mers and their reverse complements.
3. Index only the canonical form of the k-mers.

If we only index the k-mers as they appear in the unitigs, and we want to query a k-mer x, then we have to deal with the case when the index shows that x does not belong to any unitig. In this case we have to query the reverse complement $\bar{x}$ of x. This means that we have to perform two accesses to the index for those k-mers for which the index indicates that they do not belong to the graph from the first access.
Indexing both the k-mers and their reverse complements is even worse. In such a case, we didn't resolve the problem of double access, but we double the amount of memory needed to build the index.
To resolve such an issue, we index each k-mer in its canonical form. Then, a queried k-mer is searched in its canonical form.

3.3.2 Integrating MPHF in the index

Efficient indexing of k-mers in a compacted de Bruijn graph can be achieved through hashing. Here we argue about the choice of hashing strategy adopted during the development of our method.

MPHF over hash tables

One possible way to index the k-mers is to build a hash table whose keys are the canonical k-mers and whose values are the positions of these k-mers in the graph. This requires explicitly storing the k-mers and their positions. Take an example of a graph that

has 10^9 k-mers. If $k = 31$ then to store a k-mer we need to use 8 bytes per k-mer. The position takes as well 8 bytes. In this case, storing these k-mers requires 14.9 Gigabytes. However, the k-mers are already in the compacted unitigs, so we do not need to store them in a hash table. We need to associate to a given k-mer x its position in the graph. To do so, we can construct a minimal perfect hash function f from these k-mers. The positions of the k-mers can be stored in a vector V, and the position of a k-mer x can be retrieved thanks to $V[f(x)]$.

PHOBIC over other MPHFs

The construction of a minimal perfect hash function has been extensively studied in the field of computer science, resulting in a variety of methods such as BBHash [62], RecSplit [35], PTHash [82] and PHOBIC [42] to name a few. In the work of this thesis, we decided to integrate PHOBIC for constructing the MPHFs in the index. While other methods compromise between memory usage and construction and query time, PHOBIC results show the best query performance with respect to the state of the art.
LPHash [81] preserves locality of k-mers, i.e., consecutive k-mers get consecutive hash values. This is suitable for building one MPHF from all the set of k-mers to preserve the locality of k-mers. However, to efficiently update a compacted de Bruijn graph, its set of k-mers must be partitioned as will be shown in the subsequent sections. In such a case, the locality of the k-mers is not preserved.

3.3.3 k-mers distribution into buckets

As argued earlier, we can construct an MPHF f from the k-mers in the graph, and we store the positions in a vector V. There are two observations to be made here:

— At query time, f can give valid hash values for alien k-mers, which are k-mers that are not present in the graph. To handle this, we compare the queried k-mer to the actual k-mer in the graph, whose position is retrieved thanks to $V[f(canonical(x))]$ where $canonical(x)$ is the canonical form of x.
— Adding new k-mers requires to recompute f.

This last point is problematic, since for every addition operation, f must be recomputed, which is a linear-time operation in terms of the number of indexed k-mers. To resolve this, we divide the set of k-mers in the graph into multiple subsets called “buckets”. Each

subset is indexed using its own MPHF. MPHFs are computed using PHOBIC [42]. The key idea being that while adding sequences to a graph, only a subset of the buckets are modified, and so only a subset of the MPHFs have to be recomputed. At query time, the bucket of the queried k-mer x is retrieved, and the corresponding MPHF provides the position of x in the graph.

Formally, we define $\{b_0, b_1, ..., b_{n-1}\}$ buckets. For each of these buckets b_i, an MPHF f_i is computed on its k-mer.

The k-mers in the graph are partitioned into buckets based on their minimizers. Note that we use the canonical form of the minimizer of the k-mer. k-mers sharing the same canonical minimizer cannot be distributed into different buckets.

Hashed minimizers over lexicographic minimizers

We need to calculate for every k-mer x in the graph its minimizer of length m where $m < k$. Recall that a minimizer is the smallest m-mer of all the m-mers of x according to a defined order. There are different strategies to compute the minimizer of a k-mer x. One way is to take the lexicographic order. In such a case, the minimizer of a k-mer x is the lexicographically smallest m-mer of x. However, this scheme would result on very large buckets affected by the presence of A's or T's in if we are taking a canonical minimizer, and very small buckets otherwise. To resolve this, we can use a random hashing scheme that minimize the discrepancy between the sizes of the buckets.

Buckets and super-buckets

Although we adopted a random hashing scheme to compute the minimizer of k-mer, the strategy of relying on the minimizers may result in a well-known problem of non-uniform distribution of the k-mers in the buckets [21]. Some buckets could be orders of magnitudes larger than some others. Also, the small buckets are problematic for the construction of an MPHF using PHOBIC, as higher number of $bits/k-mer$ is required for small buckets (see Figure 3.3). In fact all the methods that construct an MPHF from a small set of keys uses higher $bits/k-mer$ compared to constructing the MPHF from a larger set of keys. This highlights two important problems:

1. How to achieve uniform partitioning of k-mers to avoid these distribution issues.
2. How to efficiently handle difficult cases for MPHFs, especially when dealing with small buckets that require disproportionately more resources.

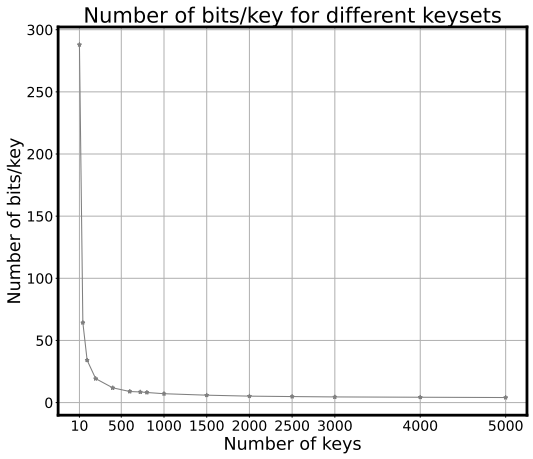

Figure 3.3 – **The number of bits/key required for building a MPHF with PHOBIC**. MPHFs from different sets of keys of sizes ranging from 10 to 5000 random 64-bit keys were computed by PHOBIC, and *bits/key* were then measured.

All in all, we propose a strategy so that each of the batches contains a minimum number of k-mers.

— The number of k-mer sharing the same minimizer should be at least equal to a parameter ρ for creating a bucket. Note that the size of a bucket does not have an upper-bound.

— For the remaining k-mers, we process them by groups of k-mers where the k-mers within a group share the same minimizer. From these groups, we create what so-called "super-buckets" which are buckets containing k-mer that share different minimizers. We start with an empty super-bucket S_0 to which we add the groups of k-mer one by one. Once the number of k-mer added to S_0 exceeds $\gamma \times \rho$ k-mers (with γ a user defined multiplicative factor), we create a new super-bucket. We keep creating and filling super-buckets until all groups of k-mers are processed. The rationale behind this strategy is to achieve a balanced distribution of k-mers on the super-buckets. It is important to note that the size of a super-bucket has an upper bound, which we address in section 3.4.2.

Figure 3.4 shows an example of distributing the k-mers of a compacted de Bruijn graph into buckets when $\rho = 5$ and $\gamma = 1$. In this example two buckets were created since the number of k-mers shring the same minimizer is at least 5. The remaining k-mers were grouped in two super-buckets so that each super-bucket contain at least $\gamma\rho = 5$ k-mers.

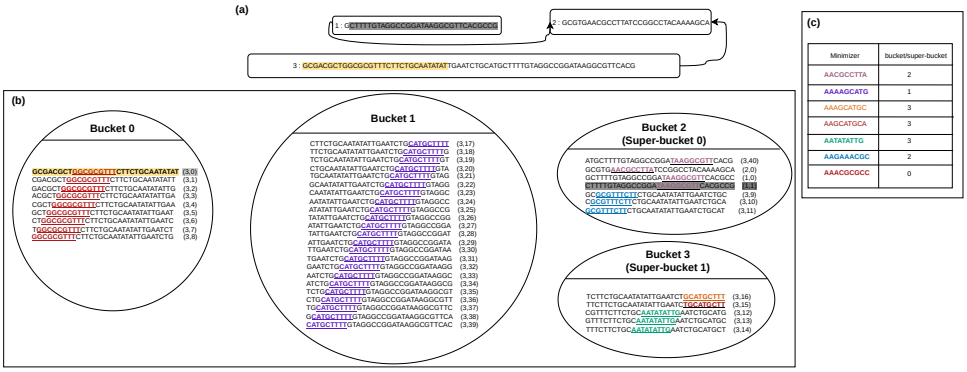

Figure 3.4 – **Example of partionning the k-mers into buckets for $\rho = 5$ and $\gamma = 1$. (a)** a compacted de Bruijn graph with $k = 31$. **(b)** the resultant buckets using using $\rho = 5$ and $\gamma = 1$. **(c)** the set of lexigraphic minimizers.

Finally the data-structure is composed of the following components, represented in Figure 3.5:

1. A hash table T that maps each minimizer to its bucket identifier. The hash table will be used to identify the bucket that may contain a given k-mer x. The identifier of the bucket is then $b_i = T[minimizer(x)]$.
2. A hash table F that maps each bucket identifier to its MPHFs. Hence, $F[b_i]$ is the MPHF computed from the set of k-mers in bucket b_i.
3. A hash table U that maps the identifier of each unitig of the graph to its sequence. Hence, $U[u_i]$ is the unitig sequence whose identifier is u_i.
4. The positions of the k-mers in the graph are stored in a 2-D vector P. $P[b_i]$ is the vector of positions for the k-mers in bucket b_i. $P[b_i][F[b_i](x)]$ is the tuple position $< u_{id}, u_{off}, orientation >$ for the k-mer x in the bucket b_i.

Overall given the position of a k-mer x, x can be retrieved by retrieving the unitig $u = U[u_{id}]$ from the hash table U, hence $x = u(u_{off}, u_{off} + k)$ (Figure 3.5.c). Note that if the minimizer of x is not present in T, then x does not belong to the graph (Figure 3.5.b).

3.4 Updating the graph and its index

To update the compacted de Bruijn graph, we need to compact the k-mers and to handle the splits and joins that might occur while compacting the k-mers. We also

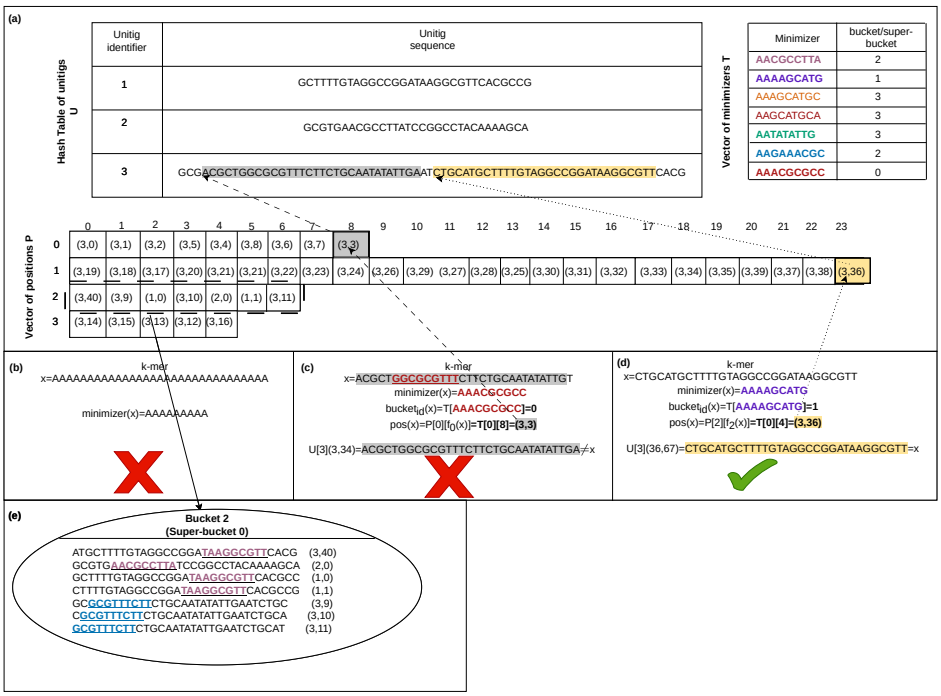

Figure 3.5 – **Overview of the data structure**. **(a)** the hash table U of the unitigs of a compacted de Bruijn graph with $k = 31$; the hash table T of minimizers that maps a minimizer to its corresponding bucket or super-bucket; and the vector P of positions of the k-mers as a tuple (u_{id}, u_{off}) where u_{id} is the identifier of the unitig in which the k-mer occur at offset u_{off}. Note that the values on top of the vector P represent the hash value of the k-mers computed by the MPHF of their bucket or super-bucket, and the values to the left of P represent the bucket or super-bucket identifier. **(b)** querying an absent k-mer whose minimizer is not in the index. **(c)** querying an absent k-mer whose minimizer is in the index. **(d)** querying a present k-mer. **(e)** the dashed box is converted back to a super-bucket of the k-mers.

update the index to reflect the changes in the graph.

3.4.1 Updating the graph

To update the graph, we need to compute the unitigs from the set of new k-mers. As explained in Section 3.2, the addition of the new k-mers lead to the split of some unitigs or the join of some other unitigs. Therefore, after compacting the new k-mers, we need to perform the splits and the joins.

Computing future unitigs

Recall that N denotes the set of k-mers to be added to a compacted de Bruijn graph G. In Section 3.1.2 we proposed an overview of algorithms in which k-mers from N are added one after another to G. In practice, for performance reasons, we first compact k-mers from N into what we call "funitigs" (for *future unitigs*).

The funitigs are not simply the unitigs of N as any k-mers of those funitigs that is already in G must be either a prefix or a suffix of a funitig. Doing so, the funitigs are not split latter when added to the graph. The details about the funitig construction are given in Algorithm 1.

Once the funitigs are constructed, each of them is added to the graph one after the other. The rules described in section 3.2 for adding a k-mer to G exactly apply for adding a funitig to G. The Cdbgtricks [41] tool exactly implements those rules, that we do not recall here.

Post funitig computation: splits and joins

As explained in section 3.2, adding new k-mers may lead to split a unitig or to join two unitigs. In Cdbgtricks, we do not perform a split or a join once it is observed (Figure 3.6). Alternatively, we store all the split positions (u_{id}, u_{off}) in a buffer, and the join positions in another buffer. Once all the funitigs are computed, we sort the array of splits by u_{id} then by u_{off}. This ensures that all the splits positions for every unitig are in consecutive order. The benefit from this is to minimize the cache misses when performing all the splits of a given unitig at once. A unitig is loaded once, and all its splits are performed in a linear scan over its splits. Concerning the joins, for every unitig in the join buffer, we associate to it the identifier of the funitig with which it will be joined. A unitig will be join with a funitig either from the left (prefix) or from the right (suffix). If we perform

Algorithm 1 Test k-mer presence

Require: Hash Table of unitigs U, k-mer s, Table of buckets T, Table of mphfs F
Ensure: s belongs to a unitig

```
function KmerPresence(H,s,T,F)
    mini_s ← canonicalminimizer(s)        ▷ compute the canonical minimizer of s
    i ← T[mini_s]                         ▷ retrieve the identifier of the bucket of mini_s
    if i ≠ NIL then                       ▷ the minimizer is in the index
        f_i ← F[i]                        ▷ retrieve the mphf of the bucket
        q ← canonical(s)                  ▷ compute the canonical form of s
        j ← f_i(q)                        ▷ compute the hash value of q
        p_j ← P[i][j]                     ▷ retrieve the position tuple of q
        id ← p_j.u_id                     ▷ retrieve the unitig identifier
        o ← p_j.u_off                     ▷ retrieve the offset of q in this unitig
        u ← U[id]                         ▷ retrieve the unitig
        return canonical(u(o, o + k)) = q ▷ compare the k-mers
    return false   ▷ the minimizer is not in the index, so s is not in the graph
```

the joins before the splits, we have to update all the offset of splits for the unitigs that are under split and join at the same time. To avoid this, we perform all the splits before the joins.

3.4.2 Updating the index

The Cdbgtricks software enables to update the index of a compacted de Bruijn graph, after the addition of sequences. The updated index can serve for any future update on the graph and for k-mer queries against the graph. The update of the index is tackled in several steps as will be detailed in this section.

Updating positions of k-mers in the graph

When performing several splits on a unitig u, the new generated unitigs from u are assigned new identifiers. The offsets of the k-mers of u are changed. Hence, the position tuples $(u_{id}, u_{offset}, orientation)$ should be updated. The same thing holds when merging a unitig with a funitig. For every split or join, the positions tuples for a k-mer x are recomputed and are re-stored in the position vector $P[i][j]$ where $i = minimizer(x)$ and $j = F[i](x)$.

Algorithm 2 Construct funitigs

Require: Hash table of new k-mers K,Index of the graph I
Ensure: A set of funitigs X

```
function ConstructFunitig(K)
    X ← ∅
    for each k-mer s ∈ K do
        if s is not marked as used then
            mark s as used
            s_1 ← ExtendRight(s)
            s_2 ← ExtendRight(s̄)
            X ← X ∪ {s̄_2 ⊙ s_1(k − 1, |s_1|)}
    return X
function ExtendRight(s)
    q ← s
    g ← s(1, k)                                  ▷ the (k − 1)-suffix of s
    while True do
        a ← FindRightExtensions(g)
        if a ≠ ϵ then
            b ← FindRightExtensions(ḡ)
            if b ≠ ϵ then
                q ← q ⊙ a
                g ← g(1, k − 1) ⊙ a
            else
                break
        else
            break
    return q
function FindRightExtensions(x)                  ▷ x is a (k − 1)-mer
    extension ← ϵ
    count ← 0
    for each a ∈ ∑ do
        q ← x ⊙ a
        if q ∈ K then
            extensions ← a
            count ← count + 1
        else if KMERPRESENCE(I.U,s,I.T,I.F) then   ▷ x is found in the graph
            extension ← ϵ
            break
    if count ≠ 1 then
        extension ← ϵ
    return extension
```

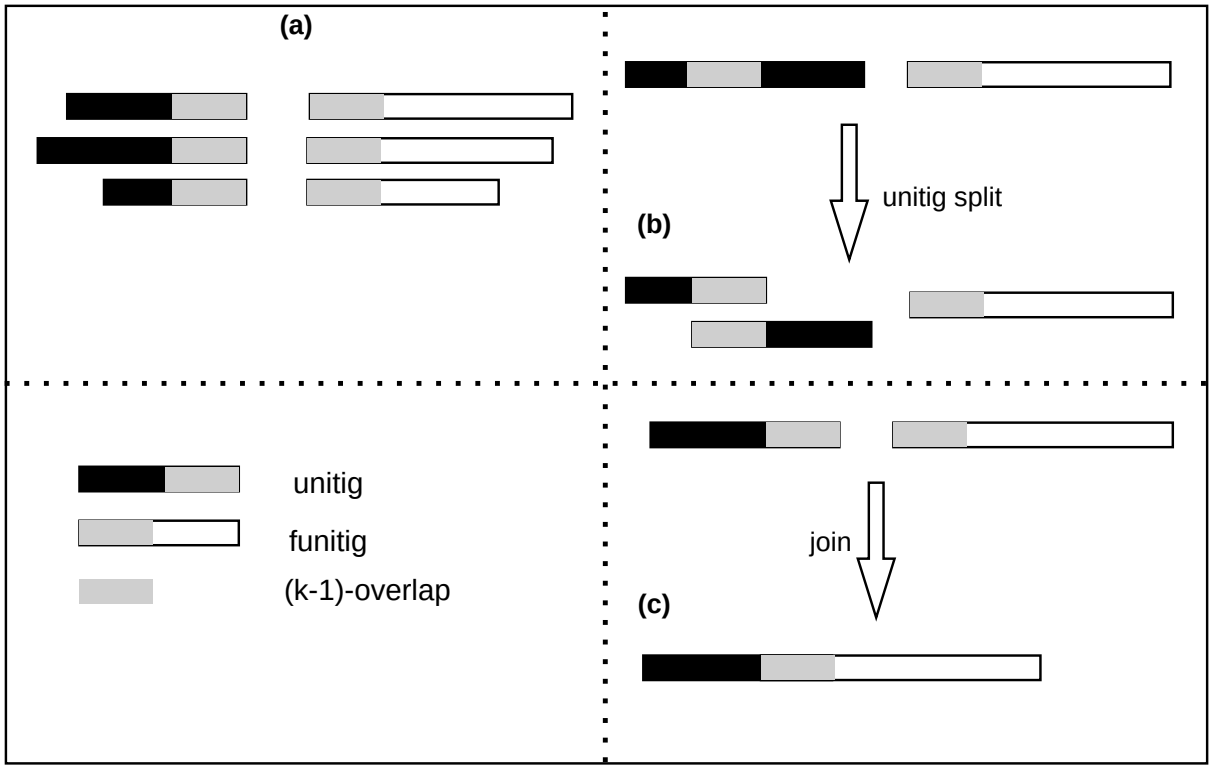

Figure 3.6 – Unitigs remain unchanged **(a)**. Split a unitig **(b)**. Join a unitig with a funitig **(c)**.

Greedy approach to partition a super-bucket

The minimizers of some of the new k-mers may already be associated with some buckets or super-buckets. The size of a bucket does not have an upper bound. Although we could technically enforce an upper bound on the size of buckets, this will lead to associating several buckets for the same minimizer and will slow down the query operation. On the other hand, we have set an upper bound on the size of super-buckets. The upper bound is double the original size of a super-bucket ($\gamma \times \rho$). Once the size of a super-bucket is doubled, it gets divided into two super-buckets. We need to ensure that the sizes of the new super-buckets are approximately close. To do so, we adopted a greedy approach to partition a super-bucket (see Algorithm 3).

Algorithm 3 Divide a super-bucket into two super-buckets

Require: A super-bucket of k-mers B
Ensure: Two balanced super-buckets B_1 and B_2

```
Initialize four empty sets, B1, B2, M1, M2
for each k-mer x ∈ B do
    m ← minimizer(x)
    if m ∈ M1 then
        B1 ← B1 ∪ {x}
    else if m ∈ M2 then
        B2 ← B2 ∪ {x}
    else if |B1| < |B2| then
        B1 ← B1 ∪ {x}
        M1 ← M1 ∪ {m}
    else
        B2 ← B2 ∪ {x}
        M2 ← M2 ∪ {m}
```

Re-computing the MPHFs

The MPHF of the buckets and the super-buckets to which new k-mers are added should be re-computed. Similarly, each of the super-buckets that results from partitioning a super-bucket is assigned to a new MPHF. We retrieve the k-mers of these buckets from the unitigs, and we re-compute the MPHF. Note that the hash values of these re-computed MPHFs are not the same as the previous ones. This means that the position tuples in the vector P needs to be re-arranged with respect to the hash values of these MPHFs.

3.5 Results

All presented results are reproducible using command lines and versions of tested tools, that are given in this repository `https://github.com/khodor14/Cdbgtricks_experiments`. The executions were performed on the GenOuest platform on a node with 4×8 cores Xeon E5-2660 2,20 GHz with 128 GB of memory.

Genome datasets

We tested Cdbgtricks in two frameworks, corresponding to two input datasets of distinct size and complexity. The first one, called "*human*" is composed of 100 assembled human genomes that were used in the GGCAT experiments [1]. These genomes are available on zenodo (`10.5281/zenodo.7506049,10.5281/zenodo.7506425`). The second set, called "*coli*" is composed of 7055 *E. coli* genomes downloaded from NCBI (`https://ftp.ncbi.nlm.nih.gov/genomes/all/GCF/030/`).
The idea is to test the performance of Cdbgtricks on large genomes (human genomes) each of which is composed approximately of 3 billion k-mers, and on smaller genomes (the *E. coli* genomes) each of which is composed approximately of 3 million k-mers.

While Cdbgtricks is capable of initially constructing a compacted de Bruijn graph from scratch, it is worth noting that there are faster alternatives for creating the initial graph. As such, for each dataset, we created an initial graph in fasta format from one genome (chosen as the first in the alphabetic order of the file names) using Bifrost. Once created Cdbgtricks can be used for indexing this initial graph. Subsequently, for each dataset, we added one by one the remaining genomes.

Used Parameters

The used parameters are the same for the two datasets. In all experiments we used $k = 31$ and minimizers of size $m = 11$. During the update experiments the tools were executed using 32 threads.

For Cdbgtricks, the parameters controlling the bucket size were set to default. The bucket lower bound size is $\rho = 5000$ and a the super-bucket multiplicative factor γ is set to 4. This setting of parameters means that the size of a super-bucket is approximately 20000 k-mers, and once it reaches 40000 k-mers, it gets divided into two super-buckets. The values of the parameters were chosen to ensure satisfactory results that will be shown in the subsequent sections.

3.5.1 Statistical results

While performing the update on the compacted de Bruijn graph and its index, we measured the percentage of updated buckets. As explained Section 3.3.3, one of the key ideas in Cdbgtricks is to distribute indexed k-mers into multiple buckets, each bucket being indexed with its own MPHF. Doing so, we expect that, while adding k-mers from novel sequences, k-mers are added to only a fraction of the buckets, and then only, a fraction of the MPHFs have to be recomputed. More precisely, we expect that the percentage of modified buckets, i.e. $100 \times \frac{\text{number of modified buckets}}{\text{total number of buckets}}$ decreases as the number of genomes in the graph increases. Note that here we do not differentiate buckets and super-buckets and we regroup these two notions in the term "buckets".

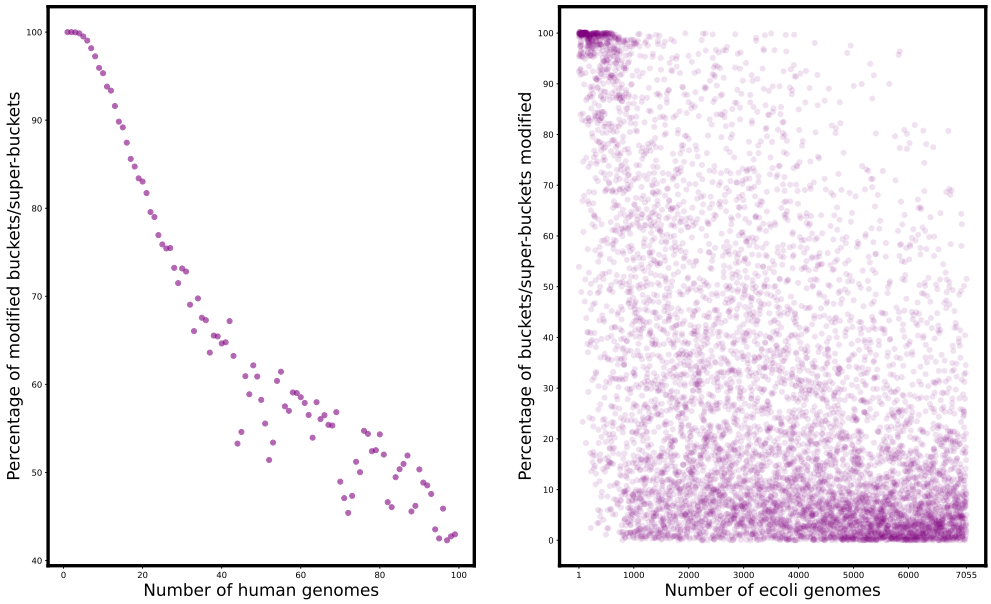

Figure 3.7 – **Percentage of modified buckets**.

In this section we test this expectation on the *human* and *E. coli* datasets. The results about the percentage of modified buckets are shown Figure 3.7. Results show that, as expected, the percentage of modified buckets decreases with respect to the number of genomes. The shaded cluster of points for the *E. coli* dataset shows that in the majority of cases, the percentage of modified buckets and super-buckets is less than 20%. More

specifically, results with, say, more than 5000 *E. coli* genomes show that, except for some outliers, less than 10% of the buckets are modified when adding a new genome.

These results validate the chosen default parameters, and they confirm the expectation that lesser and lesser buckets are modified while increasing the number of genomes of the same species in a compacted de Bruijn graph.

3.5.2 Time, Memory and disk

One of the main objective of Cdbgtricks is the time performances when updating a compacted de Bruijn graph with new sequences. In this context, we compared the Cdbgtricks update time, with the update time obtained thanks to Bifrost, also able to update an already created compacted de Bruijn graph. Furthermore, although GGCAT does not provide graph updating capabilities, we included it in our comparison due to its efficiency. While there is a need for a full comprehensive benchmark to compare the performance of Cdbgtricks to the existing dynamic construction methods of de Bruijn graph such as dynamicBoss [4] and bufBoss [2], we decided to stick to Bifrost as it was shown to be faster than these methods [2]. There exist also dynamic indices of a set of k-mers such as CBL [70] and Dynamic Mantis [7], but they do not keep track of the unitigs, meaning that one should traverse the index to retrieve the new set of unitigs after performing the update operation.
The memory and disk for the update with Cdbgtricks and Bifrost and for the construction with GGCAT are also reported. Results for the *human* dataset are shown in Figure 3.8. Note that, on the *human* dataset, GGCAT reached a timeout we set at two days on more than 71 human genomes. Hence, only the results for the first 71 genomes were reported for GGCAT. Globally, the results on this dataset show that Cdbgtricks is at least 2× faster than Bifrost on graphs composed of 50 genomes or more. Compared to GGCAT, Cdbgtricks is slower on this small number of genomes. Given that as the number of genomes in the graph increases, the GGCAT construction time naturally increases while the Cdbgtricks update time decreases, one can expect Cdbgtricks to be faster when dealing with more genomes than those tested here. However, given the observe GGCAT limitation after 71 genomes, we could not verify this fact in practice, at least for human genomes. The memory used by Cdbgtricks and Bifrost are slightly the same and are limited to a few dozen gigabytes. GGCAT uses much less memory, but needs up to order of magnitude more disk. Results for the *E. coli* dataset are shown in Figure 3.9. For the sake of clarity of presenting the results of the *E. coli*, each point represents the median time over a window

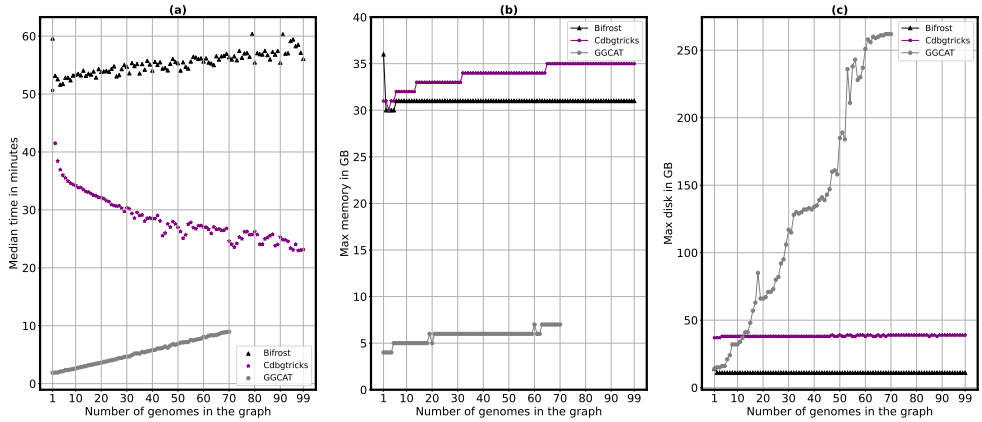

Figure 3.8 – **Results on human genomes dataset.** Time (a), memory (b) and disk usage (c) are given for updating a graph for Cdbgtricks and Bifrost and for constructing a graph from scratch for GGCAT.

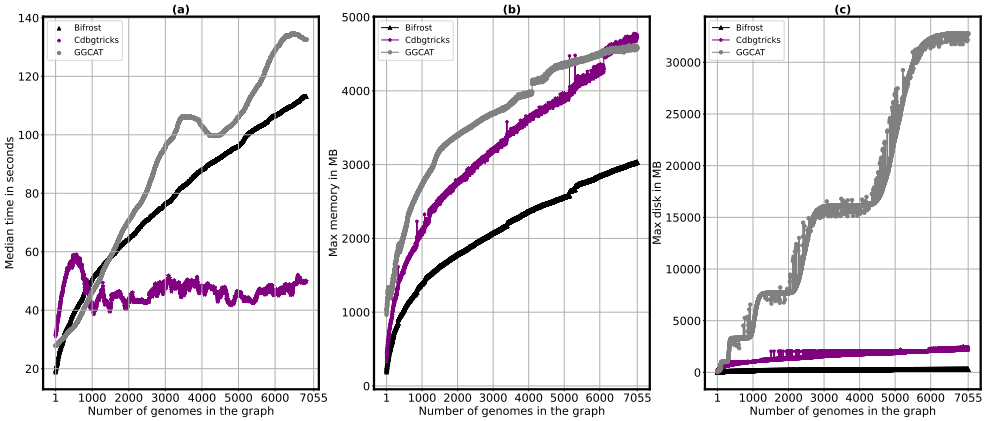

Figure 3.9 – **Results on *E. coli* genomes dataset.** Time (a), memory (b) and disk usage (c) are given for updating a graph for Cdbgtricks and Bifrost and for constructing a graph from scratch for GGCAT. The time is given as the median over a window of 200 consecutive points.

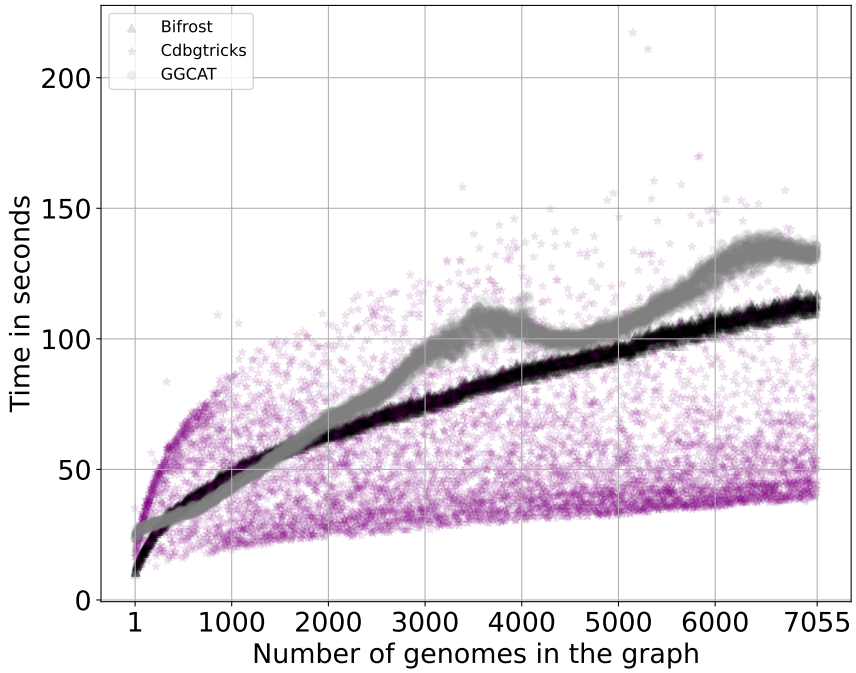

Figure 3.10 – **Results on *E. coli* genomes dataset.** Time is given for updating a graph for Cdbgtricks and Bifrost and for constructing a graph from scratch for GGCAT.

of 200 genomes. The detailed presentation of execution time is shown in figure 3.10. On this dataset, both Bifrost and GGCAT were faster than Cdbgtricks on graphs composed of less than a thousand genomes. However, in the vast majority of cases, when the number of genomes get higher than, say, 2000 genomes, Cdbgtricks is 2x to 3x faster than GGCAT and Bifrost. With Cdbgtricks, adding an *E. coli* genome to a compacted de Bruijn graph graph containing already few thousands genomes requires between 30 and 50 seconds. Cdbgtricks uses slightly the same amount of memory compared to GGCAT and roughly twice the amount of memory compared to Bifrost. Cdbgtricks uses up to 4x more disk compared to Bifrost, while it uses much less disk compared to GGCAT.

3.6 Conclusion

In this chapter, we presented the main contribution of this thesis, Cdbgtricks, a novel method for updating a compacted de Bruijn graph when adding new sequences such as

full genomes. The dynamicity of the proposed index is achieved thanks to the distribution of k-mers into multiple buckets, each bucket being indexed using a minimal perfect hash function (MPHF). The addition of new k-mers affects only a fraction of the buckets, for which the MPHF has to be recomputed. It is worth noting that the current version of Cdbgtricks supports k values up to 32. We will afford to support in the upcoming version larger values of k. We can adopt for example the method of Diego D. et al. [31]. In practice, when indexing a large number of genomes (dozens of human genomes or thousand of *E. coli* genomes) Cdbgtricks outperforms the computation time of state-of-the-art tools dedicated to the creation of the update of compacted de Bruijn graphs.

Chapter 4

Querying a compacted de Bruijn graph

This chapter addresses query strategies on compacted de Bruijn graphs (cDBGs), a critical component in various biological applications. It discusses different types of queries that can be performed with Cdbgtricks which is was introduced in chapter 3.

4.1 Choice of query type

In genomic application, one would like to match a given pattern q to a collection of sequences represented a compacted de Bruijn graph G. This process is referred as querying a compacted de Bruijn graph. There are two types of queries depending on the computation biology application:

1. Finding the presence or absence of q in the sequence of any path of G.
2. Finding a path p (a set of consecutive nodes) whose sequence best aligns with q.

Knowing the presence or absence of a query sequence q in G can be used to know whether or not q is similar to the collection of genomes used to build G. It can also be used to detect the variants such as single nucleotide polymorphisms (SNPs) presented in q with respect to G [48].

Find the path p whose sequence best aligns with q can also be used to detect variants such as single nucleotide polymorphisms, insertions or deletions. The approach of aligning the sequence q to the graph is also used to correct the sequencing errors of sequencing reads [60].

4.1.1 Presence or absence of a query

In some computational biology applications, we just need to know whether or not a given read R is present in compacted de Bruijn graph. In such a case we need to test the

presence or absence of every k-mer of R. Hence, we need only to index the k-mers of the graph without other information such as their positions in the unitigs.
One may use approximate methods, which use less memory than exact methods at the cost of precision given as:

$$Precision = \frac{true\ positives}{true\ positives\ +\ false\ positives} \tag{4.1}$$

where a false positive occur when reporting an element (a k-mer in this context) as present, but in fact it is absent in the dataset. One way to decrease the ratio of false positives is by increasing the Bloom filter size which require larger memory to store the Bloom filter. Another way was introduced in findere [87] that indexes the s-mers of all the k-mers with $s < k$. To query a k-mer x, findere checks all its s-mers. If at least one of these $k - s + 1$ s-mers is absent, findere reports x as absent, otherwise x is considered to be present. This was based on the observation that consecutive false positives are unlikely to appear consecutively. Hence, it helps, when querying consecutive s-mers, filter out false positive k-mers, which in turn decrease the false positive rate.

To completely eliminate false positives, one may use exact methods based on hashing such as BLight [68], SSHash [79] and GGCAT [1], or text based methods that employs the Burrow Wheeler Transform [17] such as SBWT [3] offers exact matching of query sequences. Finally, The CBL [70] is also an exact membership data structure.

4.1.2 Mapping

Mapping sequences to a compacted de Bruijn graph is more complex than simply reporting their presence or absence. When checking for the presence or absence of a k-mer, the task is relatively straightforward: we only need to verify if each k-mer from the queried sequence exists in the graph. However, mapping a sequence to the graph involves aligning the entire sequence to a path within the graph. This is a more challenging task because it requires not only knowing whether a k-mer is present, but also identifying which unitig it belongs to and its exact position within that unitig. The matched k-mers need then to be chained to find the best alignment path using combinatorial methods. This requires more sophisticated data structures, and more complex algorithm to perform this task, which makes the problem in turn harder than the problem of presence or absence.

4.1.3 Pseudo-mapping

Instead of finding the path whose sequence best aligns with a given query q, one may need to only determine the set of unitigs that contain certain k-mers of q. A similar definition was introduced by Bray N. et al [15] but in the context of RNA-seq quantification. This task lies between the simplicity of reporting the presence or absence of the sequence and the more sophisticated task which is mapping the sequence to the graph. For knowing whether or not some k-mers of the sequence belong to the graph and to which unitig they belong, we need similar data structures used to map the sequences to the graph. Nevertheless, the task is simpler because we don't need to find the best path using some complex approaches.

4.2 Queries in Cdbgtricks

As detailed in chapter 3, in Cdbgtricks [41], each k-mer x of a compacted de Bruijn graph is associated with its tuple position $(u_{id}, u_{off}, orientation)$. The k-mers of the graph are partitioned into buckets based on their minimizers. This means that for every queried k-mer x' we can test if it belongs to the graph by finding its minimizer and retrieving its tuple position p, and then comparing it to the k-mer of the graph associated with p. This index allows us to query reads and to find the set of unitigs where the k-mers of a read appear.

4.2.1 Query presence/absence in Cdbgtricks

In the context of querying a genomic sequence in a compacted de Bruijn graph, the objective is to evaluate the presence of the sequence by examining its constituent k-mers. A read R is decomposed into $|R|-k+1$ overlapping k-mers, each of length k. To determine the presence of R within the graph, we check the presence of each k-mer in the compacted de Bruijn graph. The read R is considered present if at least $\theta \times (|R|-k+1)$ of its k-mers are found in the graph with $0 < \theta \leq 1$ being a user defined threshold. If fewer than $\theta \times (|R| - k + 1)$ of the k-mers are present, the read is considered as absent.

4.2.2 Streaming queries

In some cases, some consecutive k-mers of a queried sequence q may appear in consecutive order of a queried unitig. Rather than determining the unitig for each k-mer individually, retrieving the corresponding k-mer y of each queried k-mer x, and comparing x to y, we opted for a streaming approach taking advantages of the presence of some consecutive k-mers of a query q in the same unitig.
To find if a k-mer x belong to the graph, we use the index described in chapter 3. First we find the canonical form y of x. We find the minimizer m of y which points to the bucket b in which x may appear. We then find the position tuple $< u_{id}, u_{off}, orientation >$ thanks to $f(y)$ where f is the minimal perfect hash function computed over the k-mers of b. Finally, we compare y to the canonical form of the k-mer that appear in the unitig u whose id is u_{id} at offset u_{off}.

Now the streaming of query is done by keeping track of the unitig u and the offset u_{off} in u where the first present k-mer x of q appear. For the successor k-mer z of x in q, we find its unitig u'_{id} and its offset u'_{off}. If $u_{id} = u'_{id}$, we check then if whether $u'_{off} = u_{off} + 1$ or $u'_{off} = u_{off} - 1$. Finally we compare z to the k-mer retrieved by shifting one character to the right or the left depending on u'_{off}. In the case when $u'_{id} \neq u_{id}$, the unitig and the offset are updated. Additionally, if $u_{id} = u'_{id}$ but $u'_{off} \neq u_{off} + 1$ or $u'_{off} \neq u_{off} - 1$, then only the offset is updated.

This approach reduces the complexity of checking the presence of a k-mer from $o(k)$ to $o(1)$ when streaming consecutive k-mer queries.

4.2.3 Pseudo-mapping of sequences in Cdbgtricks

In addition to reporting the presence or absence of a queried sequence q, we provided the ability to retrieve the set of nodes and the offsets of the present k-mers of q. This is done by reporting the set of uni-MEM of q as defined in [64]. Each uni-MEM is a tuple $< u_{id}, u_{start}, u_{end}, q_{start}, q_{end} >$ where u_{start} and u_{end} are the start and end positions of mapping on the unitig whose identifier is u_{id}, and q_{start} and q_{end} are the start and end positions of mapping on the queried sequence q. In other words, the k-mers whose offsets in the read are between q_{start} and q_{end} are found in the unitig u_{id} between u_{start} and u_{end}.
A uni-MEM is found through the extension of the first common k-mer between the read and a unitig. The extension ends in one of the following cases:

1. A mismatch is encountered.
2. Either the end of the read or the end of the unitig is encountered.

4.3 Results

All presented results are reproducible using command lines and versions of tested tools, that are given in this repository `https://github.com/khodor14/Cdbgtricks_experiments`. The executions were performed on the GenOuest platform on a node with 4×8 cores Xeon E5-2660 2,20 GHz with 128 GB of memory.

4.3.1 Genome datasets

We tested Cdbgtricks in two frameworks, corresponding to two input datasets of distinct size and complexity. The first one, called "*human*" is composed of 10 assembled human genomes that were used in the GGCAT experiments [1]. These genomes are available on zenodo (`10.5281/zenodo.7506049,10.5281/zenodo.7506425`). The second set, called "*E. coli*" is composed of 15,806 *E. coli* genomes downloaded from NCBI (`https://ftp.ncbi.nlm.nih.gov/genomes/all/GCF/030/`).

4.3.2 Results querying sequences

We propose some experiments for comparing the query performances of Cdbgtricks with those of Bifrost, GGCAT and SSHash. Note that these four tools do not offer the same query features. Bifrost and GGCAT can report as well the genomes in which the k-mers of the query appear if the graph is constructed in color mode. However, in the context of this experiment, they were executed without colors. These results must be considered as rough estimations showing the main tendencies.

Using Bifrost, we constructed a graph from 15,806 *E. coli* genomes, and a graph from 10 human genomes. Then we constructed an index for each graph using either Cdbgtricks, Bifrost or SSHash. We have differentiated results obtained with positive queries (querying sequences present in the graph) and those obtained with negative queries (querying random sequences) with k-mers that are not in the two constructed graphs. The positive queries are a subset of unitigs from each graph. The negative queries are

composed of one million random sequences of length between 500 and 1000 base pairs. The querying results are shown Table 4.1.

Table 4.1 – **Performances of sequence queries using a compacted de Bruijn graph for Cdbgtricks, Bifrost, SSHash, and GGCAT**

Dataset	Query type	Tool	Memory (MB)	Disk (MB)	time (mm:ss)
E. coli	Negative	Cdbgtricks	4723	0	10:02
		Bifrost	4362	0	06:43
		SSHash	725	0	00:07
		GGCAT	560	3325	01:32
	Positive	Cdbgtricks	4724	0	02:15
		Bifrost	4362	0	01:43
		SSHash	725	0	01:10
		GGCAT	644	2978	01:26
human	Negative	Cdbgtricks	25520	0	12:25
		Bifrost	27376	0	11:37
		SSHash	6090	0	00:07
		GGCAT	615	6861	4:55
	Positive	Cdbgtricks	25520	0	04:23
		Bifrost	27376	0	06:37
		SSHash	6090	0	01:14
		GGCAT	746	7053	05:04

The results shows that Cdbgtricks and Bifrost obtained similar results in term of memory, while Cdbgtricks is slightly slower. While GGCAT is faster than Bifrost and Cdbgtricks, and it uses the smallest amount of memory in these query experiments, it uses few Gigabytes of disk. Despite SSHash being the fastest tool, it consumes an order of magnitude more memory than GGCAT in the case of *human* dataset. These results shows that Cdbgtricks offer queries in a reasonable amount of time, and the performance of Cdbgtricks is close to the performance of Bifrost.

4.4 Conclusion

In this chapter, we showed a different use of the index of Cdbgtricks. Although some of the state of the art tools are more performant than Cdbgtricks in querying a compacted de Bruijn graph, Cdbgtricks, being a dynamic method, is able to query the graph in a reasonable amount of time. We presented as well that different types of queries can be performed depending on the biological applications. The choice of the type of query needed determine the complexity of the data structure and the algorithm. Cdbgtricks affords to report the presence or absence of the queried sequences as well as the positions of their k-mers in the graph.

Chapter 5

Conclusion and Perspectives

5.1 Conclusion

As a consequence of the advancement and the affordability of sequencing, the size of DNA databases is growing exponentially. With this drastic growth, there is always a need to query these databases in a time- and memory-efficient manner. This highlights the need for efficient and dynamic data structures to handle such amount of data. The main research question that we explored in this thesis is to tackle the mutability of a compacted de Bruijn graph.

The reasons for providing a dynamic data structures are mainly to be able to incorporate in a memory and time-efficient manners new sequenced data to an already built structure. This allows for incremental updates once the data are generated. The second reason is that current data structures scales linearly in terms of construction time with the size of the input data, which points to the problem of paying the construction time for every sequenced samples as the structure should be re-constructed from scratch.

We explored providing a method to update a compacted de Bruijn graph taking advantage of kmricks [56], a tool that provides an efficient method to offer k-mer set operations. We developed a novel method for indexing a compacted de Bruijn graph, based on the assumption that in large graphs, newly added genomes contribute only a small subset of distinct k-mers. Consequently, only a small portion of the graph and its index require updates. We allowed as well the update of the graph index which can serve for future update on the graph and to perform k-mer queries against the graph. The fruit of this work is an open source software Cdbgtricks. Experimental evaluations on a dataset composed of a hundred human genomes and a larger dataset composed of thousands of *E. coli* genomes show use-cases where Cdbgtricks offers better scalability than the state of the art tools. All the details about the methods and the experiments are provided in chapter 3.

Taking advantage of the proposed index, we implemented two additional functionalities in Cdbgtricks. The first one provides the ability to query reads against the graph. The output of such a functionality is the fact that either the read is present if it shares a user input percentage of k-mers with the graph, or it is considered as absent otherwise. The second functionality associates additional information to each queried read. For each read query, it reports the set of unitig maximal exact matches (uni-MEM) [64]. Experimental evaluations showed that our query method did not provide improvement with respect to the state of the art methods, but it showed that queries in Cdbgtricks are answered in a reasonable amount of time. The implementation details and the experiments are presented in chapter 4.

5.2 Contribution

The work of this thesis leads to the following paper with an open source code software.

Cdbgtricks: Strategies to update a compacted de Bruijn graph: The paper [41] presented a novel method to update a compacted de Bruijn graph. Cdbgtricks was accepted and presented at the Prague Stringology Conference [1].
Cdbgtricks is shown to be faster than GGCAT and Bifrost on a dataset of 100 human genomes and a set of thousands of *E.coli* genomes. The index proposed in Cdbgtricks offers the possibility of reporting the sub-graph that shares a desired percentage of k-mers with a query sequence with the ability to query a set of reads. The open source code of Cdbgtricks is available at `https://github.com/khodor14/Cdbgtricks` and all the details about the datasets and the commands used in the experiments are available at `https://github.com/khodor14/Cdbgtricks_experiments`.

5.3 Future of this work and perspectives

The contribution presented in this manuscript can be continued in different research directions, by proposing new applications or integrating other data to the proposed structure. These perspectives are presented here.

1. https://www.stringology.org/event/2024/

5.3.1 Short term perspectives

Colored and Compacted de Bruijn graph

The key advantage over a compacted de Bruijn graph is that for every k-mer, the identifiers (the color) to the sequences to which it belongs is maintained. Many efforts were made to compress the storage of colors [1, 36, 80]. In the current implementation of Cdbgtricks, the color information are not taken into account. One of the future contribution would be to implement an algorithm that efficiently updates the colors of a colored and compacted de Bruijn graph. This will necessitate minor yet potentially expensive operations. In the current implementation, we reduce the problem from adding a new genome to adding a set of new k-mers that have not been added yet to the graph. In the colored implementation, we have to take into account the set of shared k-mers between the new genome. The set of shared k-mers indicates the color sets that must be updated, i.e., the identifier of the new genome should be added to these sets. Additionally, a new color set that points only to the new genome is assigned to the new k-mers.

Sequence to graph distance estimation

Sequence comparison helps identify structural, functional and or evolutionary relationships between two or more sequences [97]. Some proposed methods in the literature work on pairwise sequence alignment where they can estimate the distance between two sequences [57], while other methods provide read to graph sequence alignment [84]. These methods are suitable for finding variants and estimate how similar two sequences are. However the limitation of these methods is that either they do not scale to align a sequence to the graph, or they are not able to capture all the variants. Other approaches uses the Jaccard index which is simply the following equation:

$$J(A, B) = \frac{|A \cap B|}{|A \cup B|} \tag{5.1}$$

Where A and B are two sets of words. In the context of sequence comparison, A and B designate two sets of k-mers. This metric divides the number of common k-mers between sample A and sample B to the total number of distinct k-mers in A and B. A value

close to one suggests that the two samples are similar, while a value close to zero signifies that they are different. Mash [76] uses a subsampling approach to estimate the Jaccard index between two genomes. While this approach fits two genomes, it cannot be applied to estimate the distance between a genome and a graph or two graphs. For example, if we take a graph of *E. coli* that contains 12×10^7 k-mers and a genome of *E. coli* that has 3×10^6 k-mers, and suppose that the genome shares 10^6 k-mers with the graph, then the estimated Jaccard index would be $J = 0.008$ which suggests that the genome and the graph are dissimilar while they are coming from the same species. The larger the graph size in terms of the number of k-mers, the smaller the estimated value of the index. Sourmash [50] uses the containment index to estimate the distance between the sequence and the graph. The containment index is defined in the following formula:

$$C(A, B) = \frac{|A \cap B|}{|A|} \tag{5.2}$$

It measures the proportion of A that lay in B. This approach is used to provide more accurate estimatation of the distance when one of the samples is very large in size compared to the second one. In the previous example, $\frac{1}{3}$ of the *E. coli* genome lay in the graph. When adding new sequences to the graph some untigs are split into smaller unitigs, and some new unitigs are created. This points to the fact that the topology of the graph gets changed. We can use the change in topology of the graph to estimate the distance of the sequences to the graph or between two graphs. One could simply consider the ratio of conserved (unchanged) unitigs to the total number of unitigs.

Optimizing k-mer partitioning

Although the experiments in Cdbgtricks show that a small number of buckets are modified for large graphs, which suggests that it is a time-efficient structure, yet the structure can be more optimized. The current structure groups the buckets of k-mers based on their sizes to guarantee balanced super-buckets each of size $\gamma\rho$ approximately. This strategy does not take into account that two consecutive and different minimizers may be grouped together. In such a case, querying k-mers of different minimizers may lead to cache misses as the k-mers are located in different buckets. A cache miss happens when the system asks to retrieve data from the cache memory, but the data is not in the cache. Grouping buckets of k-mers not only based on their sizes, but also on the locality of the minimizers they share results in indexing consecutive k-mers with different minimizers in

the same bucket which minimize the number of cache misses at query time. To implement this, we need to add a constraint during the phase of creating super-buckets. We need to ensure that the minimizers of the k-mers appear in the same unitigs. This may not only minimize the number of cache misses, but also may minimize the number of buckets that should be updated because consecutive k-mers of different minimizers should be added now to the bucket instead of adding them to different buckets.

Dynamic minimal perfect hash function

We have seen a minimal perfect hash function (MPHF) can reduce the memory storage of a set of keys compared to storing the keys in a hash table. However, a limitation of an MPHF is that it is static. One cannot add new keys to the MPHF after its construction. Recall that an MPHF f maps bijectively a set of n keys to the set $\{i|\ i \in \mathbb{N}\ and\ 0 \leq i < n\}$. Dynamicity in this context is that after adding m new keys, the MPHF f will map bijectively the set of $n+m$ keys to the set of $\{i|\ i \in \mathbb{N}\ and\ 0 \leq i < n+m\}$. Bentley and Saxe [12] proposed an algorithm to transform a static data structure into dynamic one. The key idea is that given a static data structure that contains n items, it gets decomposed into $l = \log_2 n$ levels. Each level l_i has 2^i items. To insert an item into the structure, a value r is determined such that l_r is empty. The item is inserted in l_r, and all the levels from l_0 to l_{r-1} are merged into l_r, and then they are emptied. The algorithm was adopted in Dynamic Mantis [7] to implement a dynamic index. We can take advantage of this algorithm to provide a dynamic minimal perfect hash function. The set of n keys is partitioned into l buckets. The number of buckets is the same as the number of distinct minimizers. Each bucket b_i has n_i keys. To each bucket b_i, we associate an MPHF f_i. To determine the hash value of an item x in the interval $[0; n[$, we need to store the cumulative sum of the sizes in a vector S where $S[0] = n_0$, $S[1] = n_0 + n_1$ and so on. Given a key x that belong to bucket b_i, its hash value in the interval $[0; n_i[$ is determined thanks to $h_x = f_i(x)$. If $i = 0$, then h_x is returned, otherwise we return $h_x + S[i]$. To insert a new element y, we determine first to which bucket b_i it should be assigned. We re-compute then f_i and update S accordingly.

5.3.2 Long term perspectives

Updating a compacted de Bruijn graph constructed on a larger alphabet

Most of methods for constructing the de Bruijn graph or its compacted version have traditionally operated on input sequences in the base space, using the classical DNA alphabet $\sigma = A, C, G, T$. However, mdBG [34] introduced an algorithm to construct the graph in the minimizer space σ^m, where m is the size of the minimizer (figure 5.1). The method proposed the concept of k-min-mer which is a k-mer in the minimizer space (the concatenation of k minimizers of size m). The mdBG tool generally ignores small variants, but it captures heterozygosity which causes branching in the graph [9].

A future direction of our work is to implement an algorithm to update a compacted de

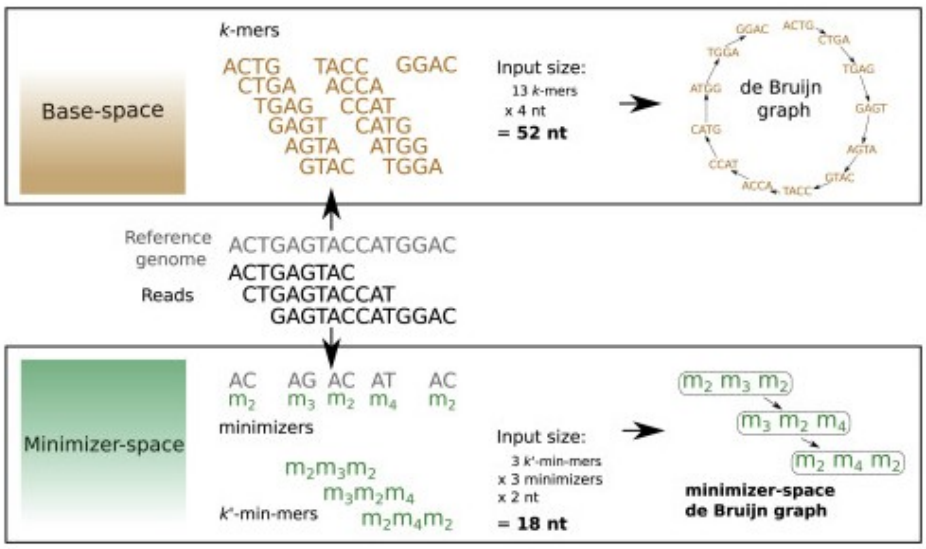

Figure 5.1 – The de Bruijn graph in the base space and in the minimizer space. Figure is taken from [34].

Bruijn graph constructed on the alphabet of minimizers. We need first to identify the set of new k-min-mers that belong to the new genome and that are absent in the graph. we can use the compaction algorithm of mdBG to compact consecutive k-min-mers if the $(k-1)$-min-mer suffix of the first is the $(k-1)$-min-mer prefix of the second. This requires performing $2 \times \sigma^m$ queries, which must be efficiently managed to ensure optimal performance.

Partitioning a compacted de Bruijn graph

When we need to update a compacted de Bruijn graph or to query k-mers against it, the graph is totally loaded in the memory. One of the perspectives could be to implement a graph partitioning algorithm which partitions the graph into several sub-graphs. By implementing such a strategy, the overall memory requirement can be significantly reduced. Each partition can be processed independently, allowing the use of memory more efficiently. Partitioning the graph will facilitate the parallel processing of the sub-graphs. This should compromise the time and memory usage. Partitioning a compacted de Bruijn graph makes it possible to distribute the subgraphs across different nodes in a distributed computing environment. It also facilitates visualizing large graphs as sub-graphs can be loaded independently. A minimizer based approach can be used to partition the graph. Each unitig has a set of minimizers. Different unitgis may share subsets of their minimizer sets. We can group unitigs in the same partition in such a way to maximise the number of unitigs inside a partition and to maximise the number of shared minimizers. We need also to ensure that partitions are balanced.

BIBLIOGRAPHY

[1] Cracco A. and Tomescu A., « Extremely-fast construction and querying of compacted and colored de Bruijn graphs with GGCAT », *in*: *bioRxiv* (2022), DOI: https://doi.org/10.1101/2022.10.24.513174.

[2] Jarno Alanko et al., « Buffering updates enables efficient dynamic de Bruijn graphs », *in*: *Computational and Structural Biotechnology Journal* 19 (2021), pp. 4067–4078, ISSN: 2001-0370, DOI: https://doi.org/10.1016/j.csbj.2021.06.047, URL: https://www.sciencedirect.com/science/article/pii/S2001037021002853.

[3] Jarno N. Alanko, Simon J. Puglisi, and Jaakko Vuohtoniemi, « Succinct k-mer Set Representations Using Subset Rank Queries on the Spectral Burrows-Wheeler Transform (SBWT) », *in*: *bioRxiv* (2022), DOI: 10.1101/2022.05.19.492613, URL: https://www.biorxiv.org/content/early/2022/05/20/2022.05.19.492613.

[4] Bahar Alipanahi et al., « Succinct dynamic de Bruijn graphs », *in*: *Bioinformatics* 37.*14* (July 2021), pp. 1946–1952, ISSN: 1367-4803, DOI: 10.1093/bioinformatics/btaa546, eprint: https://academic.oup.com/bioinformatics/article-pdf/37/14/1946/50578828/btaa546_supplementary_data.pdf, URL: https://doi.org/10.1093/bioinformatics/btaa546.

[5] Fatemeh Almodaresi, Mohsen Zakeri, and Rob Patro, « PuffAligner: a fast, efficient and accurate aligner based on the Pufferfish index », *in*: *Bioinformatics* 37.*22* (June 2021), pp. 4048–4055, ISSN: 1367-4803, DOI: 10.1093/bioinformatics/btab408, eprint: https://academic.oup.com/bioinformatics/article-pdf/37/22/4048/50335370/btab408.pdf, URL: https://doi.org/10.1093/bioinformatics/btab408.

[6] Fatemeh Almodaresi et al., « A space and time-efficient index for the compacted colored de Bruijn graph », *in*: *Bioinformatics* 34.*13* (June 2018), pp. i169–i177, ISSN: 1367-4803, DOI: 10.1093/bioinformatics/bty292, eprint: https://academic.oup.com/bioinformatics/article-pdf/34/13/i169/50316242/bioinformatics_34_13_i169.pdf, URL: https://doi.org/10.1093/bioinformatics/bty292.

[7] Fatemeh Almodaresi et al., « An incrementally updatable and scalable system for large-scale sequence search using the Bentley–Saxe transformation », *in*: *Bioinformatics* 38.*12* (Mar. 2022), pp. 3155–3163, ISSN: 1367-4803, DOI: 10.1093/bioinformatics/btac142, eprint: https://academic.oup.com/bioinformatics/article-pdf/38/12/3155/49885664/btac142_supplementary_data.pdf, URL: https://doi.org/10.1093/bioinformatics/btac142.

[8] Francesco Andreace et al., « Comparing methods for constructing and representing human pangenome graphs », *in*: *Genome Biology* (Nov. 2023), DOI: 10.1186/s13059-023-03098-2, URL: https://doi.org/10.1186/s13059-023-03098-2.

[9] Francesco Andreace et al., « Comparing methods for constructing and representing human pangenome graphs », *in*: *Genome Biology* 24.*1* (Nov. 2023), p. 274, ISSN: 1474-760X, DOI: 10.1186/s13059-023-03098-2, URL: https://doi.org/10.1186/s13059-023-03098-2.

[10] Jasmijn A. Baaijens et al., « Computational graph pangenomics: a tutorial on data structures and their applications », *in*: *Natural Computing* 21.*1* (Mar. 2022), pp. 81–108, ISSN: 1572-9796, DOI: 10.1007/s11047-022-09882-6.

[11] Djamal Belazzougui et al., « Fully Dynamic de Bruijn Graphs », *in*: *String Processing and Information Retrieval*, ed. by Shunsuke Inenaga, Kunihiko Sadakane, and Tetsuya Sakai, Cham: Springer International Publishing, 2016, pp. 145–152, ISBN: 978-3-319-46049-9.

[12] Jon Louis Bentley and James B Saxe, « Decomposable searching problems I. Static-to-dynamic transformation », *in*: *Journal of Algorithms* 1.*4* (1980), pp. 301–358, ISSN: 0196-6774, DOI: https://doi.org/10.1016/0196-6774(80)90015-2, URL: https://www.sciencedirect.com/science/article/pii/0196677480900152.

[13] Burton H. Bloom, « Space/time trade-offs in hash coding with allowable errors », *in*: *Commun. ACM* 13.*7* (July 1970), pp. 422–426, ISSN: 0001-0782, DOI: 10.1145/362686.362692.

[14] Alexander Bowe et al., « Succinct de Bruijn Graphs », *in*: *Algorithms in Bioinformatics*, ed. by Ben Raphael and Jijun Tang, Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 225–235, ISBN: 978-3-642-33122-0.

[15] Nicolas L Bray et al., « Near-optimal probabilistic RNA-seq quantification », *in*: *Nature Biotechnology* 34.*5* (May 2016), pp. 525–527, ISSN: 1546-1696, DOI: 10.1038/nbt.3519, URL: https://doi.org/10.1038/nbt.3519.

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

[17] Michael Burrows et al., « A Block-sorting Lossless Data Compression Algorithm », *in*: 1994, URL: https://api.semanticscholar.org/CorpusID:2167441.

[18] Yeow Meng Chee et al., « Locally-Constrained de Bruijn Codes: Properties, Enumeration, Code Constructions, and Applications », *in*: *IEEE Transactions on Information Theory* 67.*12* (2021), pp. 7857–7875, DOI: 10.1109/TIT.2021.3112300.

[19] Rayan Chikhi, Antoine Limasset, and Paul Medvedev, « Compacting de Bruijn graphs from sequencing data quickly and in low memory », *in*: *Bioinformatics* 32.*12* (June 2016), pp. i201–i208, ISSN: 1367-4803, DOI: 10.1093/bioinformatics/btw279.

[20] Rayan Chikhi and Guillaume Rizk, « Space-efficient and exact de Bruijn graph representation based on a Bloom filter », *in*: *Algorithms for Molecular Biology* 8.*1* (Sept. 2013), p. 22, ISSN: 1748-7188, DOI: 10.1186/1748-7188-8-22.

[21] Rayan Chikhi et al., « On the Representation of De Bruijn Graphs », *in*: *Journal of Computational Biology* 22.*5* (2015), PMID: 25629448, pp. 336–352, DOI: 10.1089/cmb.2014.0160, URL: https://doi.org/10.1089/cmb.2014.0160.

[22] Rayan Chikhi et al., « On the Representation of de Bruijn Graphs », *in*: *Research in Computational Molecular Biology*, ed. by Roded Sharan, Cham: Springer International Publishing, 2014, pp. 35–55.

[23] Thomas C. Conway and Andrew J. Bromage, « Succinct data structures for assembling large genomes », *in*: *Bioinformatics* 27.*4* (Jan. 2011), pp. 479–486, ISSN: 1367-4803, DOI: 10.1093/bioinformatics/btq697.

[24] Thomas H. Cormen et al., *Introduction to Algorithms*, 3rd, Cambridge, MA: The MIT Press, 2009, ISBN: 978-0-262-03384-8.

[25] Andrea Cracco and Alexandru Tomescu, *Human genomes for GGCAT benchmarks - part 1*, Zenodo, Jan. 2023, DOI: 10.5281/zenodo.7506049, URL: https://doi.org/10.5281/zenodo.7506049.

[26] Andrea Cracco and Alexandru Tomescu, *Human genomes for GGCAT benchmarks - part 2*, Zenodo, Jan. 2023, DOI: 10.5281/zenodo.7506425, URL: https://doi.org/10.5281/zenodo.7506425.

[27] Victoria G Crawford et al., « Practical dynamic de Bruijn graphs », *in*: *Bioinformatics* 34.*24* (June 2018), pp. 4189–4195, ISSN: 1367-4803, DOI: 10.1093/bioinformatics/bty500, eprint: https://academic.oup.com/bioinformatics/article-pdf/34/24/4189/48919718/bioinformatics_34_24_4189.pdf, URL: https://doi.org/10.1093/bioinformatics/bty500.

[28] Ralf Dahm, « Friedrich Miescher and the discovery of DNA », *in*: *Developmental Biology* 278.*2* (2005), pp. 274–288, ISSN: 0012-1606, DOI: https://doi.org/10.1016/j.ydbio.2004.11.028, URL: https://www.sciencedirect.com/science/article/pii/S0012160604008231.

[29] Sebastian Deorowicz, Agnieszka Debudaj-Grabysz, and Szymon Grabowski, « Disk-based k-mer counting on a PC », *in*: *BMC Bioinformatics* 14.*1* (May 2013), p. 160, DOI: 10.1186/1471-2105-14-160.

[30] Sebastian Deorowicz et al., « KMC 2: fast and resource-frugal k-mer counting », *in*: *Bioinformatics* 31.*10* (Jan. 2015), pp. 1569–1576, ISSN: 1367-4803, DOI: 10.1093/bioinformatics/btv022.

[31] Diego Díaz-Domínguez, Miika Leinonen, and Leena Salmela, « Space-efficient computation of k-mer dictionaries for large values of k », *in*: *Algorithms for Molecular Biology* 19.*1* (Apr. 2024), p. 14, DOI: 10.1186/s13015-024-00259-1, URL: https://doi.org/10.1186/s13015-024-00259-1.

[32] Keith Dufault-Thompson and Xiaofang Jiang, « Applications of de Bruijn graphs in microbiome research », *in*: *iMeta* 1 (Mar. 2022), DOI: 10.1002/imt2.4.

[33] Jordan M. Eizenga et al., « Pangenome Graphs », *in*: *Annual Review of Genomics and Human Genetics* 21.*1* (2020), PMID: 32453966, pp. 139–162, DOI: 10.1146/annurev-genom-120219-080406, eprint: https://doi.org/10.1146/annurev-genom-120219-080406, URL: https://doi.org/10.1146/annurev-genom-120219-080406.

[34] Barış Ekim, Bonnie Berger, and Rayan Chikhi, « Minimizer-space de Bruijn graphs », *in*: *bioRxiv* (2021), DOI: 10.1101/2021.06.09.447586.

[35] Emmanuel Esposito, Thomas Mueller Graf, and Sebastiano Vigna, « RecSplit: Minimal Perfect Hashing via Recursive Splitting », *in*: *2020 Proceedings of the Symposium on Algorithm Engineering and Experiments (ALENEX)*, pp. 175–185, DOI: 10.1137/1.9781611976007.14, eprint: https://epubs.siam.org/doi/pdf/10.1137/1.9781611976007.14, URL: https://epubs.siam.org/doi/abs/10.1137/1.9781611976007.14.

[36] Jason Fan et al., « Fulgor: A fast and compact k-mer index for large-scale matching and color queries », *in*: *Algorithms for Molecular Biology* 19 (2024), DOI: 10.1186/s13015-024-00251-9, URL: https://doi.org/10.1186/s13015-024-00251-9.

[37] Travis Gagie, Giovanni Manzini, and Jouni Sirén, « Wheeler graphs: A framework for BWT-based data structures », *in*: *Theoretical Computer Science* 698 (2017), Algorithms, Strings and Theoretical Approaches in the Big Data Era (In Honor of the 60th Birthday of Professor Raffaele Giancarlo), pp. 67–78, ISSN: 0304-3975, DOI: https://doi.org/10.1016/j.tcs.2017.06.016, URL: https://www.sciencedirect.com/science/article/pii/S0304397517305285.

[38] Guillaume Gautreau et al., « PPanGGOLiN: Depicting microbial diversity via a partitioned pangenome graph », *in*: *PLOS Computational Biology* 16.*3* (Mar. 2020), pp. 1–27, DOI: 10.1371/journal.pcbi.1007732.

[39] Hongzhe Guo et al., « deGSM: Memory Scalable Construction Of Large Scale de Bruijn Graph », *in*: *IEEE/ACM Transactions on Computational Biology and Bioinformatics* 18.*6* (2021), pp. 2157–2166, DOI: 10.1109/TCBB.2019.2913932.

[40] Mihail R. Halachev, Nicholas J. Loman, and Mark J. Pallen, « Calculating Orthologs in Bacteria and Archaea: A Divide and Conquer Approach », *in*: *PLOS ONE* 6.*12* (2011), e28388, DOI: 10.1371/journal.pone.0028388.

[41] Khodor Hannoush, Camille Marchet, and Pierre Peterlongo, « Cdbgtricks: strategies to update a compacted de Bruijn graph », *in*: *Proceedings of the Prague Stringology Conference 2024 (PSC 2024)*, Czech Technical University in Prague, 2024, pp. 202–205, URL: https://psc.fit.cvut.cz/event/2024/.

[42] Stefan Hermann et al., « PHOBIC: Perfect Hashing with Optimized Bucket Sizes and Interleaved Coding », *in*: *arXiv* (2024), eprint: 2404.18497, URL: https://arxiv.org/abs/2404.18497.

[43] Guillaume Holley and Páll Melsted, « Bifrost: highly parallel construction and indexing of colored and compacted de Bruijn graphs », *in*: *Genome Biology* 21.*1* (Sept. 2020), p. 249, ISSN: 1474-760X, DOI: 10.1186/s13059-020-02135-8.

[44] JE Hopcroft, *Introduction to Automata Theory, Languages, and Computation*, 2001.

[45] Samuel T. Horsfield, Nicholas J. Croucher, and John A. Lees, « Accurate and fast graph-based pangenome annotation and clustering with ggCaller », *in*: *Genome Research* (2023), DOI: 10.1101/gr.277733.123, URL: https://genome.cshlp.org/content/33/9/1622.short.

[46] Bin Hou, Rongshu Wang, and Jianhua Chen, « Long Read Error Correction Algorithm Based on the de Bruijn Graph for the Third-generation Sequencing », *in*: *2021 4th International Conference on Information Communication and Signal Processing (ICICSP)*, 2021, pp. 616–620, DOI: 10.1109/ICICSP54369.2021.9611869.

[47] Shien Huang et al., « A Comprehensive Review of the de Bruijn Graph and Its Interdisciplinary Applications in Computing », *in*: *Engineered Science* 28 (2024), p. 1061, ISSN: 2576-9898, DOI: 10.30919/es1061.

[48] Zamin Iqbal et al., « De novo assembly and genotyping of variants using colored de Bruijn graphs », *in*: *Nature Genetics* 44 (2012), pp. 226–232, ISSN: 1546-1718, DOI: 10.1038/ng.1028, URL: https://doi.org/10.1038/ng.1028.

[49] Zamin Iqbal et al., « De novo assembly and genotyping of variants using colored de Bruijn graphs », *in*: *Nature Genetics* 44.*2* (Feb. 2012), pp. 226–232, ISSN: 1546-1718, DOI: 10.1038/ng.1028, URL: https://doi.org/10.1038/ng.1028.

[50] Luiz Irber et al., « Lightweight compositional analysis of metagenomes with FracMinHash and minimum metagenome covers », *in*: *bioRxiv* (2022), DOI: 10.1101/2022.01.11.475838, eprint: https://www.biorxiv.org/content/early/2022/01/17/2022.01.11.475838.full.pdf, URL: https://www.biorxiv.org/content/early/2022/01/17/2022.01.11.475838.

[51] B. G. Jackson et al., « Parallel de novo assembly of large genomes from high-throughput short reads », *in*: *"2010 IEEE International Symposium on Parallel and Distributed Processing (IPDPS)"*, 2010, pp. 1–10, DOI: 10.1109/IPDPS.2010.5470397.

[52] M. Frans Kaashoek and David R. Karger, « Koorde: A Simple Degree-Optimal Distributed Hash Table », *in*: *Peer-to-Peer Systems II*, ed. by M. Frans Kaashoek and Ion Stoica, Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, pp. 98–107, ISBN: 978-3-540-45172-3.

[53] Jamshed Khan and Rob Patro, « Cuttlefish: fast, parallel and low-memory compaction of de Bruijn graphs from large-scale genome collections », *in*: *Bioinformatics* 37.*Supplement*$_1$ (July 2021), pp. i177–i186, ISSN: 1367-4803, DOI: 10.1093/bioinformatics/btab309, eprint: https://academic.oup.com/bioinformatics/article-pdf/37/Supplement_1/i177/50694391/btab309.pdf, URL: https://doi.org/10.1093/bioinformatics/btab309.

[54] Jamshed Khan et al., « Scalable, ultra-fast, and low-memory construction of compacted de Bruijn graphs with Cuttlefish 2 », *in*: *Genome biology* 23.*1* (2022), p. 190, DOI: https://doi.org/10.1186/s13059-022-02743-6.

[55] Marek Kokot, Maciej Długosz, and Sebastian Deorowicz, « KMC 3: counting and manipulating k-mer statistics », *in*: *Bioinformatics* 33.*17* (May 2017), pp. 2759–2761, ISSN: 1367-4803, DOI: 10.1093/bioinformatics/btx304.

[56] Téo Lemane et al., « kmtricks: efficient and flexible construction of Bloom filters for large sequencing data collections », *in*: *Bioinformatics Advances* 2.*1* (Apr. 2022).

[57] Heng Li, « Minimap2: pairwise alignment for nucleotide sequences », *in*: *Bioinformatics* 34.*18* (May 2018), pp. 3094–3100, ISSN: 1367-4803, DOI: 10.1093/bioinformatics/bty191, eprint: https://academic.oup.com/bioinformatics/article-pdf/34/18/3094/48919122/bioinformatics_34_18_3094.pdf, URL: https://doi.org/10.1093/bioinformatics/bty191.

[58] Ruiqiang Li et al., « De novo assembly of human genomes with massively parallel short read sequencing », *in*: *Genome research* 20 (Dec. 2009), pp. 265–72, DOI: 10.1101/gr.097261.109.

[59] Antoine Limasset, Jean-François Flot, and Pierre Peterlongo, « Toward perfect reads: self-correction of short reads via mapping on de Bruijn graphs », *in*: *Bioinformatics* 36.*5* (Feb. 2019), pp. 1374–1381, ISSN: 1367-4803, DOI: 10.1093/bioinformatics/btz102, eprint: https://academic.oup.com/bioinformatics/article-pdf/36/5/1374/50552810/bioinformatics_36_5_1374.pdf, URL: https://doi.org/10.1093/bioinformatics/btz102.

[60] Antoine Limasset, Jean-François Flot, and Pierre Peterlongo, « Toward perfect reads: self-correction of short reads via mapping on de Bruijn graphs », *in*: *Bioinformatics* 36.*5* (Feb. 2019), pp. 1374–1381, DOI: 10.1093/bioinformatics/btz102, URL: https://doi.org/10.1093/bioinformatics/btz102.

[61] Antoine Limasset et al., « Fast and Scalable Minimal Perfect Hashing for Massive Key Sets », *in*: *16th International Symposium on Experimental Algorithms (SEA 2017)*, Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2017.

[62] Antoine Limasset et al., *Fast and scalable minimal perfect hashing for massive key sets*, 2017, arXiv: 1702.03154 [cs.DS], URL: https://arxiv.org/abs/1702.03154.

[63] Antoine Limasset et al., « Read mapping on de Bruijn graphs », *in*: *BMC Bioinformatics* (2016), DOI: 10.1186/s12859-016-1103-9, URL: https://doi.org/10.1186/s12859-016-1103-9.

[64] Bo Liu et al., « deBGA: read alignment with de Bruijn graph-based seed and extension », *in*: *Bioinformatics* 32.*21* (July 2016), pp. 3224–3232, ISSN: 1367-4803, DOI: 10.1093/bioinformatics/btw371, eprint: https://academic.oup.com/bioinformatics/article-pdf/32/21/3224/49021959/bioinformatics_32_21_3224.pdf, URL: https://doi.org/10.1093/bioinformatics/btw371.

[65] Paul G. Livingstone, Russell M. Morphew, and David E. Whitworth, « Genome Sequencing and Pan-Genome Analysis of 23 Corallococcus spp. Strains Reveal Unexpected Diversity, With Particular Plasticity of Predatory Gene Sets », *in*: *Frontiers in Microbiology* 9 (2018), ISSN: 1664-302X, DOI: 10.3389/fmicb.2018.03187.

[66] Nina Luhmann, Guillaume Holley, and Mark Achtman, « BlastFrost: fast querying of 100,000s of bacterial genomes in Bifrost graphs », *in*: *Genome Biology* 22.*1* (Jan. 2021), p. 30, DOI: 10.1186/s13059-020-02237-3, URL: https://doi.org/10.1186/s13059-020-02237-3.

[67] Guillaume Marçais and Carl Kingsford, « A fast, lock-free approach for efficient parallel counting of occurrences of k-mers », *in*: *Bioinformatics* 27.*6* (Jan. 2011), pp. 764–770, ISSN: 1367-4803, DOI: 10.1093/bioinformatics/btr011.

[68] Camille Marchet, Mael Kerbiriou, and Antoine Limasset, « BLight: efficient exact associative structure for k-mers », *in*: *Bioinformatics* 37.*18* (Apr. 2021), pp. 2858–2865, ISSN: 1367-4803, DOI: 10.1093/bioinformatics/btab217, eprint: https:

//academic.oup.com/bioinformatics/article-pdf/37/18/2858/50579086/btab217.pdf, URL: https://doi.org/10.1093/bioinformatics/btab217.

[69] Camille Marchet et al., « REINDEER: efficient indexing of k-mer presence and abundance in sequencing datasets », *in*: *Bioinformatics* 36.*Supplement_1* (July 2020), pp. i177–i185, ISSN: 1367-4803, DOI: 10.1093/bioinformatics/btaa487, eprint: https://academic.oup.com/bioinformatics/article-pdf/36/Supplement_1/i177/50967118/bioinformatics_36_supplement1_i177.pdf, URL: https://doi.org/10.1093/bioinformatics/btaa487.

[70] Igor Martayan et al., « Conway-Bromage-Lyndon (CBL): an exact, dynamic representation of k-mer sets », *in*: *bioRxiv* (2024), DOI: 10.1101/2024.01.29.577700.

[71] Duccio Medini et al., « The microbial pan-genome », *in*: *Current Opinion in Genetics Development* 15.*6* (2005), Genomes and evolution, pp. 589–594, ISSN: 0959-437X, DOI: https://doi.org/10.1016/j.gde.2005.09.006, URL: https://www.sciencedirect.com/science/article/pii/S0959437X05001759.

[72] Jason R. Miller, Sergey Koren, and Granger Sutton, « Assembly algorithms for next-generation sequencing data », *in*: *Genomics* 95.*6* (2010), pp. 315–327, ISSN: 0888-7543, DOI: https://doi.org/10.1016/j.ygeno.2010.03.001, URL: https://www.sciencedirect.com/science/article/pii/S0888754310000492.

[73] Ilia Minkin, Son Pham, and Paul Medvedev, « TwoPaCo: an efficient algorithm to build the compacted de Bruijn graph from many complete genomes », *in*: *Bioinformatics* 33.*24* (Sept. 2016), pp. 4024–4032, ISSN: 1367-4803, DOI: 10.1093/bioinformatics/btw609.

[74] National Center for Biotechnology Information, *National Center for Biotechnology Information*, 2024, URL: https://www.ncbi.nlm.nih.gov/.

[75] Sergey Nurk et al., « Assembling Genomes and Mini-metagenomes from Highly Chimeric Reads », *in*: *Research in Computational Molecular Biology*, ed. by Minghua Deng et al., Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 158–170.

[76] Brian D. Ondov et al., « Mash: fast genome and metagenome distance estimation using MinHash », *in*: *Genome Biology* 17.*1* (June 2016), p. 132, ISSN: 1474-760X, DOI: 10.1186/s13059-016-0997-x.

[77] Andrew J. Page et al., « Roary: rapid large-scale prokaryote pan genome analysis », *in*: *Bioinformatics* 31.*22* (July 2015), pp. 3691–3693, ISSN: 1367-4803, DOI: 10.1093/bioinformatics/btv421.

[78] Prashant Pandey et al., « Mantis: A Fast, Small, and Exact Large-Scale Sequence-Search Index », *in*: *Cell Systems* (2018), DOI: doi:10.1016/j.cels.2018.05.021, URL: https://doi.org/10.1016/j.cels.2018.05.021.

[79] Giulio Ermanno Pibiri, « Sparse and skew hashing of K-mers », *in*: *Bioinformatics* 38.*Supplement_1* (June 2022), pp. i185–i194, ISSN: 1367-4803, DOI: 10.1093/bioinformatics/btac245, eprint: https://academic.oup.com/bioinformatics/article-pdf/38/Supplement_1/i185/49887045/btac245.pdf, URL: https://doi.org/10.1093/bioinformatics/btac245.

[80] Giulio Ermanno Pibiri, Jason Fan, and Rob Patro, « Meta-colored Compacted de Bruijn Graphs », *in*: *Research in Computational Molecular Biology*, ed. by Jian Ma, Cham: Springer Nature Switzerland, 2024, pp. 131–146, ISBN: 978-1-0716-3989-4.

[81] Giulio Ermanno Pibiri, Yoshihiro Shibuya, and Antoine Limasset, « Locality-preserving minimal perfect hashing of k-mers », *in*: *Bioinformatics* 39.$Supplement_1$ (June 2023), pp. i534–i543, ISSN: 1367-4811, DOI: 10.1093/bioinformatics/btad219, eprint: https://academic.oup.com/bioinformatics/article-pdf/39/Supplement_1/i534/50741423/btad219.pdf, URL: https://doi.org/10.1093/bioinformatics/btad219.

[82] Giulio Ermanno Pibiri and Roberto Trani, « PTHash: Revisiting FCH Minimal Perfect Hashing », *in*: (July 2021), pp. 1339–1348, DOI: 10.1145/3404835.3462849.

[83] Felix Putze, Peter Sanders, and Johannes Singler, « Cache-, hash-, and space-efficient bloom filters », *in*: *ACM J. Exp. Algorithmics* 14 (Jan. 2010), ISSN: 1084-6654, DOI: 10.1145/1498698.1594230.

[84] Mikko Rautiainen and Tobias Marschall, « GraphAligner: rapid and versatile sequence-to-graph alignment », *in*: *Genome Biology* 21.*6* (2020), pp. 2157–2166, DOI: 10.1186/s13059-020-02157-2, URL: https://doi.org/10.1186/s13059-020-02157-2.

[85] Guillaume Rizk, Dominique Lavenier, and Rayan Chikhi, « DSK: k-mer counting with very low memory usage », *in*: *Bioinformatics* 29.*5* (Jan. 2013), pp. 652–653, ISSN: 1367-4803, DOI: 10.1093/bioinformatics/btt020.

[86] Michael Roberts et al., « Reducing storage requirements for biological sequence comparison », *in*: *Bioinformatics* 20.*18* (July 2004), pp. 3363–3369, ISSN: 1367-4803, DOI: 10.1093/bioinformatics/bth408, eprint: https://academic.oup.com/bioinformatics/article-pdf/20/18/3363/48906547/bioinformatics_20_18_3363.pdf, URL: https://doi.org/10.1093/bioinformatics/bth408.

[87] Lucas Robidou and Pierre Peterlongo, « findere: Fast and Precise Approximate Membership Query », *in*: *String Processing and Information Retrieval*, ed. by Thierry Lecroq and Hélène Touzet, Cham: Springer International Publishing, 2021, pp. 151–163.

[88] Leena Salmela and Eric Rivals, « LoRDEC: accurate and efficient long read error correction », *in*: *Bioinformatics* 30.*24* (Aug. 2014), pp. 3506–3514, ISSN: 1367-4803, DOI: 10.1093/bioinformatics/btu538, eprint: https://academic.oup.com/bioinformatics/article-pdf/30/24/3506/48931364/bioinformatics_30_24_3506.pdf, URL: https://doi.org/10.1093/bioinformatics/btu538.

[89] F. Sanger et al., « Nucleotide sequence of bacteriophage ϕX174 DNA », *in*: *Nature* 265.*5596* (Feb. 1977), pp. 687–695, ISSN: 1476-4687, DOI: 10.1038/265687a0.

[90] Eric E. Schadt, Steve Turner, and Andrew Kasarskis, « A window into third-generation sequencing », *in*: *Human Molecular Genetics* 19.*R2* (Sept. 2010), R227–R240, ISSN: 0964-6906, DOI: 10.1093/hmg/ddq416, eprint: https://academic.oup.com/hmg/article-pdf/19/R2/R227/1798881/ddq416.pdf, URL: https://doi.org/10.1093/hmg/ddq416.

[91] Michael Schatz, Arthur Delcher, and Steven Salzberg, « Assembly of large genomes using second-generation sequencing », *in*: *Genome research* 20 (Sept. 2010), pp. 1165–73, DOI: 10.1101/gr.101360.109.

[92] Stephan C Schuster, « Next-generation sequencing transforms today's biology », *in*: *Nature Methods* 5.*1* (Jan. 2008), pp. 16–18, ISSN: 1548-7105, DOI: 10.1038/nmeth1156.

[93] Siavash Sheikhizadeh et al., « PanTools: representation, storage and exploration of pan-genomic data », *in*: *Bioinformatics* 32.*17* (Aug. 2016), pp. i487–i493, ISSN: 1367-4803, DOI: 10.1093/bioinformatics/btw455, eprint: https://academic.oup.com/bioinformatics/article-pdf/32/17/i487/49023566/bioinformatics_32_17_i487.pdf, URL: https://doi.org/10.1093/bioinformatics/btw455.

[94] Cecilie Bækkedal Sonnenberg, Tim Kahlke, and Peik Haugen, « Vibrionaceae core, shell and cloud genes are non-randomly distributed on Chr 1: An hypothesis that links the genomic location of genes with their intracellular placement », *in*: *BMC Genomics* 21.*1* (Oct. 2020), p. 695, ISSN: 1471-2164, DOI: 10.1186/s12864-020-07117-5.

[95] Xin Victoria Wang et al., « Estimation of sequencing error rates in short reads », *in*: *BMC Bioinformatics* 13.*1* (July 2012), p. 185, DOI: 10.1186/1471-2105-13-185.

[96] J. D. WATSON and F. H. C. CRICK, « Molecular Structure of Nucleic Acids: A Structure for Deoxyribose Nucleic Acid », *in*: *Nature* 171.*4356* (Apr. 1953), pp. 737–738, URL: https://doi.org/10.1038/171737a0.

[97] Andrzej Zielezinski et al., « Alignment-free sequence comparison: benefits, applications, and tools », *in*: *Genome Biology* 18.*1* (Oct. 2017), p. 186, ISSN: 1474-760X, DOI: 10.1186/s13059-017-1319-7, URL: https://doi.org/10.1186/s13059-017-1319-7.

COLLEGE DOCTORAL BRETAGNE
MATHS, TELECOMS
INFORMATIQUE, SIGNAL
SYSTEMES, ELECTRONIQUE

Université de Rennes

Titre : Graphes dynamiques de pangénome

Mot clés : Structures de données, graphe de de Bruijn, indexation, k-mères, génomes

Résumé : Les progrès rapides des technologies de séquençage ont révolutionné la génomique, conduisant à des bases de données génomiques massives et à des milliers de génomes assemblés. Cette croissance exponentielle des données a mis en évidence les limites des modèles traditionnels basés sur des références et a motivé le développement de représentations pan-génomiques qui reflètent la diversité des espèces. Parmi ces représentations, les graphes de de Bruijn compactés (cDBG) constituent une approche de pointe pour le stockage et les requêtes sur les grands ensembles de données génomiques. En regroupant les séquences redondantes et en représentant efficacement les chevauchements des k-mères, les cDBG minimisent la mémoire et le coût de calcul. Cependant, l'ajout de nouveaux génomes sur le cDBG pose des problèmes en raison de la nature statique de la plupart structures de données basées sur des cDBG, qui nécessitent souvent une reconstruction complète, ce qui les rend coûteux et inefficaces.

Pour relever le défi de l'ajout de séquences, des méthodes permettant des mises à jour dynamiques des cDBG sans reconstruction complète sont nécessaires. Cette thèse présente, Cdbgtricks, une méthode de mise à jour d'un cDBG et de son index en ciblant les régions du graphe qui doivent être modifiées. En utilisant l'index mis à jour, Cdbgtricks permet de requêter une séquence et de rapporter les positions de ses k-mères dans le graphe, avec la possibilité de requêter des millions de séquences.

Title: Dynamic Pangenome Graphs

Keywords: Data structures, de Bruijn graph, indexing, k-mers, genomes

Abstract: The rapid advancements in sequencing technologies have revolutionized genomics, leading to massive genomic databases and thousands of assembled genomes. This exponential growth of data exposed the limitations of traditional reference-based models and motivated the development of pan-genomic representations that reflect species diversity. Among these, compacted de Bruijn graphs (cDBGs) are a cutting-edge approach for storing and querying large genomic datasets. By collapsing redundant sequences and efficiently representing k-mer overlaps, cDBGs minimize memory and computational overhead. However, adding new genomes to a cDBG creates challenges due to the static nature of most cDBG data structures, which often require complete reconstruction, making them costly and inefficient. To address the challenge of adding sequences, methods that allow dynamic updates of cDBGs without full reconstruction are needed. This thesis presents, Cdbgtricks, a method for updating a cDBG and its index by targeting the regions in the graph that needs to be updated. Using the updated index, Cdbgtricks enables querying a sequence and reporting the positions of its k-mers in the graph, with the ability to query millions of sequences.