Sorbonne Université

2021

Habilitation à diriger les Recherches
en
Informatique

présentée et soutenue publiquement le 17 septembre 2021 par

Rayan CHIKHI

K-mer Data Structures in Sequence Bioinformatics

Institut Pasteur
Ecole Doctorale "EDITE"

COMPOSITION DU JURY

M.	Kucherov Gregory	Rapporteur	DR,
Mme	Schbath Sophie	Rapportrice	DR,
M.	Stoye Jens	Rapporteur	Professeur,
Mme	Carbone Alessandra	Examinatrice	Professeur,
M.	Gascuel Olivier	Examinateur	DR,
M.	Richard Hugues	Examinateur	MCF,
M.	Vallenet David	Examinateur	DR,
M.	Vinar Tomas	Examinateur	Professeur

Acknowledgements

Thanks to Paul Medvedev and Solon Pissis for comments on Chapter 1, to Gregory Kucherov, Sophie Schbath and Jens Stoye for their reports on this manuscript, to my colleagues and co-authors for making Academia such a good place to work, to Olivier Gascuel for inciting me to write this HdR early enough, and to Marie Lefevre for helpful graphical design tips.

Contents

Forewords

This document is a "Habilitation à diriger les Recherches" (HdR), which can be described as a thesis that is written 6-10 years after one's PhD thesis. It is mandatory in the French system to write and defend such a thesis in order to be allowed to supervise PhD students without (the sometimes virtual implication of) a more senior colleague, and to apply to certain types of jobs such as University Professor or Research Director. Whether the writing of this traditional rite of passage is a productive allocation of research time is up for debate.

Unlike a regular thesis, an HdR does not contain new unpublished research. The rules of writing it vary among universities, but it is generally supposed to be the retelling of previous works. For Sorbonne Université, it is: *"la synthèse et les perspectives des travaux du candidat d'environ 50 pages (hors les articles joints)."* As all my articles are available in open-access either on arXiv, bioRxiv, and/or as a PDF on my personal web page (http://rayan.chikhi.name), I will not be joining any article along with this document.

This document will retrace some of the research endeavors I did since my PhD thesis in 2012. It will have an apparence of a survey yet it will be biased towards
• my contributions (marked as dots symbols in the left margin) and their immediate scientific neighborhood. Thus, it should not be treated as an unbiased or exhaustive survey of the state of the art.

Summary (English)

Bioinformatics is a relatively young research field concerned with applying computational techniques to biology. In this context, I will focus more specifically on algorithms that take as input DNA sequencing data, i.e. short fragments of DNA.

In the first Chapter of this document I relate the tale of how the community and myself have investigated efficient data structures for storing and representing DNA sequencing data, which has had widespread applications throughout sequence bioinformatics. Specifically, I will focus on the representations of k-mer data structures and de Bruijn graphs. They are closely tied to the problem of genome assembly, i.e. the reconstruction of an organism's chromosomes using a large collection of overlapping short fragments. I start by highlighting this connection, noting that assembling genomes is a computationnally intensive task, and then focus our attention on the reduction of the space taken by de Bruijn graph data structures. This Chapter is a retrospective centered around my own previous work in this area. It complements a recent review [CHM19] by providing a less technical and more introductory exposition of a selection of concepts.

In the second part, I present a selection my other contributions to bioinformatics, and attempt to classify them into two types of 'meta' research, based on their relation to the state of the art. I will present in an intuitive fashion the following works: REINDEER for indexing large amounts of samples, BBHash for constructing minimal perfect hash functions, BCALM2 for constructing compacted de Bruijn graphs; pugz for decompressing gzip files in parallel; Minia for representing de Bruijn graphs efficiently.

Finally, I will conclude with some perspective on my future research directions.

Résumé (French)

La bioinformatique est un domaine de recherche relativement jeune qui s'intéresse à l'application de techniques informatiques à la biologie. Dans ce contexte, je m'intéresserai plus particulièrement aux algorithmes qui prennent en entrée des données de séquençage d'ADN, c'est-à-dire des courts fragments d'ADN.

Dans le premier chapitre de ce document, je raconte comment la communauté et moi-même avons investigué des structures de données efficaces pour stocker et représenter les données de séquençage de l'ADN, qui ont eu de nombreuses applications en bioinformatique des séquences. Plus précisément, nous nous concentrons sur les représentations des structures de données k-mer et les graphes de de Bruijn. Ils sont étroitement liés au problème de l'assemblage du génome, c'est-à-dire la reconstruction des chromosomes d'un organisme à l'aide de une grande collection de fragments courts qui se chevauchent. Nous commençons par souligner cette connexion, notant que l'assemblage des génomes est une tâche de calcul intensif, puis concentrer notre attention sur la réduction de l'espace pris par le graphe de de Bruijn. Ce chapitre est une rétrospective centrée sur mes propres travaux antérieurs dans ce domaine. Il complète une revue récente [CHM19] en fournissant une exposition moins technique et plus introductive d'une sélection de concepts.

Dans la deuxième partie, je présente une sélection de mes autres contributions à la bioinformatique, et tente de les classer en deux types différents de « méta » recherche. Je présenterai de façon intuitive les ouvrages suivants : REINDEER pour l'indexation de grandes quantités d'échantillons, BBHash pour la construction de fonctions de hachage parfaites minimales, BCALM2 pour la construction de graphes de Bruijn compactés ; pugz pour décompresser les fichiers gzip en parallèle ; Minia pour représenter efficacement les graphes de de Bruijn.

Enfin, je conclurai par quelques perspectives sur mes futures orientations de recherche.

Chapter 1

A tale of optimizing the space taken by de Bruijn graphs

1.1 Context

Back when I started a PhD in bioinformatics in 2008, my advisor Dominique Lavenier told me about a relatively new problem consisting in reconstructing genomes using DNA sequencing, termed *de novo* genome assembly. It was a somewhat "fresh" problem at the time: many genomes were already assembled, e.g. the Human Genome Project completed at the beginning of the 2000s, yet performing the assembly of any organism was just starting to be within reach for most biological labs. The vast majority of organisms did not have their genomes assembled (and as of today: they still do not). So the challenge was to create software that any individual lab could use, not just large organizations. The main type of data at the time were *short reads*, i.e. fragments of around 100 nucleotides, meaning that only a tiny fraction of a genome could be read contiguously at a time. (Genomes of viruses are in the order of thousands of nucleotides, but for most other organisms they range from millions to billions.) By repeatedly sequencing fragments from random locations, reads would significantly overlap which makes genome reconstruction possible. Short reads were produced mainly from the company Illumina, still a market leader on DNA sequencing today; some of the previous technologies (e.g. 454) were on their way out.

The EULER-SR assembler was one of the first specialized genome assembly software for short reads, and it came out in 2008. It achieved an assembly of a bacterium (*E. coli*) in 199 pieces [CP08]. This means that the genome was near-completely reconstructed, yet in a fragmented way to due ambiguities. This may seem unremarkable by current standards, as nowadays we can reconstruct nearly all bacteria in a single piece per chromosome. Yet the task was fundamentally hindered by the length of the short reads. Still, at the time it was clear that the next frontier would be to assemble larger genomes, e.g. animals or plants, even if the final assembly would still be largely fragmented.

The widely-used Velvet [ZB08], ABySS [SWJ+09] and SOAPdenovo [LZR+10] assemblers appared in the following two years. And indeed, the last two were able to assemble a human genome using a cluster or a single large-memory machine. These assemblers were all based on a certain representation of the input data, the de Bruijn

graph, that we will explain in the next section. These graphs come from mathematics and had not yet been widely used outside of some networking applications. This was before the era of more advanced assemblers (IDBA/SPAdes [PLYC10, BNA+12]); early assemblers only constructed a single graph, as opposed to iterating over multiple graphs with different parameters.

Even so, the construction of a large de Bruijn graph was the most computation-intensive step of genome assembly at the time. This should come as no surprise, as 1) this was the first period in history when one had to construct large de Bruijn graphs in any domain; there existed no previous litterature describing how to do it efficiently, and no software library. 2) It did not matter so much if construction was slow or memory-intensive, as long as some large-memory machine managed to run it. 3) The volume of input data was really large by historical standards: in the order of a hundred of gigabytes in compressed form. Yet, as genome assembly later became a routine task, along with the advent of huge instances such as metagenomics (the analysis of multiple genomes at once), the efficient construction de Bruijn graphs naturally became a critical aspect of genome assembly. It also turns out that de Bruijn graphs would be useful for other biological sequence analysis tasks, such as the sequencing of RNA [PDL+17], the compression of genomic data [HWSH17], and the detection or representation of variations across a single or multiple genomes [ENS+20].

The goal of this chapter is to retrace some of the steps that the community and I took towards achieving space-efficient representations of de Bruijn graphs, starting from the initial attempts in the first assemblers, making a detour through theoretical lower bounds, and finishing with current advances and some perspectives.

1.2 Problem formulation

Let us introduce some of the concepts. A DNA sequence is seen as a string over four possible characters (A,C,T,G). A **k-mer** is a portion of DNA sequence of length k, e.g. ACT is a 3-mer. The **de Bruijn graph** is a directed graph where nodes are k-mers, and edges are the exact suffix-prefix overlaps of length $k-1$ between two nodes; e.g. ACT→CTA or AAA→AAT, but ACT and AAA are not connected by an edge. See Figure 1.1 for another example. Note that in practice, k is typically greater than 20. A de Bruijn graph is constructed by inserting all the possible k-mers present in an input dataset. If the same k-mer is seen multiple times, all of its occurrences are associated to the same node.

The scientific question we will be interested in this Chapter can be informally stated as follows: given a set of nodes of the de Bruijn graph, stored on disk, construct an in-memory representation[1] that supports a reasonable subset of standard graph operations, e.g. determine all the neighbors of a node, determine if some putative node is present or absent, etc. The representation should take as little memory as possible, and answer queries reasonably fast, although as we will see next, the main limiting factor here is typically not the query time but the representation size.

Note that prior to circa 2012, the problem as stated above was not recognized

[1] Such a representation is also often called a **data structure**, and the abstract model that encompasses all the data structures supporting the same operations is called an *abstract data type*.

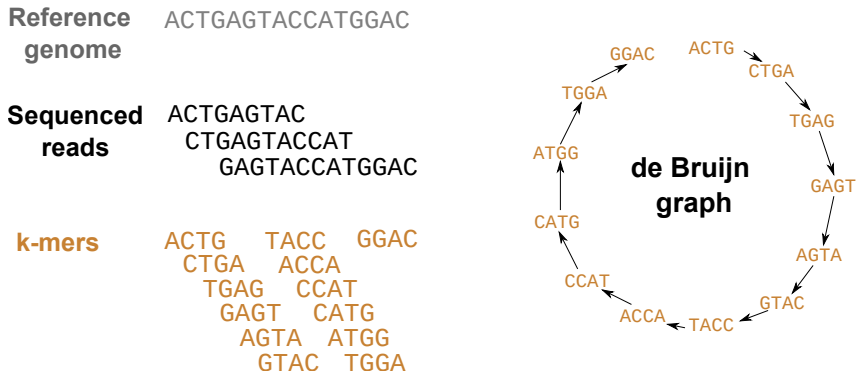

Figure 1.1: Left panel: example of a toy reference genome sequence, a set of 3 sequenced reads, and the corresponding 4-mers extracted from the reads. Right panel: the de Bruijn graph constructed these reads with $k = 4$ and drawn using a circular layout.

as its own area of investigation within bioinformatics nor computer science. Arguably it became one when several data structures were published as stand-alone articles [CB11, CR13, BOSS12].

1.3 Caveats

We will focus here on only a selection of major milestones, where space usage was reduced, ignoring other features such as query times. The presentation will also
- sacrifice some technical accuracy in favor of accessibility. For a more complete and technical exposition, please refer to this review [CHM19].

Note that genome assembly cannot be reduced to the representation of the de Bruijn graph. In fact, many older tools even used different paradigms [MKS10]. Among those which do use a de Bruijn graph, they implement many steps before (e.g. error correction) and after (e.g. graph cleaning) the construction of the graph that crucially affect results quality. However, for the sake of keeping the story coherent, we will set aside this broader environment to focus solely on the efficiency of graph representation.

1.4 The early days

The early assembly programs from the 2008–2010 era did not particularly aim to optimize the space usage of de Bruijn graphs. Therefore, their memory usage may be seen as wasteful by current standards, yet they laid the bases for future progress.

The EULER-SR assembler reported building the graph using what they describe as "an efficient hashing structure" which was then transformed into a sorted list of vertices, queried using binary search. Notably, k-mers were represented explicitly as strings.[2]

[2]The total space usage of the graph was reported to be $O(L) * (v + k)$ bytes, where L is the genome size, k is the k-mer length and v is the memory allocated per vertex, reported to be 40

Similarly the Velvet assembler, published the same year, used a hash table to record for each k-mer "the ID of the first read encountered containing that k-mer and the position of its occurrence within that read". It is natural to want to keep track of where each k-mer is coming from, however as we will see next, storing this information in the graph is prohibitively expensive. The authors note: "The main bottleneck, in terms of time and memory, is the graph construction. The initial graph of the *Streptococcus* reads needs 2.0 [gigabytes] of RAM." Given that the *Streptococcus* genome is 2 million nucleotides in length, and under the assumption that there were roughly 10x more erroneous k-mers than correct ones, we infer that the de Bruijn graph representation of Velvet required in the order of 100 bytes per k-mer.

The SOAPdenovo assembler followed the Velvet assembler strategy, except that its authors realized that one could achieve nearly identical (or even better) results despite discarding a lot of space-intensive information in the hash table (i.e. read locations and paired-end information). Its graph representation required 120 GB of memory for storing 5 billion nodes of a human genome [MKS10], i.e. around 24 bytes per k-mer. This prowess demonstrated that the quality of genome assembly was not sacrified when trimming down the de Bruijn graph data structure. There existed some minimal set of supported operations that would make a de Bruijn graph fit for purpose, although this set was not described at the time. As long as a data structure would support all these features, then computer scientists would be free to optimize it as much as they could.

The Meraculous assembler, published in 2011, took a radically different approach by storing the de Bruijn graph using collision-free hashing. Its representation only supports the lookup of the next nucleotide following a k-mer (i.e. the out-neighbor of a node), where k-mers having multiple out-neighbors were previously discarded during a pre-processing step. As in other assemblers, there are further steps taken to attempt to "fill the gaps" between the discarded k-mers and to orient the assembled fragments, yet these are outside our current scope. The Meraculous de Bruijn graph structure does not support enumeration of vertices. Despite the apparent minimality in terms of supported operations, it appeared to be sufficient for enabling genome assembly. While this technique was not further re-used by other assembly tools, we revisited it 6 years later to develop the general-purpose *minimal perfect hashing* library BBHash [LRCP17].

1.5 The birth of a line of research

The years 2011–2012 saw a remarkable amount of independent contributions proposing new ways to represent the de Bruijn graph in a space-efficient manner. In retrospect, the field was ripe for such contributions as there was an important problem to be solved (genome assembly of human genomes was taking a prohibitive amount of memory), which was well-defined computationally[3], and there were no previous "clever" solutions apart from using off-the-shelf data structures.

bytes.

[3] At least implicitly, as to my knowledge, it has not been explicitly formulated as an open question in an article.

To my knowledge, the first article on this topic was published in 2011 by T. Conway and A. Bromage [CB11]. They describe an encoding of the de Bruijn graph using an existing state-of-the-art efficient encoding of bit arrays. Furthermore, they also show that their representation is 'optimal', in the following sense: information theory dictates that any other exact de Bruijn graph representation will have to use as many bits per k-mer in the worst case. The key words of the previouse sentence are "exact" and "worst case", and we will revisit this statement later in this document. But for now, it is sufficient to note that to this date, the Conway-Bromage data structure is provably optimal. Then, does this mark the end of the line of research on the representation of de Bruijn graphs?

Not quite, despite the lower bound argument being convincing. We will briefly expose it here. Observe that to represent a de Bruijn graph, one only needs to represent its vertices[4]. The edges are indeed implicit in the representation, as one could determine all the neighbors of a certain k-mer by querying for the presence of all the potential k-mers shifted to the left or to the right.

Then, a bijection is established between the set of all possible sets of n vertices, and the set of all possible binary vectors having n ones and $4^k - n$ zeros. The bijection is actually rather straightforward: each k-mer is directly encoded as an integer in base 4 (see Figure 1.2 middle panel), and a bit vector has a **1** at position i if and only if i is the encoding of a k-mer that belongs to the set of graph vertices. Since the number of possible bit vectors is classically known, one deduces that to represent a de Bruijn graph for a certain parameter k having n vertices, one must uses in the worst case as many bits as the logarithm of the number of possible bit vectors of size 4^k that have n ones.

1.6 Beating the lower bound (by inexactness)

As it turns out, this lower bound did not discourage researchers from proposing data structures with even lower space usage than dictated by the bound. Although this seems impossible, we are about to see why it is not. One such data structure is the encoding of a de Bruijn graph using a Bloom filter by Pell *et al* [PHCK+12] (Fig 1.2). By inserting all the vertices of the graph inside a probabilitic membership data structure (here, the Bloom filter), it is possible to represent a set of k-mers approximatively. With a trade-off: the graph is not exactly represented, yet the space usage is an order of magnitude lower than the one dictated by the Conway-Bromage lower bound: around 4 bits per k-mer. Pell *et al* showed that despite having many false positive nodes resulting from the approximate representation, it was still possible to perform useful analysis on the graph – not quite genome assembly, but another related task (read partitioning).

- Following this, G. Rizk and I proposed to extend this representation to make it exact within a certain setting, and perform genome assembly [CR13]. Due to the lower bound, any attempt at removing all the false positives of the Bloom filter would result in a data structure that would necessarily be at least as large as the

[4] We omit a technicality here that will only be of interest to specialists. We only consider node-centric de Bruijn graphs. For edge-centric de Bruijn graphs, the argument stated in this paragraph does not apply. Yet, edge-centric graphs are tightly related to node-centric ones and in practice, using one definition or the other does not matter.

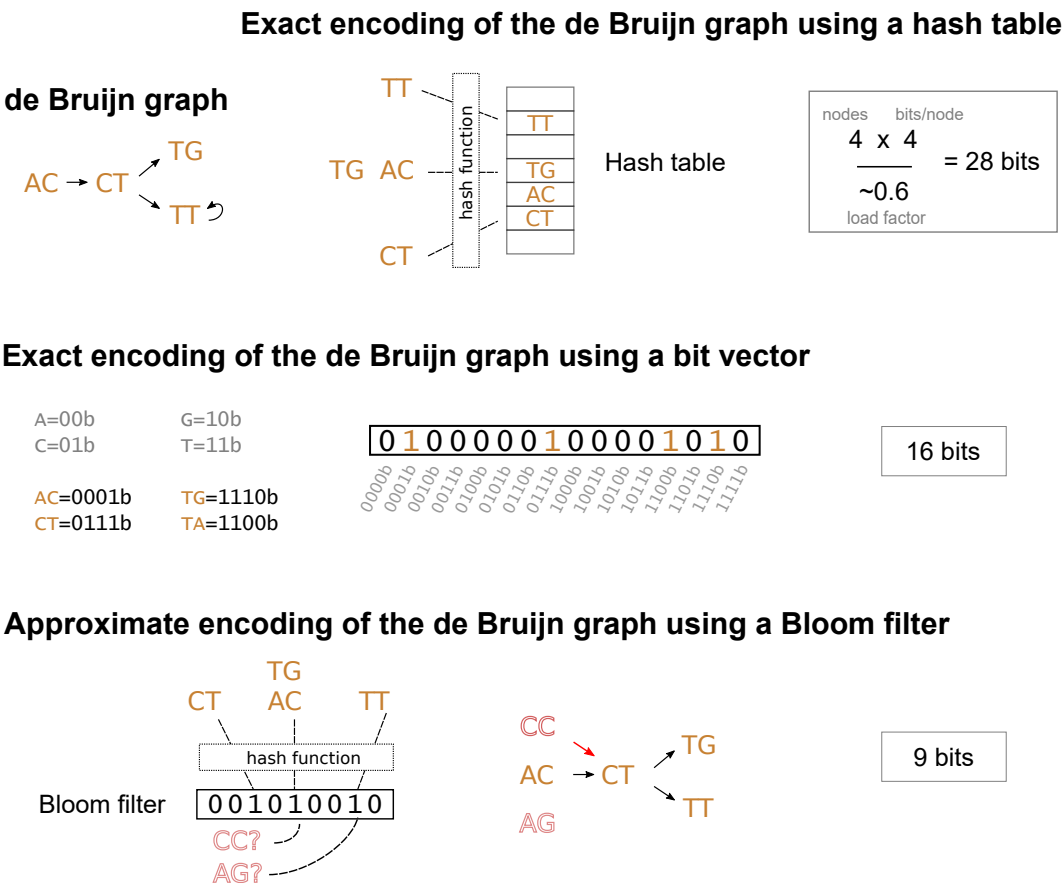

Figure 1.2: Example of a de Bruijn graph (top left panel) and three possible encodings. Top right panel: hash table, each node is inserted at a position given by a random hash function. The collision between TG and AC is resolved using linear probing, i.e. by inserting AC at the next free slot in the table. The load factor is number of occupied cells over total cells. Middle panel: bit vector, storing each node converted into an integer using the classical binary encoding of characters `A,C,G,T=00b,01b,10b,11b`, where `b` indicates that the number is written in binary. Bottom panel: Bloom filter, where each node is inserted at a position given by a random hash function. Two false positives nodes (CC, AG) are shown in red. They arise because the hash function causes collisions between any possibly existing node and true nodes.

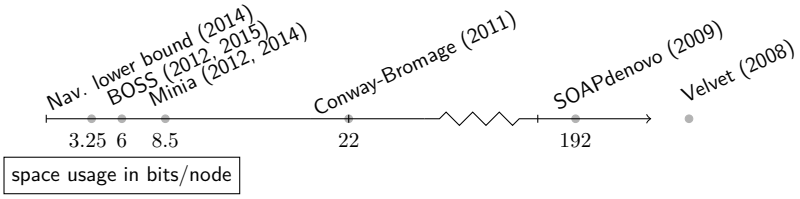

Figure 1.3: Space taken by various representations of de Bruijn graphs, in bits per node. "Conway-Bromage" is both the Conway-Bromage exact lower bound and its matching upper bound. "Nav. lower bound" is the navigational lower bound for general de Bruijn graphs from [CLJ+14]. BOSS is the flavor of [LLL+15].

one from Conway-Bromage. The key insight was to realize that only a fraction of the false positives of the Bloom filter mattered: those which were neighbors of a true positive vertex. By explicitly storing them in a "blacklist" hash table, our data structure managed to represent a graph in typically ≈ 13 bits per k-mer[5], and the neighbor query operation would be exact. This "beats" the Conway-Bromage bound by a factor of roughly 2x. The caveat is that the representation is not exact everywhere, but as long as a user traverses the graph from a true positive vertex, then the representation would act identically to the exact one. We implemented this data structure inside a genome assembler, Minia [CR13].

Another instance of an inexact de Bruijn graph representation is the *sparse de Bruijn graph* [YMC+12], which is a de Bruijn graph that skips g intermediate k-mers, providing roughly a $1/g$ space saving, where g was set to 16. Finally, along the same line of thought the A-Bruijn graph formalism [LYK+16] selects an arbitrary set of strings, and creates an edge when two strings appear consecutively in at least one read. This concepts generalizes de Bruijn graphs; yet A-Bruijn graphs may contain one or several orders of magnitude less nodes than de Bruijn graphs. There are some potentially interesting parallels between A-Bruijn graphs and sparse de Bruijn graphs, yet to the best of my knowledge they have not been explored.

1.7 Beating the lower bound (by instance specificity)

Independently of Minia, and presented at the same session of the WABI conference in 2012, the BOSS data structure proposes a completely different yet exact de Bruijn graph representation [BOSS12]. It uses a variant of the Burrows-Wheeler transform specifically tailored for k-mers. A complete description of the BOSS structure, or even the Burrows-Wheeler transform, would be beyond the technical level of this document, and can be found in [AKP+20]. Intuitively, the Burrows-Wheeler transform [BW94] is a permutation of the characters of a string that facilitates substring search and compression. BOSS extends this concept by storing a permutation of the last characters of each k-mer[6] together with a bit array. The result is a data

[5] Which would later be improved by Sahlikov & Kucherov to ≈ 8.5 bits per k-mer, using cascading Bloom filters [SSK14].

[6] along with some additional artificial k-mers to "pad" those which do not have a large enough neighborhood.

structure that supports efficient membership queries and neighborhood traversal of the graph, all in around 6 bits per k-mer in practice. While in the first few years the construction of this structure was relatively impractical, recent improvements lifted those limitations, allowing to process even terabases of input data [KMD+20].

Taking a step back, BOSS is an exact representation that appears to somehow beat the Conway-Bromage lower bound. How is this even possible? While this aspect was not discussed in nearly all of the publications related to BOSS, it turns out that BOSS has been mainly applied to k-mer sets that have a so-called *spectrum-like property* [CHM19], i.e. where all the k-mers originate from some underlying long strings. Should BOSS be applied to an arbitrary set of k-mers, its space usage would mechanically be raised to match or exceed the Conway-Bromage lower bound; yet, this fact has to my knowledge never been properly tested in practice.

Regardless, the spectrum-like property and the effectiveness of BOSS are important insights: a data structure may do better than the worst-case lower bound while still remaining exact, when it is restricted to a certain class of inputs that matter in practice. Then, a natural next question arises: what would be a more realistic lower bound for representing 'practical' de Bruijn graphs, i.e. those having spectrum-like property?

- Several collaborators and I addressed this question in [CLJ+14], where we formulated several concepts. First, we defined a *navigational* data structure as one that enables navigation in the graph but does not necessarily support membership queries. We showed that navigational data structures for general de Bruijn graphs require at least 3.24 bits per k-mer in the worst case. When restricted to the family of linear de Bruijn graphs (i.e. graphs where all nodes have a single in-neighbor and/or single out-neighbor), then a lower bound for navigational data structures is 2 bits per k-mer. This last lower bound is tight, as representing the linear de Bruijn graph using the Burrows-Wheeler transform (or its optimized flavor, FM-index [FM00]) yields also a data structure that is asymptotically close to 2 bits per k-mer. In [CLJ+14], we also proposed a new data structure for de Bruijn graphs having the spectrum-like property, using the Burrows-Wheeler transform, and showed that it takes $2+(k+2)c/n$ bits per k-mer, where c is essentially the maximal number of k-mer-disjoint strings the k-mers could have been generated from[7].

- Lastly, one may also wonder how a de Bruijn graph could be further compressed, e.g. to be stored on disk. Supposedly such a compressed representation would be even smaller than the previously mentioned data structures. The trade-off is the inability to perform fast queries. Two independent works [BBK20, RCM20], one on *simplitigs* and the other on *spectrum-preserving string sets* which I was associated with, proposed to store non-overlapping paths of the compacted de Bruijn graph (defined later) as sequences, and store them in compressed form on disk. Despite the representation apparently storing an incorrect representation of the graph, due to paths being constructed by choosing edges arbitrarily, one may observe that the original graph can be reconstructed losslessly from its path representation. Such a disk representation achieved a space very close to the 2 bits per k-mer lower bound: 4.1 bits per k-mer for a whole human genome read dataset, and 2.7 bits per k-mer for a human metagenome.

[7] For specialists, c is the number of unitigs.

1.8 Construction algorithms

An apparté will be made in this section, where we will briefly mention the data structure construction algorithms. One typical pre-processing step commonly done prior to creating a de Bruijn graph data structure is k-mer counting. This step takes the input sequencing data and yields the set of distinct k-mers present in the input along with their abundances. It essentially constructs the nodes of the de Bruijn graph.

During the development of Minia, we had ran into an issue. The graph representation was so succinct that other steps of the genome assembly pipeline acted as bottlenecks, including k-mer counting. At the time, the most efficient k-mer counter was Jellyfish [MK11], which used a custom thread-safe hash table optimized specifically to store k-mers. Yet, Jellyfish would have used much more memory than Minia. We therefore set out to design a low-memory k-mer counting tool that would use the disk to alleviate memory usage (DSK [RLC13]). This strategy was also used by other popular k-mer counting tools, e.g. KMC [DDGG13].

The problem of k-mer counting is fascinating in its simplicity but also difficult to engineer correctly, given that large volumes of input sequences need to be processed with high CPU utilization, low memory usage, and bandwidth-limited disk accesses. A relatively current review is [MS18]. After k-mer counting, nearly all of the data structures presented above have their own, customized construction algorithms. As such, there does not exist an 'universal' construction algorithm for de Bruijn graph that would then be slightly adapted to derive a particular data structure.

However, several recent data structures (the one presented in [CLJ+14], Pufferfish [ASSP18], BLight [MKL21]) require as input a common object: the **compacted de Bruijn graph**. It is obtained from a classical de Bruijn graph by transforming each non-branching path into a single node, similarly to suffix tries are transformed into suffix trees by collapsing paths of vertices having one child. However, this is a circular situation: in order to construct an efficient representation of the classical de Bruijn graph, one must have already constructed a compacted de Bruijn graph, which itself is obtained from the classical de Bruijn graph. In order to break this

- circularity, My colleagues and I proposed an efficient construction algorithm for the compacted de Bruijn graph [CLJ+14], which uses a constant amount of memory. It was further extended to make use of multiple threads efficiently [CLM16].

1.9 Current state of the art

Since the influx of de Bruijn graph data structures in 2012, several more have been published in the recent years. As it turns out, many of them are based on minimal perfect hashing. It is a variation of a hash table which does not store its keys, yet still manages to resolves collisions. Minimal perfect hashing is unable to confirm if an arbitrary key is present or absent in the structure, however for any key that was inserted during its construction, it returns an exact answer. This makes the

- structure highly space-efficient. Along with colleagues, we proposed a fast parallel C++ library for constructing minimal perfect hashes (BBHash [LRCP17]) which has been engineered to scale significantly better than other existing implementations at

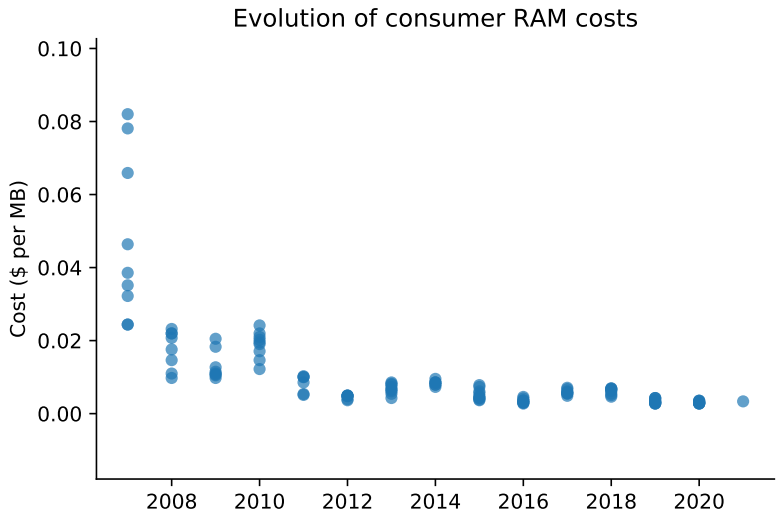

Figure 1.4: Consumer RAM costs from the 2007–2021 period. Each dot is a retail DIMM product sold on the current year. Source: https://jcmit.net/memoryprice.htm

the time.

Among current de Bruijn graph data structures, I will briefly highlight Pufferfish [ASSP18] and Blight [MKL21] which are both based on the compacted de Bruijn graph, and queries are supported by an additional minimal perfect hashing structure that quickly locates positions within nodes of the compacted graph. The Bloom Filter trie [HWS16] and Bifrost [HM20] structures both use Bloom filters in addition with other auxiliary data structures to keep the representation exact, yet even brushing their algorithmic details would be too technical for this survey. Counting quotient filters [PBJP17] improve upon Bloom filters by also storing the number of occurrences of each node in the input data. Belazzougui *et al.* proposed a navigational data structure based on minimal perfect hashing, with a clever addition of a tree data structure to restore membership queries. For more details on all of these data structures, see [CHM19].

In a way, one might acknowledge that "the dust has settled" in the landscape of de Bruijn graph data structures. The bioinformatics algorithms community has attempted for several years to come up with solutions that combine low space usage, fast query speed and a reasonable set of features. The outcome is a set of current data structures that achieve reasonable trade-offs, with space close to the known lower bounds. As a result, de Bruijn graphs are no longer a bottleneck in genome assembly, partly also due to decreasing RAM costs (Fig 1.4).

Then, is this the end of this line of research? Not quite, as the natural next frontier is the representation of multiple genomes within a generalization of the de Bruijn graph.

1.10 Colored de Bruijn graphs

As a coincidence of dates (or perhaps not), 2012 was not only the year where many seminal data structures for de Bruijn graphs were proposed, but also the year when the term *colored de Bruijn graph* was coined (in [ICT+12]), which will pave the way to the next type of contributions that we will mention here. Colored de Bruijn graphs generalize de Bruijn graphs to multiple samples. When faced with multiple samples, a classical de Bruijn graph would bundle them together and consider the union of all samples as a single "mega-sample". Colored de Bruijn graph also do that, but they add additional information associated to the nodes so that one can tell the origin of each node across samples. Naturally, speaking in terms of lower bounds, storing such a graph for multiple samples should require strictly more space than storing the graph of any subset of samples.

Several data structures have been proposed to store colored de Bruijn graph, the first of which was based on an efficient hash table [ICT+12], then later using the
- Burrows-Wheeler transform [MBN+17]. More recently my colleagues and I proposed the REINDEER structure, based on compacted de Bruijn graphs and minimal perfect hashing [MIG+20], with the distinctive feature of not only storing the presence/absence of nodes, but also the approximate frequency of each node within each sample.

To the best of my knowledge, there has been no attempt made at formulating space lower bounds for colored de Bruijn graphs. One may obtain one through an immediate application of existing lower bounds to the union graph disregarding color information. However, this would be a loose bound as much of the difficulty of storing colored graphs lies in the color information.

1.11 Wrap-up and open questions

As we reviewed above, many data structures have been proposed to store de Bruijn graphs, achieving several order of magnitudes improvement in space usage compared to using off-the-shelf data structures for graph storage. From this perspective, the theoretical study of data structures along with their practical implementations has been successful at providing performance gains for widely-used software tools (e.g. [BNA+12, LLL+15]). Looking back, the improvements have mainly be due to two realizations. 1) Data structure exactness can be sacrified yet still provide exact results in a certain frame of operations. 2) The theoretical worst-case analysis of data structures inadequately applies to practical instances. The latter realization is the topic of an upcoming article from Medvedev [Med], critically reflecting on the analysis of bioinformatics algorithms more broadly.

Several topics were not covered in this document, to keep it simple. One is double-strandedness, which forces all the data structures mentioned above to consider that a k-mer and its reverse-complement should be the same object; this adds theoretical and especially practical complications, yet does not fundamentally change the exposition of the data structures. An additional one is the use of multiple k values. Nowadays genome assembly tools on short reads typically construct multiple de Bruijn graphs iteratively. This is a somewhat orthogonal matter as presented

here, given that each individual graph is represented using one of the techniques above. We note however that some works have attempted to unify multiple graphs into one [BBG+15, CR20]. Another consideration is how to store the number of times each k-mer is seen in the input. All these considerations are discussed in more details in [CHM19].

We summarize here a few open questions:

1. Can compressed representations e.g. spectrum-preserving string sets be made efficiently queryable? This would lead to even more compressed de Bruijn graphs.
2. What would be a space lower bound for exactly representing a colored de Bruijn graph of n samples, each sample i having D_i distinct k-mers?
3. A matching upper bound of the above.
4. How to efficiently represent not only the presence/absence of a node but also its abundance in colored de Bruijn graphs (improving upon REINDEER [MIG+20]).

Chapter 2

An attempt at taxonomizing research problems, with a bioinformatics case study

2.1 Introduction

In this Chapter I will present a more varied selection of other contributions, in the spirit of illustrating a sufficient body of work for defending an HdR. In doing so, I will also categorize these contributions at a more abstract level, i.e. in terms of what sort of research questions they are, independently of their application domain. The goal of this categorization is partly self-reflective, i.e. to help me understand what I am keen on tackling and what sort of project ends up being fruitful. (The intersection between the two categories is unfortunately not the union.) It is also partly illustrative, to provide the reader with some considerations on my research philosophy, in the hope that this sort of content akin to a "retrospective research statement" fits the scope of an HdR document. I will also highlight the collaborative nature of all these works, some of which were performed by students under my supervision, and will describe the source of the research funding in each case.

2.2 K and U problems

My personal definition of performing applied research is to achieve a non-trivial goal X, where X was never done previously. I consider two distinct types of problems:

- **Type "K"** (for **K**nown) problems: we have some idea of where to start on the path towards achieving X, and possibly could outline the whole research plan.
- **Type "U"** (for **U**nknown) problems: we have no idea of where to start, or we hold the belief that X is impossible. (Yet, X will turn out to be possible.)

The term "we" in those definitions refers to any subset of researchers. To simplify, I will consider that "we" is the entirety of a particular research community, e.g. the set of people working on algorithmic bioinformatics.

K problems may be ideal for applying for certain sources of funding requiring that a complete methodology is laid out, risks are assessed, and evaluators will be

Tool name	Year	URL	Type	Citations (2021)
REINDEER	2020	github.com/kamimrcht/REINDEER	K	9
BBHash	2017	github.com/rizkg/BBHash	K	50
BCALM2	2016	github.com/GATB/bcalm	K	113
pugz	2019	github.com/Piezoid/pugz	U	9
Minia	2012	github.com/GATB/minia	U	376

Table 2.1: Overview of my presented research works that will serve as illustrations to type K and U problems.

confident that the research program has a chance to be successful. K problems are more straightforward to outline than U problems, as sketching the research plan does not require to have started carrying the research. Conversely, U problems might require large amounts of preliminary work to even figure out the research program, which could very well end up being the hardest part of the research. I suspect that it is easier for K problems to be qualified as "incremental work" or "engineering" than U ones.

One can be tempted to introduce another type, the Type "I": where X is truly impossible to achieve. I will argue here that such a type is unnecessary as one could formulate a dual problem: "prove that X is truly impossible" which would end up in either category K or U. The original type "I" problem is then subsumed and brought into one of our two defined classes.

Determining if an open problem is K or U is unfortunately an endeavour of uncertain outcome. If a problem is truly U but an individual thinks that it is K, then it is likely that the envisioned research plan would end up to be unfruitful. One may argue that all problems are truly K yet some are put into the U category until someone figures out the right steps. Indeed, if there existed an Oracle capable of guessing any research plan perfectly, this would be the case. I will argue that such an Oracle does not exist, and while some individuals may approximate such an Oracle with reasonable accuracy, there will always be a space for U problems.

The K/U duality bears some similarities with the NP complexity class. Indeed, a NP problem has a polynomial-sized solution that may have taken an exponential time to find. Similarly, one can think of a valid research plan of an U problem as a polynomial-sized string. In the case of K problems, such a string is often readily guessed. For U problems, finding such a string may take up to exponential time.

2.3 Overview of the presented works

Table 2.1 summarizes the research works that will be presented in the rest of this section. They are all either my direct contributions or works I was closely involved with.

2.4 Some K problems

2.4.1 REINDEER

- As a transition from the previous Chapter, I will describe in more details the REINDEER algorithm [MIG+20]. This will illustrate a K problem starting from a familiar context.

All the algorithmic ideas were proposed by Camille Marchet, who was my postdoc at University of Lille at the time. Along with my co-advisor Mikaël Salson, we set Camille on the course of this project and provided feedback along the way. This work was funded by a national research grant (ANR Transipedia, headed by Daniel Gautheret) aiming to perform large-scale transcriptomes analysis.

Given a collection of samples (in our case, short-read RNA-sequencing data), REINDEER is a method that constructs a representation of the de Bruijn graph of the union of the samples. It also records the frequency of each k-mer in each sample. It is furthermore able to support fast k-mer queries, i.e. the graph is indexed.

The main algorithmic ingredients of REINDEER are 1) minimizers, 2) count-vectors, 3) monotigs, and 4) the BLight index method:

1. Minimizers [RHH+04] are substrings of fixed length (typically 10-12), which correspond to the lexicographically smallest string within a longer string. Minimizers are a now classical notion in sequence analysis, as they help partition sequences into buckets, and facilitate alignment or assembly of similar sequences.

2. A count-vector is simply the vector of frequences of a given k-mer across all the samples. For example given three samples, if a k-mer is seen three times in sample 1 and four times in sample 2, and not in sample 3, its count vector is $\{3, 4, 0\}$.

3. Monotigs are sequences of a path in the de Bruijn graph where all k-mers have the same minimizer and the same count-vector.

4. BLight [MKL21] is an indexing data structure for k-mers akin to a hash table, optimized for space efficiency and query speed.

These ingredients are combined in the following way: monotigs are constructed then indexed by BLight, which uses minimizers behind the scenes. A count vector is associated to each monotig. To improve space efficiency, count-vectors are deduplicated and BLight only stores references to a table of distinct count-vectors. Figure 2.1 illustrates how these ingredients are combined.

The main algorithmic novelty in REINDEER is the concept of monotig, and putting all the other ingredients together forms a new method. This statement however hides several other parts of the work that are no less important, e.g. coming up with a monotig construction algorithm, making a robust software overall, performing evaluations on large datasets, etc..

In terms of results, REINDEER is the only known implementation capable of storing k-mer counts along with their presence/absence, thus we could not directly

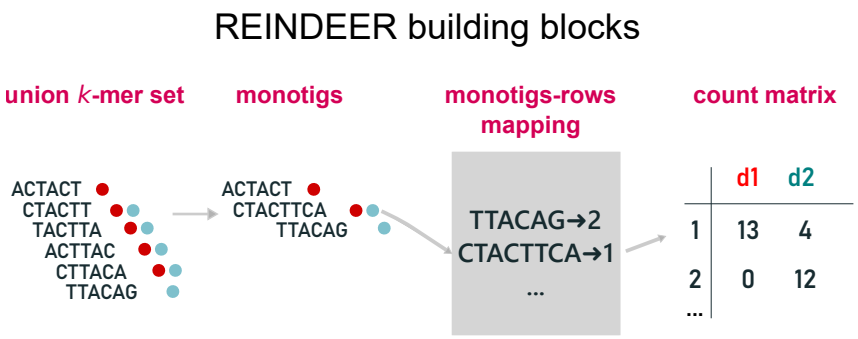

Figure 2.1: Illustration of the REINDEER method. A union k-mer set is constructed from all the samples, then converted into monotigs. A mapping between monotigs and count matrix rows is created (gray box), and in addition a mapping between k-mers and monotigs is realized (not shown in the figure) using BLight. Finally, a count matrix is created which records the average count of k-mers across monitigs, for each sample. (Figure by Camille Marchet, adapted)

compare it with any other method. That said, it achieves slightly higher but comparable index sizes than methods that only index the presence/absence of k-mers, with similar construction time and peak RAM. However, it requires 1-2 orders of magnitude more disk during construction.

Why is it a K problem There were several possible routes towards achieving the goal of REINDEER. Building upon classical k-mer counters and compacted de Bruijn graph construction, obtaining unitigs along with their mean k-mer abundance was a straightforward first step. From this base, Camille decided to go with the "monotig" route, and ended up using an efficient existing component (BLight) to index them. It might have also been feasible to e.g. directly store k-mers and their count-vectors in BLight, which would have resulted in similar functionality yet likely inferior performance. Alternatively, we could also have indexed the compacted de Bruijn graph using a FM-index and then add abundance information, which would also possibly have been less effective in terms of construction speed. Perhaps we could also have designed a different type of "monotig"[1]. Either way, we were confident that several routes, some of which consisting of piecing together existing building blocks, would lead to a successful outcome. I believe this is a hallmark of a K problem.

2.4.2 BBHash

Also mentioned in the previous Chapter, the research around BBHash is another K problem by the nature of its algorithmic ingredients. Here it is even debatable whether the contribution is of engineering nature or of research nature, however I will argue later that it is indeed in the research realm.

Again, this work was mainly performed by colleagues: Guillaume Rizk and Antoine Limasset designed the algorithm and coded the software, joined by Pierre Pe-

[1]See this excellent blog post on the existing -tig concepts: https://kamimrcht.github.io/webpage/tigs.html

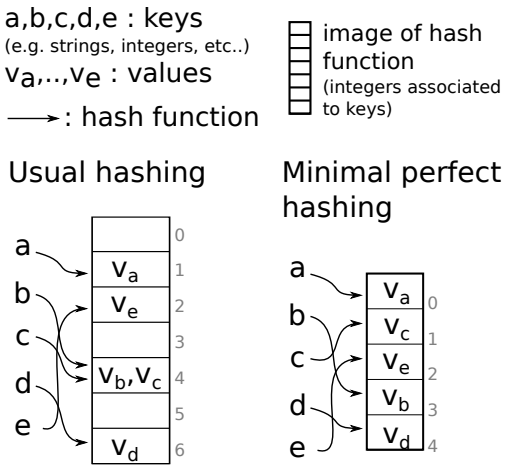

Figure 2.2: Classical hashing versus minimal perfect hashing. A set of keys $\{a, b, c, d, e\}$ is to be associated to respective values $\{v_a, \ldots, v_e\}$. The bottom left panel indicates a possible memory organization for a classical hash table, where some locations are unoccupied, and some others are occupied by one or more values associated to keys. In the case of multiple values, this is known as a collision. To achieve this organization in practice, each location can be implemented as e.g. a linked list. On the bottom right panel, a minimal perfect hash function allows to maximize utilization by having no empty location, and none of the cells have collisions, therefore each location is e.g. a memory chunk of the size of the value type.

terlongo and myself for the manuscript preparation. However I take pride in the role of highlighting that it was a problem worthy of being investigated, and pointed out the limitations of the state of the art at the time through the creation of a comprehensive benchmark implementation (`https://github.com/rchikhi/benchmphf`) that served as a touchstone for BBHash. This work was not funded by any particular grant, all authors contributed to it on their own principal research time.

- BBHash [LRCP17] is a **minimal perfect hash function**, i.e. a hash function that associates an integer range to a pre-defined set of keys. The "minimal" part refers to the fact that integer range has the same cardinality as the set of input keys. The "perfect" part indicates that there are no collisions, i.e. two keys must be associated to two different integers. See Figure 2.2 for an illustration.

  BBHash works in the following way. An set of elements are given as input. They can be strings, integers or more complex objects, as long as one can design a classical hash function to hash them. A Bloom filter is created over this input set. Then, all the elements that collide over this Bloom filter, i.e. which correspond to the same '1', are recoded in a separate set. Another Bloom filter is created for this separated set, and all the colliding elements are recorded again. These steps are repeated as lon as the set of colliding elements is larger than a certain predefined size. Finally, if there exists a remaining (small) set of colliding elements, they are stored in a classical hash table. Figure 2.3 illustrates this process.

  Through exhaustive benchmarks, we demonstrated that BBHash achieves at least two orders of magnitudes better space efficiency during construction, and at

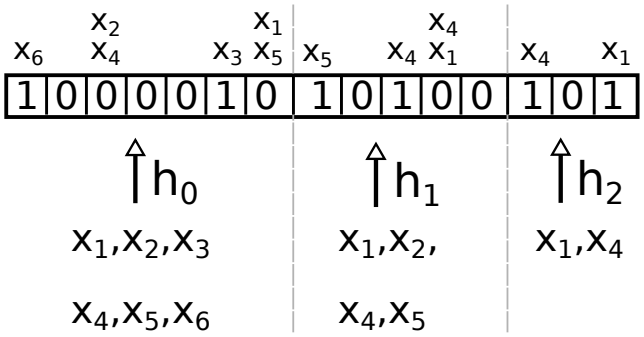

Figure 2.3: Illustration of the BBHash algorithm. The set $\{x_1, \ldots, x_6\}$ are the input elements to be hashed. The first Bloom filter is the 7 leftmost values in the bit array above. The second (resp. third) Bloom filter are the following 5 (resp. 3) values. A different hash function is used each time (h_0, h_1, h_2). Here, no additional hash tables as in the final level, x_1 and x_4 do not collide.

least one order of magnitude faster construction [LRCP17]. The resulting function has slightly higher space usage and faster or comparable query times than other contemporary methods.

It turns out that the BBHash algorithm is not entirely novel. A similar idea was proposed in another software (Meraculous [CHS+11], coincidentally for storing de Bruijn graphs) and another perfect hash function [MSSZ14]. Yet, at the time of designing BBHash, no existing implementation was capable of handling billions of keys. This was likely due to suboptimal engineering, but also and importantly, some missing ingredients in available methods. BBHash therefore provided a novel and effective mix of ingredients that yields a very effective minimal perfect hash function in practice.

Why is it a K problem Framed in the following way, the research around BBHash is a K problem: "Can one re-use previously published ideas (e.g. from Meraculous) to improve the state of the art of minimal perfect hashing?". Of course, if it is framed in another way it could become a U problem, e.g. "Can one come up with a way to beat the best existing minimal perfect hashing method?". I cannot tell what was inside the mind of Antoine and Guillaume when they designed BBHash, however I have a strong suspicion that they indeed treated the problem as a U problem and did not know that the ideas they came up with were partly previously published. However, in the way we framed the contribution in our article, we made abundantly clear that BBHash re-uses previously published ingredients, making the whole contribution appear as the outcome of a K problem.

2.4.3 BCALM2

As a final example of a K problem, we also expand on a method briefly mentioned in the previous Chapter. BCALM1, the first version of BCALM, was a single-threaded algorithm that constructs the compacted de Bruijn graph.

- The algorithmic ideas of BCALM2 [CLM16] were contributed by Paul Medvedev, Antoine Limasset, and myself. Unlike the other projects presented above, I realized

most of the implementation of BCALM2 in C++, building on top of code provided by Antoine Limasset for BCALM1, and code provided by Paul Medvedev for the a module. In particular, BCALM2 uses the GATB [DRC+14] C++ library for k-mers and de Bruijn graphs, and is also integrated as a component within it. This work benefited greatly from previous developments funded by an Inria technological grant (ADT GATB). Yet BCALM2 was mainly carried on my own research time at CNRS.

Although it was sketched in the previous chapter, we provide the definition of a compacted de Bruijn graph here. A path in a de Bruijn graph is said to be an *unitig* if it is either a single node, or for each internal node x of the path (i.e. not the extremities of the path), then the in-degree and the out-degree of x is 1, the out-degree of the first node of the path is 1, and the in-degree of the last node of the path is 1. A unitig is said to be maximal if it cannot be extended by a vertex on either side. The compacted de Bruijn graph is the graph obtained from a classical de Bruijn graph by considering all maximal unitigs as nodes, and edges remain the $(k-1)$ exact suffix-prefix overlaps between unitigs, i.e. the same overlap rule as in classical de Bruijn graphs.

The main idea behind BCALM1 was to partition k-mers according to their minimizers and write those partitions on disk. Then, in a particular order, process each partition and construct the compacted de Bruijn graph of it. Finally each of the unitigs of a partition was either output or appended to a later partition. It is clear that this algorithm is not straightforward to parallelize given that the partitions needed to be processed in a particular order.

In BCALM2 we found that by doubling a certain subset of k-mers and writing those to two partitions, then the partitions could all be processed and compacted independently and in parallel. Some examples of instances that illustrate this process are shown in Figure 2.4. However, a last post-processing stage would be necessary to 'glue' the resulting unitigs because some of them contain duplicated k-mers at their extremities. The key aspect is that the final glue is a relatively inexpensive operation that can also be performed in parallel. The whole pipeline is summarized in Figure 2.5.

BCALM2 is still to this date one of the most effective implementation for constructing the compacted de Bruijn graph of large sets of reads. It can process human genome reads in around an hour with a couple GB of memory, and was engineered to support even larger samples. A close competitor is Bifrost [HM20], which was tested up to human-sized genomes. Bifrost also supports several additional features, among which: dynamicity, i.e. nodes can be inserted or deleted, and graph indexing, i.e. nodes can be queried. For constructing de Bruijn graphs of (collections of) reference genomes, several recent techniques have appeared using a completely different approach, e.g. TwoPaCo and Cuttlefish [MPM17, KP20].

Why is it a K problem Whether BCALM1 stems from a U problem is an interesting question that I will not tackle here. I believe it is easier to argue that coming up with BCALM2 is a K problem, framed as follows: given that we know of a sequential algorithm that constructs the compacted de Bruijn graph (BCALM1), is it possible to design another algorithm that runs using multiple threads? In some cases, when the initial single-threaded algorithm performs steps that are independent, the answer to such a question can be as trivial as transforming an iterative 'for' loop into

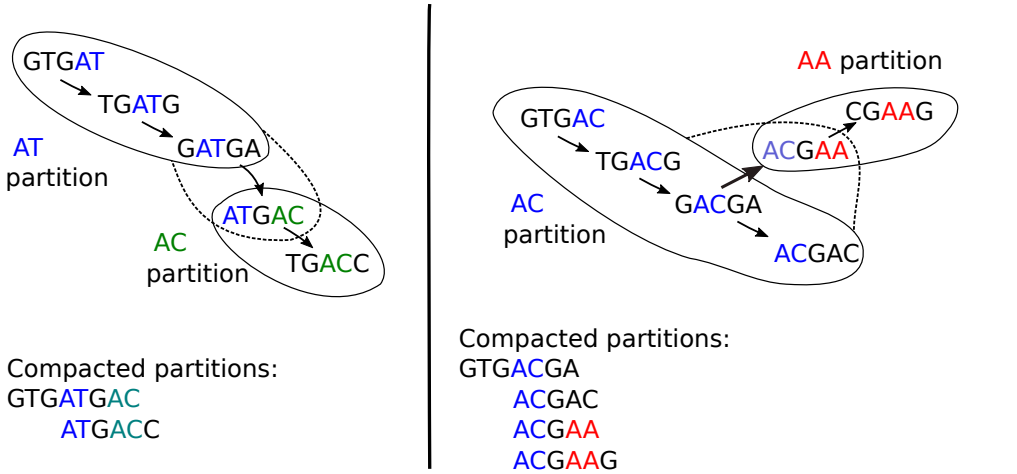

Figure 2.4: Two examples of de Bruijn graphs that illustrate the BCALM2 algorithm. Nodes are partitioned according to their minimizer (in color). Partitions are denoted by solid areas. Dotted areas indicate the addition of another k-mer to a partition due to its prefix having a different minimizer than its suffix. Then, each partition is compacted independently, resulting in the sequences shown at the bottom of each panel.

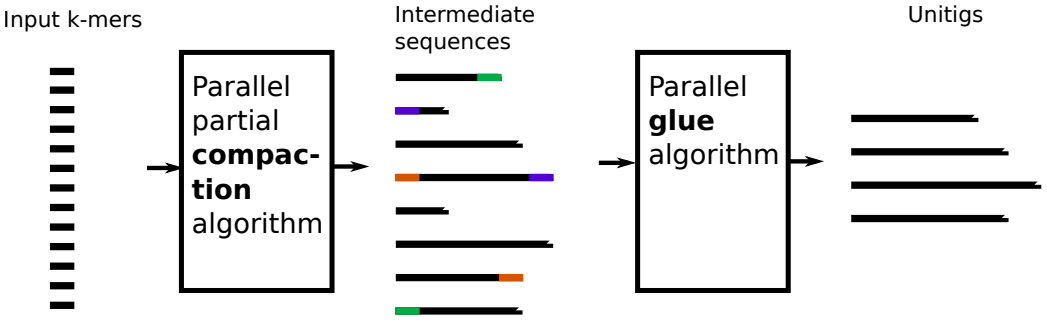

Figure 2.5: Illustration of the inputs/outputs and the two building blocks of the BCALM2 algorithm. The parallel partial compaction algorithm is illustrated by Figure 2.4. Each color at the extremity of a sequence corresponds to a particular k-mer. The parallel glue algorithm conceptually merges two sequences at their identically colored endpoints.

a 'parallel for'. Unfortunately for BCALM this was not the case, as we had to significantly redesign the algorithm and even introduce novel algorithmic ingredients in order to make it run in parallel. That said, there were little doubts that *some* way could be found to make it parallel, although it was not obvious at first how to best do it. This is in my opinion a clue that the problem was truly a K problem and also not just an engineering question.

2.5 Some U problems

2.5.1 Parallel gzip decompression

We now move to U territory with our first illustration of a U problem. Here we will present an algorithm that decompresses gzip-compressed files in parallel.

The main algorithmic idea was contributed by Mael Kerbiriou, who has also implemented it. I proposed to him this project and showed a preliminary proof of concept hinting that the task was maybe possible. It was originally a side-project of programming a faster parsing of compressed read files, however this became a project of its own given its potential for more broad interest. The work was funded by an Inria technological grant (ADT) for creating a C++ library for handling alignment seeds.

Before we give algorithmic details, let us briefly outline why this problem has a U flavor. There exists a parallel gzip *compression* algorithm, implemented in the `pigz` program (`https://github.com/madler/pigz`). However, `pigz` does not decompress in parallel. There is a fundamental reason to this: data dependency. A compressed block makes references to the previous block, therefore one needs to have decompressed block i before decompressing block $i+1$. Thus it appears impossible to perform decompression in parallel.

Now I will explain some of the algorithmics of gzip and the parallel gzip decom-
- pression method we developed in `pugz` [KC19]. For formal definitions, see the article [KC19]; here I will focus on the intuition. A gzipped file consists of a header, then a series of compressed blocks (Figure 2.6). Each compressed block can be seen as a list of elements, where each element is either a plaintext character or a reference to a previous position along with a length, indicating to copy a certain number of characters from that previous position. An additional layer is applied to the list of elements so that it is compressed, in part using Huffman coding. The interesting part concerns the elements that are references. The references instruct to copy characters that were previously decompressed (which may themselves have been decompressed via a reference, and so on ...). A reference copy is limited to within the 32 KB window that precedes the current character; yet this window may overlap with one or more previous block(s). Thus, it is nearly always the case that the beginning of a block has references to the previous block.

Our method `pugz` [KC19] proceeds in two passes. In the first pass, the compressed file is partitioned into as many chunks as there are threads. I will skip the technicalities regarding how to reliably split a gzipped file into chunks and not fall inside the middle of an encoded element, yet the take-away is that this operation can be robustly done for ASCII text files. Then, the first chunk can be decompressed as-is as it starts from the beginning of the file. Each other chunk is a random access

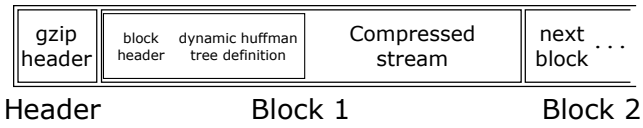

Figure 2.6: Simplified schematic of the gzip format.

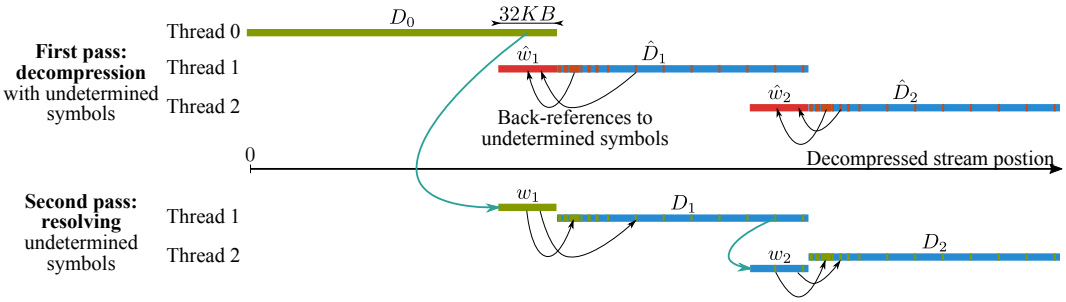

Figure 2.7: Illustration of the pugz algorithm. (Figure credit: Mael Kerbiriou)

with an unknown context, so none of the elements that are references to positions before the chunk can be reliably decompressed. The key idea is to create a table of 32,768 symbols, where each symbol corresponds to an unknown character at a certain position before the chunk, and propagate those symbols throughout the chunk, following the chain of references. Then, the second pass of `pugz` goes through the whole decompressed file from start to finish and elucidates the unknown characters, given that the end of the first decompressed chunk allows to fully decompress the second chunk, etc. This effectively completes the algorithm. The overall process is depicted in Figure 2.7.

Our approach is limited to text files as the splitting of compressed files into chunks was not reliable with arbitrary binary files. Also, the implementation is not yet robust to decompress some variants of the gzip format (multipart gzip files), yet this is not a fundamental limitation of the method and more of an engineering aspect.

Within this project I also experimented with another research direction, namely performing random accesses in gzipped files. It is also an a priori impossible task due to the dependencies between compressed blocks as we saw above (Figure 2.8). However, it turned out that those dependencies tend 'evaporate' after a while, depending on the compression level and the amount of repetitions within the uncompressed file.

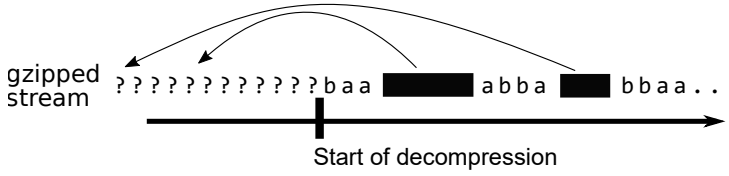

Figure 2.8: Description of the issue with performing a random access inside of a gzipped file. There exist references (black boxes) to characters that are before the start of the stream.

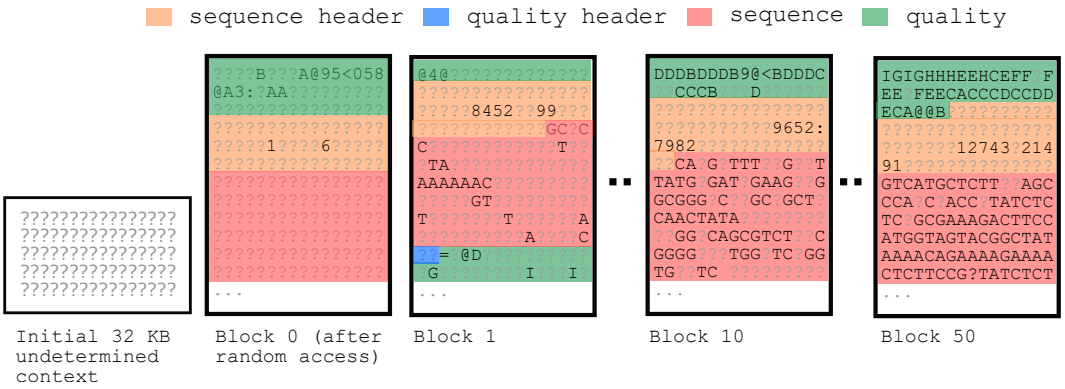

Figure 2.9: Illustration of a decompression starting from the middle of a real gzip-compressed FASTQ file. Initially the context is unknown, therefore references to previous blocks cannot be determined and are reflected by '?' characters. However, as decompression continues there exist less and less unresolved references in the following blocks and nearly all DNA sequences characters become correctly decompressed. However, quality values are still not resolved. This is due to quality values being more repetitive than DNA sequences across the file.

There was a possibility that random access may be possible. Figure 2.9 illustrates this effect on a real FASTQ file. In the `pugz` article, we reported on these findings and concluded that although it was possible to decompress nearly perfectly after a certain number of blocks following a random access, the decompression was never fully perfect even at low compression levels. This defeats the purpose of providing a general-purpose robust random access tool, although it may have an usage in forensics [Bro13]. Yet I consider this investigation of random accesses in gzipped files unsuccessful so far.

Why is it a U problem The task of performing parallel decompression of a gzipped file was previously thought to be impossible (https://github.com/madler/pigz/issues/36#issuecomment-249041503). There has never been any document attempt. Thus I believe this makes the research around `pugz` tackle a U problem. Note that this does not mean that the research is high-impact: indeed the article is still to this date almost never cited.

2.5.2 Minia

As a final example of a U problem, we will expand on a method mentioned in the previous Chapter. Minia is an exact navigational structure for the Bruijn graph based on a Bloom filter.

It is joint work with Guillaume Rizk, that I contributed at the time of my PhD defense, thus Minia did not make it into my PhD manuscript. The work was not funded by a particular grant, however my thesis was funded by governmental funds. The work sparked several other projects, such as the GATB library, which itself led to several other contributions from my former team or other teams in France:

- DiscoSNP [URL+15] for SNP detection, MindTheGap [RGCL14] for insertion de-

tection, Leon [BLL+15] for read compression, Bloocoo [BLLR14] for short read correction, LoRDEC [SR14] for long read correction.

To get an intuition of the principle behind Minia [CR13], refer to the bottom panel of Figure 1.2 which describes how a de Bruijn graph can be represented using a Bloom filter. The original idea for doing this representation was proposed in Pell *et al* [PHCK+12]. Yet, the representation is inexact: it may include nodes that are false positives, by the very nature of the Bloom filter. Minia extended this idea by detecting and storing those false positives inside a simple hash table. Some analysis was necessary to determine the optimal size of the Bloom filter in order to have not too many false positives to store separately, while at the same time balancing with having a small enough Bloom filter.

The efficiency of the Minia representation allowed to perform for the first time a genome assembly of a complete human genome on a desktop computer, and the process took less than a day. Although the contiguity of the resulting assembly is bad by long read standards, it was comparable to the quality of human genome assemblies from other genome assemblers at the time, at the contig level.

Why is it a U problem Representing a de Bruijn graph using less bits than the Conway-Bromage representation was thought to be impossible, by virtue of their information-theoretic lower bound. However, we showed with Minia, and later formalized in a finer analysis [CLJ+14], that this lower bound does not hold for a class of k-mer sets that satisfies the *spectrum-like* property, i.e. where k-mers come from some underlying set of longer strings. Thus the research work done in Minia satisfies the U condition upon the prior belief that the task was thought to be impossible. One potential objection is that Minia appeared only shortly after the Conway-Bromage article, and that the competing method BOSS [BOSS12] was also developed the same year, which may hint that both Minia and BOSS were regular and potentially even immediate contributions. I would argue that their widespread use today and lack of more efficient methods casts a doubt on this objection.

Conclusions and perspectives

In this document I have presented two Chapters that retrace some of my previous works in the domain of sequence bioinformatics. The first Chapter presented a collective research initiative on data structures for storing k-mers and de Bruijn graphs, and the second Chapter presented a collection of collaborative works related to short-read sequence data analysis.

The field of sequence bioinformatics is strongly tied to the availability of biological data, in the form of sequences. Historically, the data has typically been of one of these two types: 1) complete genes or genomes, and 2) short-read sequencing data where each sequence is partial. Nowadays, the production of biological data is rapidly changing towards a third type: 3) long-read sequencing data where each sequence potentially contains an entire gene up to an entire chromosome. The advent of this third type of sequencing data opens several research directions, some of which I plan to tackle in the future years.

There is of course more than just the length and content of the sequencing data at play. For instance, the error rate of 3rd generation sequencing data is markedly higher than that of second generation short read sequencing data. However, very recent updates to third generation sequencing technologies provide highly accurate long reads, e.g. with PacBio HiFi data. Also the throughput of the data is important as it has an immediate consequence on the depth of sequencing, which is critical in applications such as transcriptomics or pangenomics, where some sequences dominate others in terms of abundance.

In light of these considerations, I plan to tackle the following areas of research in the future: a) elucidate the missing variability inside metagenome assembly using long reads; b) perform long-read genome assembly very efficiently using accurate long reads; c) perform large-scale analysis of variants in population using k-mer matrices.

For a), my postdoc Riccardo Vicedomini is currently developing a pipeline to separate bacterial strains within sequenced metagenomes using long reads. Several methods for achieving this goal using short-read datasets have already been proposed, but none using long reads. As sequencing using accurate long reads is becoming more prevalent in metagenome sequencing projects [BKT+21], it would be desirable to take advantage of long reads to provide higher-quality metagenome assemblies. Strain separation is important not only to characterize all the organisms present in a given sample, but also to identify those that play specific functional roles in an environment. For example, there are functional differences in pathogenecity of strains of *Escherichia Coli* in the intestinal microbiota [FWC+11, CRPM+10].

Therefore, there is a possibility that strain separation will lead to the discovery of novel biological roles of strains. A prototype implementation of strain separation using long reads is being implemented, with very promising results on low-complexity microbiomes. At the moment however the method has some limitations related to its use of haplotype phasing, which was not specially designed to our analysis case (rather, to diploid phasing). Although we have shown that these issues are not critical in low complexity metagenomes, I would like to continue working around these limitations in order to improve strain separation for more complex data sets. Another promising direction is the use of long and accurate reads such as PacBio HiFi to greatly improve the contiguity and accuracy of metagenomic assemblies.

For b), along with collaborators Barış Ekim and Bonnie Berger, we are investigating an intriguing new data structure for representing de Bruijn graphs using minimizers. Long-read sequencing data has, to date, not been amenable to be assembled using classical de Bruijn graphs. There have been some variants of de Bruijn graphs designed in e.g. the ABruijn and the wtdbg2 assemblers [LYK+16, RL20], geared towards the assembly of long and error-prone reads. Fundamentally it is difficult to enumerate all genomic k-mers with a k value large enough (e.g. with $k \geq 30$) from error-prone reads, due to the high error rate leading to most k-mers containing at least one error. That said, a few long enough stretches of non-erroneous bases inside a read are sufficient to find anchors to a reference genome, which is in fact the strategy adopted by most long-read aligners such as minimap2 [Li18] (albeit with short k-mer sizes). Similarly, the aforementioned techniques of ABruijn and wtdbg2 resorted to the use of smaller k values (in ABruijn) or rely on inexact alignment (in wtdbg2) to build the nodes of their de Bruijn graph variants. In our current research plan, we significantly deviate from previous directions and instead focus on constructing a de Bruijn graph over a special type of k-mers, over a different alphabet than the alphabet of nucleotides. The trade-off is that constructing such special k-mers cannot be reliably done on error-prone long reads, and instead requires highly accurate long reads. Fortunately, an emerging technology (PacBio HiFi) produces such long reads, and our preliminary tests indicate that such data is very amenable to the construction of those special de Bruijn graphs. We will continue investigating this direction in the future as it brings significant performance improvements compared to existing approaches based on string graphs [CCF+21, NWR+20].

For c), k-mer matrices is an important object for many applications in the analysis of multiple samples of sequencing data. It consists in a matrix where the rows are each k-mer present in at least one sample, the columns correspond to samples, and the values are the abundances of each k-mer in each sample. This matrix allows several types of analysis: in RNA-seq for the detection of differentially expressed variations [APC+17], in genomics and RAD-Seq with the detection of SNPs without references [URL+15], in bacterial genomics with bacterial GWAS [JLT+18]. However, the construction of k-mer matrices is difficult in terms of computation time and of memory space resources and risk. This project consists in producing an efficient method via algorithmic improvements. A pre-print of our current work with my PhD student Teo Lemane, co-advised with Pierre Peterlongo, is available [LMCP21]. We will apply these techniques to the analysis of sequencing data from a cohort of patients with Alzheimer's disease; to determine regions of the genomes associated with the disease, with a particular interest in structural variants. This is a collabora-

tion with a team from the Institut Pasteur de Lille specializing in the genomics of Alzheimer's disease.

Although the three aforementioned research directions may appear to correspond to 1-2 years projects leading to a single article each, I will argue instead that they are the first stones in three different research programmes for which I do not yet have a complete visibility.

Brief CV

Education

• PhD in Bioinformatics, Ecole Normale Supérieure de Cachan, Rennes, France (2008 – 2012)
• Masters in Computer Science, Ecole Normale Supérieure de Cachan, Brittany, France (2005 – 2008)

Employment

• Group Leader, Institut Pasteur, Paris, France (2019 – present)
• Junior full-time CNRS researcher, University of Lille, CRIStAL laboratory, France (2014 – present)
• Postdoctoral Researcher, Pennsylvania State University, USA (2012 – 2014)

Publications (as of June 2021)
• Number of publications: 43
• H-index: 23
• I10: 32
• Citations: 4278 (Google Scholar)

Current group members (as of June 2021)
• Yoann Dufresne (engineer, Hub Pasteur)
• Riccardo Vicedomini (postdoc)
• Luca Denti (postdoc)
• Teo Lemane (PhD student, Inria Rennes, co-advised with Pierre Peterlongo)
• Camila Duitama (PhD student, PRAIRIE, EDITE, co-advised with Hugues Richard)
• Luc Blassel (PhD student, PRAIRIE, CdV)

Bibliography

[AKP+20] Bahar Alipanahi, Alan Kuhnle, Simon J Puglisi, Leena Salmela, and Christina Boucher. Succinct Dynamic de Bruijn Graphs. *Bioinformatics*, 2020. 19

[APC+17] Jérôme Audoux, Nicolas Philippe, Rayan Chikhi, Mikael Salson, Mélina Gallopin, Marc Gabriel, Jérémy Le Coz, Emilie Drouineau, Thérèse Commes, and Daniel Gautheret. De-kupl: exhaustive capture of biological variation in rna-seq data through k-mer decomposition. *Genome biology*, 18(1):1–15, 2017. 38

[ASSP18] Fatemeh Almodaresi, Hirak Sarkar, Avi Srivastava, and Rob Patro. A space and time-efficient index for the compacted colored de Bruijn graph. *Bioinformatics*, 34(13):i169–i177, 2018. 21, 22

[BBG+15] C. Boucher, A. Bowe, T. Gagie, S. J. Puglisi, and K. Sadakane. Variable-Order de Bruijn Graphs. In *2015 Data Compression Conference*, pages 383–392, 2015. 24

[BBK20] Karel Břinda, Michael Baym, and Gregory Kucherov. Simplitigs as an efficient and scalable representation of de Bruijn graphs. *bioRxiv*, 2020. 20

[BKT+21] Derek M. Bickhart, Mikhail Kolmogorov, Elizabeth Tseng, Daniel M. Portik, Anton Korobeynikov, Ivan Tolstoganov, Gherman Uritskiy, Ivan Liachko, Shawn T. Sullivan, Sung Bong Shin, Alvah Zorea, Victòria Pascal Andreu, Kevin Panke-Buisse, Marnix H. Medema, Itzik Mizrahi, Pavel A. Pevzner, and Timothy P.L. Smith. Generation of lineage-resolved complete metagenome-assembled genomes by precision phasing. *bioRxiv*, 2021. 37

[BLL+15] Gaëtan Benoit, Claire Lemaitre, Dominique Lavenier, Erwan Drezen, Thibault Dayris, Raluca Uricaru, and Guillaume Rizk. Reference-free compression of high throughput sequencing data with a probabilistic de bruijn graph. *BMC bioinformatics*, 16(1):1–14, 2015. 36

[BLLR14] Gaëtan Benoit, Dominique Lavenier, Claire Lemaitre, and Guillaume Rizk. Bloocoo, a memory efficient read corrector. In *European conference on computational biology (ECCB)*, 2014. 36

[BNA+12] Anton Bankevich, Sergey Nurk, Dmitry Antipov, Alexey A Gurevich, Mikhail Dvorkin, Alexander S Kulikov, Valery M Lesin, Sergey I

Nikolenko, Son Pham, Andrey D Prjibelski, et al. SPAdes: a new genome assembly algorithm and its applications to single-cell sequencing. *Journal of computational biology*, 19(5):455–477, 2012. 14, 23

[BOSS12] Alexander Bowe, Taku Onodera, Kunihiko Sadakane, and Tetsuo Shibuya. Succinct de Bruijn graphs. In *International workshop on algorithms in bioinformatics*, pages 225–235. Springer, 2012. 15, 19, 36

[Bro13] Ralf D Brown. Improved recovery and reconstruction of deflated files. *Digital Investigation*, 10:S21–S29, 2013. 35

[BW94] Michael Burrows and D. J. Wheeler. A block-sorting lossless data compression algorithm. Report 124, Digital Systems Research Center, Palo Alto, CA, USA, May 1994. 19

[CB11] Thomas C Conway and Andrew J Bromage. Succinct data structures for assembling large genomes. *Bioinformatics*, 27(4):479–486, 2011. 15, 17

[CCF+21] Haoyu Cheng, Gregory T Concepcion, Xiaowen Feng, Haowen Zhang, and Heng Li. Haplotype-resolved de novo assembly using phased assembly graphs with hifiasm. *Nature Methods*, 18(2):170–175, 2021. 38

[CHM19] Rayan Chikhi, Jan Holub, and Paul Medvedev. Data structures to represent sets of k-long DNA sequences. *arXiv preprint arXiv:1903.12312*, 2019. 9, 11, 15, 20, 22, 24

[CHS+11] Jarrod A Chapman, Isaac Ho, Sirisha Sunkara, Shujun Luo, Gary P Schroth, and Daniel S Rokhsar. Meraculous: de novo genome assembly with short paired-end reads. *PloS one*, 6(8):e23501, 2011. 30

[CLJ+14] Rayan Chikhi, Antoine Limasset, Shaun Jackman, Jared T Simpson, and Paul Medvedev. On the representation of de Bruijn graphs. In *International conference on Research in computational molecular biology*, pages 35–55. Springer, 2014. 19, 20, 21, 36

[CLM16] Rayan Chikhi, Antoine Limasset, and Paul Medvedev. Compacting de Bruijn graphs from sequencing data quickly and in low memory. *Bioinformatics*, 32(12):i201–i208, 2016. 21, 30

[CP08] Mark J Chaisson and Pavel A Pevzner. Short read fragment assembly of bacterial genomes. *Genome research*, 18(2):324–330, 2008. 13

[CR13] Rayan Chikhi and Guillaume Rizk. Space-efficient and exact de Bruijn graph representation based on a Bloom filter. *Algorithms for Molecular Biology*, 8(1):1–9, 2013. 15, 17, 19, 36

[CR20] Bastien Cazaux and Eric Rivals. Hierarchical overlap graph. *Information Processing Letters*, 155:105862, 2020. 24

[CRPM+10] Gabriel Cuevas-Ramos, Claude R Petit, Ingrid Marcq, Michèle Boury, Eric Oswald, and Jean-Philippe Nougayrède. Escherichia coli induces dna damage in vivo and triggers genomic instability in mammalian

cells. *Proceedings of the National Academy of Sciences*, 107(25):11537–11542, 2010. 37

[DDGG13] Sebastian Deorowicz, Agnieszka Debudaj-Grabysz, and Szymon Grabowski. Disk-based k-mer counting on a PC. *BMC bioinformatics*, 14(1):1–12, 2013. 21

[DRC+14] Erwan Drezen, Guillaume Rizk, Rayan Chikhi, Charles Deltel, Claire Lemaitre, Pierre Peterlongo, and Dominique Lavenier. Gatb: genome assembly & analysis tool box. *Bioinformatics*, 30(20):2959–2961, 2014. 31

[ENS+20] Jordan M. Eizenga, Adam M. Novak, Jonas A. Sibbesen, Simon Heumos, Ali Ghaffaari, Glenn Hickey, Xian Chang, Josiah D. Seaman, Robin Rounthwaite, Jana Ebler, Mikko Rautiainen, Shilpa Garg, Benedict Paten, Tobias Marschall, Jouni Sirén, and Erik Garrison. Pangenome graphs. *Annual Review of Genomics and Human Genetics*, 21(1):139–162, 2020. PMID: 32453966. 14

[FM00] Paolo Ferragina and Giovanni Manzini. Opportunistic data structures with applications. In *Proceedings 41st Annual Symposium on Foundations of Computer Science*, pages 390–398. IEEE, 2000. 20

[FWC+11] Christina Frank, Dirk Werber, Jakob P Cramer, Mona Askar, Mirko Faber, Matthias an der Heiden, Helen Bernard, Angelika Fruth, Rita Prager, Anke Spode, et al. Epidemic profile of shiga-toxin–producing escherichia coli o104: H4 outbreak in germany. *New England Journal of Medicine*, 365(19):1771–1780, 2011. 37

[HM20] Guillaume Holley and Páll Melsted. Bifrost: highly parallel construction and indexing of colored and compacted de Bruijn graphs. *Genome biology*, 21(1):1–20, 2020. 22, 31

[HWS16] Guillaume Holley, Roland Wittler, and Jens Stoye. Bloom Filter Trie: an alignment-free and reference-free data structure for pan-genome storage. *Algorithms for Molecular Biology*, 11(1):1–9, 2016. 22

[HWSH17] Guillaume Holley, Roland Wittler, Jens Stoye, and Faraz Hach. Dynamic alignment-free and reference-free read compression. In *International Conference on Research in Computational Molecular Biology*, pages 50–65. Springer, 2017. 14

[ICT+12] Zamin Iqbal, Mario Caccamo, Isaac Turner, Paul Flicek, and Gil McVean. De novo assembly and genotyping of variants using colored de Bruijn graphs. *Nature genetics*, 44(2):226–232, 2012. 23

[JLT+18] Magali Jaillard, Leandro Lima, Maud Tournoud, Pierre Mahé, Alex Van Belkum, Vincent Lacroix, and Laurent Jacob. A fast and agnostic method for bacterial genome-wide association studies: Bridging the gap between k-mers and genetic events. *PLoS genetics*, 14(11):e1007758, 2018. 38

[KC19] Maël Kerbiriou and Rayan Chikhi. Parallel decompression of gzip-compressed files and random access to dna sequences. In *2019 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 209–217. IEEE, 2019. 33

[KMD+20] Mikhail Karasikov, Harun Mustafa, Daniel Danciu, Marc Zimmermann, Christopher Barber, Gunnar Ratsch, and André Kahles. Metagraph: Indexing and analysing nucleotide archives at petabase-scale. *bioRxiv*, 2020. 20

[KP20] Jamshed Khan and Rob Patro. Cuttlefish: Fast, parallel, and low-memory compaction of de bruijn graphs from large-scale genome collections. *bioRxiv*, 2020. 31

[Li18] Heng Li. Minimap2: pairwise alignment for nucleotide sequences. *Bioinformatics*, 34(18):3094–3100, 2018. 38

[LLL+15] Dinghua Li, Chi-Man Liu, Ruibang Luo, Kunihiko Sadakane, and Tak-Wah Lam. MEGAHIT: an ultra-fast single-node solution for large and complex metagenomics assembly via succinct de Bruijn graph. *Bioinformatics*, 31(10):1674–1676, 2015. 19, 23

[LMCP21] Teo Lemane, Paul Medvedev, Rayan Chikhi, and Pierre Peterlongo. kmtricks: Efficient construction of bloom filters for large sequencing data collections. *bioRxiv*, 2021. 38

[LRCP17] Antoine Limasset, Guillaume Rizk, Rayan Chikhi, and Pierre Peterlongo. Fast and scalable minimal perfect hashing for massive key sets. *arXiv preprint arXiv:1702.03154*, 2017. 16, 21, 29, 30

[LYK+16] Yu Lin, Jeffrey Yuan, Mikhail Kolmogorov, Max W Shen, Mark Chaisson, and Pavel A Pevzner. Assembly of long error-prone reads using de bruijn graphs. *Proceedings of the National Academy of Sciences*, 113(52):E8396–E8405, 2016. 19, 38

[LZR+10] Ruiqiang Li, Hongmei Zhu, Jue Ruan, Wubin Qian, Xiaodong Fang, Zhongbin Shi, Yingrui Li, Shengting Li, Gao Shan, Karsten Kristiansen, et al. De novo assembly of human genomes with massively parallel short read sequencing. *Genome research*, 20(2):265–272, 2010. 13

[MBN+17] Martin D Muggli, Alexander Bowe, Noelle R Noyes, Paul S Morley, Keith E Belk, Robert Raymond, Travis Gagie, Simon J Puglisi, and Christina Boucher. Succinct colored de Bruijn graphs. *Bioinformatics*, 33(20):3181–3187, 2017. 23

[Med] Paul Medvedev. The theoretical analysis of sequencing bioinformatic algorithms. in preparation. 23

[MIG+20] Camille Marchet, Zamin Iqbal, Daniel Gautheret, Mikaël Salson, and Rayan Chikhi. REINDEER: efficient indexing of k-mer presence and abundance in sequencing datasets. *Bioinformatics*, 36(Supplement_1):i177–i185, 2020. 23, 24, 27

[MK11] Guillaume Marçais and Carl Kingsford. A fast, lock-free approach for efficient parallel counting of occurrences of k-mers. *Bioinformatics*, 27(6):764–770, 2011. 21

[MKL21] Camille Marchet, Mael Kerbiriou, and Antoine Limasset. Efficient exact associative structure for sequencing data. *bioRxiv*, 2021. 21, 22, 27

[MKS10] Jason R. Miller, Sergey Koren, and Granger Sutton. Assembly algorithms for next-generation sequencing data. *Genomics*, 95(6):315–327, 2010. 15, 16

[MPM17] Ilia Minkin, Son Pham, and Paul Medvedev. Twopaco: An efficient algorithm to build the compacted de bruijn graph from many complete genomes. *Bioinformatics*, 33(24):4024–4032, 2017. 31

[MS18] Swati C Manekar and Shailesh R Sathe. A benchmark study of k-mer counting methods for high-throughput sequencing. *GigaScience*, 7(12), 10 2018. 21

[MSSZ14] Ingo Müller, Peter Sanders, Robert Schulze, and Wei Zhou. Retrieval and perfect hashing using fingerprinting. In *International Symposium on Experimental Algorithms*, pages 138–149. Springer, 2014. 30

[NWR+20] Sergey Nurk, Brian P Walenz, Arang Rhie, Mitchell R Vollger, Glennis A Logsdon, Robert Grothe, Karen H Miga, Evan E Eichler, Adam M Phillippy, and Sergey Koren. Hicanu: accurate assembly of segmental duplications, satellites, and allelic variants from high-fidelity long reads. *Genome research*, 30(9):1291–1305, 2020. 38

[PBJP17] Prashant Pandey, Michael A Bender, Rob Johnson, and Rob Patro. A general-purpose counting filter: Making every bit count. In *Proceedings of the 2017 ACM international conference on Management of Data*, pages 775–787, 2017. 22

[PDL+17] Rob Patro, Geet Duggal, Michael I Love, Rafael A Irizarry, and Carl Kingsford. Salmon provides fast and bias-aware quantification of transcript expression. *Nature methods*, 14(4):417–419, 2017. 14

[PHCK+12] Jason Pell, Arend Hintze, Rosangela Canino-Koning, Adina Howe, James M Tiedje, and C Titus Brown. Scaling metagenome sequence assembly with probabilistic de Bruijn graphs. *Proceedings of the National Academy of Sciences*, 109(33):13272–13277, 2012. 17, 36

[PLYC10] Yu Peng, Henry CM Leung, Siu-Ming Yiu, and Francis YL Chin. IDBA–a practical iterative de Bruijn graph de novo assembler. In *Annual international conference on research in computational molecular biology*, pages 426–440. Springer, 2010. 14

[RCM20] Amatur Rahman, Rayan Chikhi, and Paul Medvedev. Disk Compression of k-mer Sets. In *20th International Workshop on Algorithms in Bioinformatics (WABI 2020)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2020. 20

[RGCL14] Guillaume Rizk, Anaïs Gouin, Rayan Chikhi, and Claire Lemaitre. Mindthegap: integrated detection and assembly of short and long insertions. *Bioinformatics*, 30(24):3451–3457, 2014. 35

[RHH+04] Michael Roberts, Wayne Hayes, Brian R Hunt, Stephen M Mount, and James A Yorke. Reducing storage requirements for biological sequence comparison. *Bioinformatics*, 20(18):3363–3369, 2004. 27

[RL20] Jue Ruan and Heng Li. Fast and accurate long-read assembly with wtdbg2. *Nature methods*, 17(2):155–158, 2020. 38

[RLC13] Guillaume Rizk, Dominique Lavenier, and Rayan Chikhi. DSK: k-mer counting with very low memory usage. *Bioinformatics*, 29(5):652–653, 2013. 21

[SR14] Leena Salmela and Eric Rivals. Lordec: accurate and efficient long read error correction. *Bioinformatics*, 30(24):3506–3514, 2014. 36

[SSK14] Kamil Salikhov, Gustavo Sacomoto, and Gregory Kucherov. Using cascading Bloom filters to improve the memory usage for de Brujin graphs. *Algorithms for Molecular Biology*, 9(1):1–10, 2014. 19

[SWJ+09] Jared T Simpson, Kim Wong, Shaun D Jackman, Jacqueline E Schein, Steven JM Jones, and Inanç Birol. ABySS: a parallel assembler for short read sequence data. *Genome research*, 19(6):1117–1123, 2009. 13

[URL+15] Raluca Uricaru, Guillaume Rizk, Vincent Lacroix, Elsa Quillery, Olivier Plantard, Rayan Chikhi, Claire Lemaitre, and Pierre Peterlongo. Reference-free detection of isolated snps. *Nucleic acids research*, 43(2):e11–e11, 2015. 35, 38

[YMC+12] Chengxi Ye, Zhanshan Sam Ma, Charles H Cannon, Mihai Pop, and W Yu Douglas. Exploiting sparseness in de novo genome assembly. In *BMC bioinformatics*, volume 13, pages 1–8. BioMed Central, 2012. 19

[ZB08] Daniel R Zerbino and Ewan Birney. Velvet: algorithms for de novo short read assembly using de Bruijn graphs. *Genome research*, 18(5):821–829, 2008. 13