Nouvelles approches pour l'exploitation des données de séquences génomique haut débit

Antoine Limasset

ANNÉE 2017

UNIVERSITÉ DE RENNES 1

UNIVERSITE BRETAGNE LOIRE

THÈSE / UNIVERSITÉ DE RENNES 1
sous le sceau de l'Université Bretagne Loire

pour le grade de

DOCTEUR DE L'UNIVERSITÉ DE RENNES 1

Mention : Informatique

Ecole doctorale MATISSE

présentée par

Antoine Limasset

préparée à l'unité de recherche 6074 IRISA
INSTITUT DE RECHERCHE EN INFORMATIQUE ET SYSTEMES ALEATOIRES
ISTIC

Nouvelles approches pour l'exploitation des données de sequençage haut débit

Thèse soutenue à Rennes
le 12 Juillet 2017
devant le jury composé de :

Hélène TOUZET
Directeur de recherche, CRYSTAL / rapporteur

Thierry LECROQ
Directeur de recherche, LITIS / rapporteur

Christophe KLOPP
Ingenieur de recherche, BIA / examinateur

Vincent LACROIX
Maitre de conférence, LBBE / examinateur

Gregory KUCHEROV
Directeur de recherche, LIGM / examinateur

Rumen ANDONOV
Professeur, IRISA / examinateur

Dominique LAVENIER
Directeur de recherche, IRISA / directeur de thèse

Pierre PETERLONGO
Chargé de recherche, IRISA / co-directeur de thèse

Acknowledgments

Administration

Je voudrais tout d'abord remercier l'école doctorale Matisse, l'Université Rennes 1, IRISA et INRIA ainsi que leur personnels sans qui cette thèse et plus généralement cet environnement de recherche et d'enseignement scientifique n'existeraient tout simplement pas. Je remercie doublement les personnels d'administration à qui j'ai eu affaire et à qui j'ai dû donner du fil à retordre. Une pensée spéciale à Marie Le Roïc qui n'a jamais paru effrayée par mon organisation plus que doûteuse et à l'administration de l'école doctorale pour avoir réussi à me faire remplir un dossier correctement pour la première fois en 26 ans.

Recherche Je voudrais bien évidemment remercier tous mes collaborateurs. Merci à Eric Rivals de m'avoir jeté dans le monde de la bioinformatique et de m'y avoir donné goût pendant mon stage de L3. Merci à Dominique Lavenier de m'avoir non seulement conseillé mais guidé vers un stage puis une thèse entre de très bonnes mains. Merci a Rayan Chikhi et Paul Medvedev de m'avoir lancé dans le petit monde du graphe de De Bruijn, je me demande souvent quand est-ce que je vais pouvoir en sortir. Je remercie ici, avec quelques années de retard, Rayan de m'avoir accueilli en ami plus qu'en encadrant, je garde de très bons souvenirs, peut-être un peu trop français, de ce voyage aux Etats-Unis. Je voudrais spécialement remercier ces premiers encadrants pour leur bienveillance qui m'a permis de me conforter dans ma capacité à contribuer à la formidable construction qu'est la recherche, je ne suis plus tout à fait le même depuis.

Je ne vais pas prendre assez de place pour remercier Pierre Peterlongo pour m'avoir encadré et suivi pendant plus de trois ans. Je pense avoir été une thésard difficile et malgré nos disjonctions tu m'as toujours laissé libre dans mon fonctionnement. Je ne pense pas que j'aurais pu trouver une meilleure personne pour cette aventure. Je trouve que nous avons formé une belle paire et je suis impressionné par la vitesse à laquelle les choses avancent.

Merci également à tout le groupe Colib'read, plein de grandes rencontres ont eu lieu pour moi lors de ces réunions. Si je garde la vision d'un groupe de recherche comme un groupe d'amis c'est un peu grâce à vous.

Je voulais aussi remercier Romain Feron, je pourrais me targuer sûrement longtemps que mon premier encadrement se soit si bien passé. J'espère que tu en garderas de bons souvenirs et que tu continueras dans ta lancée.

Petite pensée pour Thomas Rokicki et Dianne Puges qui ont trouvé l'assemblage génomique suffisamment intéressant pour participer à un petit projet avec moi. Je regrette un peu que celui-ci n'ait jamais abouti mais il n'est jamais trop tard.

Merci à Jean-Francois Flot de m'avoir invité à Bruxelles et d'être aussi enthousiaste. Je nous souhaite beaucoup de succès à l'avenir.

Merci à tous mes co-bureaux de m'avoir supporté, et je souhaite à Marie Chevallier une bonne et heureuse année. Plus généralement je remercie toute l'équipe GenScale de m'avoir acueilli et pour cette bonne ambiance au travail qui n'a tout simplement pas de prix.

These

Je remercie très solennellement Yannick Zakowski Hugo Bazille Camille Marchet et Pierre Peterlongo d'avoir accompli un travail digne des travaux d'Asterix: corriger ma prose. Je regrette presque de ne pas avoir compté le nombres de fautes corrigées par vos soins mais sachez que votre tâche a été colossale. Une pensée pour Pierre qui a dû passer plusieurs nuits blanches à relire les différentes versions de celle-ci, à se fouler le bras en rayant des centaines de pages. Un très très grand merci à Camille pour l'organisation de mon pot de thèse et des festivités, si j'avais moins procrastiné j'aurais pu davantage mettre la main a la pâte et j'aurais pu mettre Paris en bouteille.

Enseignement

J'ai eu le plus grand des plaisirs lors des missions d'enseignement qui m'ont été accordées par l'ISTIC. Je ne peux remercier tous mes élèves mais sait-on jamais, merci de votre interêt (parfois) et de votre sympathie (souvent), vous avez été super. Je voudrais également profiter de cette occasion pour remercier quelques-un de mes professeurs. Merci a Mohamed Rebbal de m'avoir fait comprendre l'essence de la résolution de problèmes. Merci à Nicolas De Granrut et Gérard Debeaumarché pour m'avoir fait confiance et permis d'accéder aux études supérieures auxquelles j'aspirais. Ma vie serait bien différente sans votre soutien et je travaille à rester digne de celui-ci.

Amis

Ces années à Rennes m'ont vu changer d'une manière saisissante et je vais remercier ici essentiellement des Rennais qui m'ont entouré lors de mes péripéties dans ce beau pays, que personne n'en prenne ombrage. Merci à Justine de m'avoir suivi jusque ici et d'avoir été ma première experience de vie commune. J'espère que tu garderas une place spéciale à nos élucubrations et diverses aventures farfelues en pays breton, elles ont pour ma part toute leur place dans la boite des bons souvenirs. Merci a Simon Rokicki et Bastien Pasdeloup d'avoir les premiers accueilli un Cachannais expatrié et d'avoir fait de moi un des vôtres. Je remercie tout les membres des Projet C2Silicium et UberML, pour avoir contribué au lignes les plus hilarantes de mon CV. Encore une fois je ne vais pas prendre assez de temps pour remercier suffisament la Complexiteam. Merci pour tout ces super moments, je m'excuse officiellement pour mon Warwick Top et j'accuse le mauvais matchup. Plus sérieusement quoi que l'on vous dise, sachez-le, dans mon coeur, nous sommes au moins Diamant. Merci à Joris Lamare d'avoir gardé le contact malgré la distance et d'être toujours là pour accueillir des voyageurs déglingués et pour nous proposer les meilleurs plan de la Terre, pas moins. Un des plus grand merci à Gaëtan Benoit pour avoir été mon plus proche collègue pendant toutes ces années. Pour ne citer qu'une de nos nombreuses aventures, merci d'avoir porté le projet de court-métrage sur l'assemblage génomique et sache que, quoi que le président du jury en dise, notre film

reste le meilleur. Je vais également remercier les Doudous, pour tout votre soutien, vous n'avez jamais répondu absents lorsque j'avais besoin d'aide, je serais etonné de vous en avoir rendu le dixième. Je n'oublie pas un remerciement spécial à Simon pour son aide et ses discussions précieuses sur la compilation et les serveurs Minecraft. Merci à Olivier Ruas et Hugo Bazille pour avoir essayé de me maintenir en forme le long de cette thèse. Merci à Gurvan Cabon d'avoir été le coloc le moins chiant du monde. Dur à classer je vais remercier également mes amis à quatre pattes. Merci Belial pour avoir écrit plus que moi sur mon document de thèse (notamment le fameux éedfffffffffffffffffffffffffffffffffffffffffff). Merci à Lux d'avoir été la plus gentille des chattes et mon premier animal. Merci à Jelly d'être un si bon petit tambour. Merci a BBhash de m'avoir laissé des doigts pour travailler.

Famille Je voudrais remercier mes parents sans qui rien de tout cela n'aurait été possible. Plus sérieusement, je vous remercie de m'avoir donné les clefs pour en arriver là. Merci pour tout votre amour et tout votre soutien. Tout ce que vous m'avez apporté a fait de moi quelqu'un d'entier et pour cela je vous remercie infiniment. Je voudrais finalement remercier ici Camille Marchet que je considère aujourd'hui comme la meilleure chose qui me soit arrivé. Si un des remerciements n'est pas assez developpé c'est celui-ci. Je n'aurais pas été jusqu'ici sans toi, merci pour ces bons moments.

Divers A mon grand regret ceux-ci ne pourront jamais lire ces remerciements. Merci a Nadine et Griffin, mes deux ordinateurs de travail, pour leur fidélité. Que Nadine me pardonne mon inattention, j'espère que tu vis heureux avec ton nouveau propriétaire qui a dû avoir bien du mal à réinstaller MacOS sur un Linux. Merci au Genocluster de m'avoir supporté tout ce temps, pardonne-moi si je t'en ai parfois trop demandé. Pour finir sur une note sérieuse je voudrais remercier les projets suivants et leurs collaborateurs : Libreoffice Linux Scihub Github Travis Gnu.

Abstract

Novel approaches for the exploitation of high throughput sequencing data
In this thesis we discuss computational methods to deal with DNA sequences provided by high throughput sequencers. We will mostly focus on the reconstruction of genomes from DNA fragments (genome assembly) and closely related problems. These tasks combine huge amounts of data with combinatorial problems. Various graph structures are used to handle this problem, presenting trade-off between scalability and assembly quality.

This thesis introduces several contributions in order to cope with these tasks. First, novel representations of assembly graphs are proposed to allow a better scaling. We also present novel uses of those graphs apart from assembly and we propose tools to use such graphs as references when a fully assembled genome is not available. Finally we show how to use those methods to produce less fragmented assembly while remaining tractable.

Nouvelles approches pour l'exploitation des données de séquencage haut débit

Cette thèse a pour sujet les méthodes informatiques traitant les séquences ADN provenant des séquenceurs haut débit. Nous nous concentrons essentiellement sur la reconstruction de génomes à partir de fragments ADN (assemblage génomique) et sur des problèmes connexes. Ces tâches combinent de très grandes quantités de données et des problèmes combinatoires. Différentes structures de graphe sont utilisées pour répondre à ces problèmes, présentant des compromis entre passage à l'échelle et qualité d'assemblage.

Ce document introduit plusieurs contributions pour répondre à ces problèmes. De nouvelles représentations de graphes d'assemblage sont proposées pour autoriser un meilleur passage à l'échelle. Nous présentons également de nouveaux usages de ces graphes, différent de l'assemblage, ainsi que des outils pour utiliser ceux-ci comme références dans les cas où un génome de référence n'est pas disponible. Pour finir nous montrons comment utiliser ces méthodes pour produire un meilleur assemblage en utilisant des ressources raisonnables.

Contents

Chapter 1

Introduction

"People say they want simple things, but they don't. Not always."

John D. Cook

In this chapter we first introduce the global context of DNA sequencing and genome assembly. Then we provide a high level description of the main methods used in this field. Thereafter we describe the limits and challenges faced nowadays in genome assembly. We finish this chapter by an outline of the thesis.

1.1 DNA, genomes and sequencing

Deoxyribonucleic acid or DNA is a molecule that stores biological information used in the functioning of all known living organisms. Most DNA molecules consist of two polynucleotides strands coiled around each other to form a double helix, composed of simpler component called nucleotides (Figure 1.1). Each nucleotide, or base, can be either identified as adenine (A), cytosine (C), guanine (G) or thymine (T). The bases of the two separate strands are bound together, according to base pairing rules: A with T and C with G, to make double-stranded DNA. The sequence of these four bases along the backbone encodes biological information. DNA is usually stored directly in the cytoplasm for prokaryotes, and in the nucleus and different organelles in eukaryotic cells. Depending on organisms, DNA can be circular or not, and characterize organisms through structures called chromosomes. The whole information within the DNA molecule of an organism is called its genome. Conceptually a genome can be represented as a word or a set of words on the alphabet $\{A, C, G, T\}$. Genomes are usually large sequences, a human genome has more than 3 billions bases arranged into 46 chromosomes. In certain viruses, the genome can be encoded in a closely related molecule, the RNA, instead of the DNA. Importantly, there exists a range of redundancy of the genomic information across the living. Some cells store their genomic information into a single set of unpaired chromosomes (haploidy) while others (like humans) have two copies of each chromosome (diploidy). Some species have even more than two copies (polyploidy).

Since DNA discovery in 1953 [1], genome study has shown itself to have huge implications in both academical and industrial fields like agronomy, medicine and ecology. Indeed the knowledge of the genome sequences gives a tremendous access to living organisms characteristics and properties, and is now commonly at the core of biology studies. Sequencing is the operation that consists in determining the bases sequence of a DNA molecule and to encode it on a numerical support for its analysis. Being able to access and study the Human genome [2] [3] is considered as one of the major milestone in scientific history. But this genomic information is partial and imperfect, as no sequencer is yet able to directly output a completely sequenced genome. Thus the apparition of sequencing technologies has created a whole field in computational biology to handle this incredibly discovery-promising data. Several kind of sequencing technologies exist, but they share common properties:

- They only produce fragments of our genome ("reads")
- The locations of the fragments are unknown
- The fragments may contain errors

We call those genomic fragments/substrings "reads" to reflect the fact that they were "read" from the genome. Because of the DNA structure, both strands are present and sequenced. The strands are bases complemented (A to T and C to G) and read in the

opposite way. A genome containing ACCTGC therefore may present reads as CCTG or CAGG: CAGG being the reverse-complement of CCTG read from the opposite strand. Several methods have been proposed to sequence DNA based on a wide range of technologies that will not be described here. The differences between sequencing technologies are essentially errors and read length distribution. We can define an error rate of a read by the ratio of the number of incorrectly sequenced bases by the size of the read.

The sequencing technologies are therefore categorized in three generations according to these characteristics and order of appearance.

- Sanger sequencing [4] was the first method available. It produce sequences of some kilo-bases length with a high accuracy (error rate around 1%).
- Next generation sequencing (NGS), often referred to as Illumina/Solexa sequencing [5], is the most broadly used technology. It produces shorter reads (hundreds of bases) with high accuracy ($< 1\%$). This technology presents an order of magnitude higher throughput and cheaper sequencing than the former technology.
- Third generation sequencing (TGS) is the last generation of sequencing technology, which includes Single Molecule Real Time sequencing [6] and Nanopore sequencing [7], producing very long sequences, up to hundreds kilo-bases. However, they exhibit a very high error rate (up to 30%).

In addition to those properties, each method may show different biases due to the protocol employed. We can mention that in the short reads from NGS, some regions rich in G/C bases are less covered [8] and read ends present higher error rate [9].

Sanger sequencing is almost no longer used because of its cost and low throughput. As for the third sequencing generation, the different technologies are extremely recent

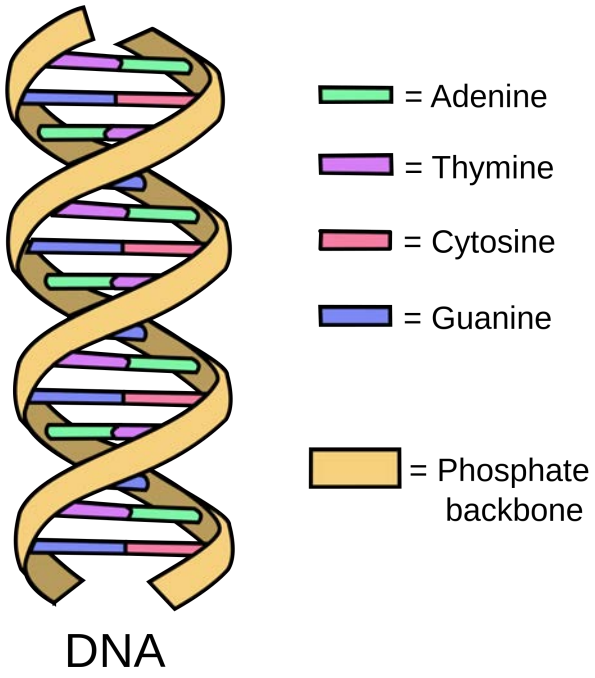

Figure 1.1: DNA representation.

and are still currently being developed in a fast moving environment. Nowadays, NGS remains the most popular technique and we will focus on this type of sequencing in this document. However TGS will eventually become widely used and the present work was partially designed to open onto the usage of such data.

For all existing technologies, the main challenge comes from the lack of information about the origin of a read. Each read could come from any strand in any position of any DNA molecule introduced in the sequencer. This absence of context and the small size of the sequences obtained, relatively to a genome size, make it difficult to use reads as such. Ideally, we would need the access to the underlying genomes in their entirety.

Since the beginning of sequencing of DNA molecules, genomes are produced by structuring and ordering reads information. Then these reconstructed genomes can be used as references. Reference genomes are the best insight we have about the one-dimensional organization of information in living cells. They give access not only to the gene sequences that lead to proteins, but also to flanking sequences that altogether impact the functioning of living beings [10]. They also reveal the inner organization of the genome such as genes relative positions or chromosomes structure. Helping understanding the genomes and organisms evolution, as well as how all the living is ruled by the encoded information. Besides, reference genomes can be seen as an entry point for biologists to use other kinds of data. For instance, they may add information about the known genes positions and functions to annotate the genome [11] [12].

Reference genome reconstruction is therefore crucial in various domains where raw, out of context reads are unusable. The task of reordering the reads to recompose the sequenced genome is called genome assembly. Tools designed for this task, called assemblers, have to make no *a priori* hypothesis over the location or the strand of each read and try to reconstruct the original sequences by ordering the reads relatively. As it will be detailed further, genome assembly is especially complex as the bases distribution is far from being uniform. Genomes present specific patterns such as large repeated sequences (repeats), regions with very specific distributions of nucleotides or extremely repeated sequences of nucleotides. Such patterns make genomes different from a uniformly distributed sequence of nucleotides. [13] shows that a human genome is largely constituted of repeated sequences of significant lengths.

Reference genomes used in this document

In this document we will present various results based on different genomes. For the sake of consistency, we choose a small number of well known and well studied genomes. The first one is the genome of the Escherichia coli (*E. coli*). *E. coli* is a bacteria with a genome of 4.6 megabase pairs constituted of one circular chromosome. The second one is the genome of Caenorhabditis elegans (*C. elegans*). *C. elegans* is a nematode and was the first multicellular organism to have its whole genome sequenced. Its genome counts 6 chromosomes and is 100 megabase pairs long. Pictures of the two organisms are shown on Figure 1.2. The third genome is the human reference genome. The version used was GRCh38, accessible at `https://www.ncbi.nlm.nih.gov/grc/human`. The genome counts 23 chromosomes and is 3,234 megabase pairs long.

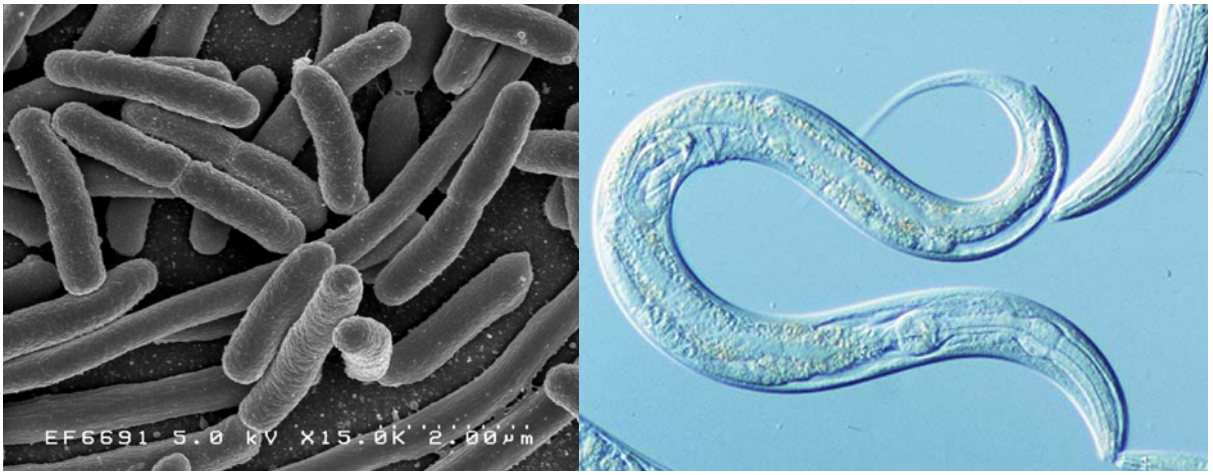

Figure 1.2: *E. coli* bacteria and *C. elegans* worm from `https://upload.wikimedia.org/wikipedia/commons/3/32/EscherichiaColi_NIAID.jpg` and `http://www.socmucimm.org/wp-content/uploads/2014/06/C.-elegans.jpg`

.

"DNA, genomes and sequencing" core messages:

- Genomes are large sequences of "ACTG", whose knowledge is essential to biological studies
- To access this information, we use machines called sequencers
- We are not able to obtain a whole genome directly out of the sequencers but only "reads" that can be seen as fragments of the genome
- Reads are way smaller than the genome and they contain errors
- The task of recovering the original genome from the "reads" is called genome assembly

1.2 Genome assembly

Two kinds of genome assembly may be distinguished: reference guided assembly and *de novo* assembly. The reference guided assembly consists in the assembly of a genome when we already have a reference for the species of the individual sequenced. We expect the new genome to be very close to the reference and we are interested in the differences between the individuals. This type of assembly is much easier because we only have to find the differences between the two genome sequences and we can mimic the reference genome to order the reads. Reference guided assemblers as STAGE [14] and Chip-seq [15] consist in two main steps:

- Reads Alignment on the reference
- Consensus between mapped sequences

If this way of proceeding makes the assembly step easier and much less costly, it can seem unsatisfying for two reasons. First because of the biases that the method present. We make the prior hypothesis that the genome to assemble is very close to the reference.

This may mislead the assembly onto something too similar to the reference. Secondly the method is obviously not self-sufficient since a reference needs a prior reference to be constructed. We do not have access to many references genome as it can be seen on Figure 1.3 even if this number should increase dramatically in the next decades. For these reasons we will focus on *de novo* (without reference) assembly in this document.

Since we do not know the original position of the reads, we will try to position them with respect to each others. For this we mainly rely on the notion of "overlaps" between reads. We say that two sequences A and B overlap if a suffix of A is equal to a prefix of B. In some cases we might consider that A and B overlap if a suffix of A is similar to a prefix of B according to some distance. The main intuition being: "If a read A overlaps a read B then it is likely that A and B are consecutive in the genome" (if the overlap is large enough to be significant). Following this intuition, assembly consists in searching for reads that overlap each others. The larger the overlap, the more significant it is, as overlaps of few bases may be spurious. In Figure 1.4 the first and fifth reads share an overlap of 2 bases (GC) although the two reads do not come from the same location of the genome in gray.

Putting aside specific biases, we can depict a sequencing experiment as a uniform distribution of reads along the genome as a first approximation. Large overlaps then mean redundancy in the reads, as bases within an overlap will be present in reads sharing this overlap. In order to obtain large overlaps between sequences, we rely on a high redundancy in the sequencing dataset. The notion of coverage (or depth) of a genome by a sequencing is often used to quantify this redundancy. For a genome of 4 millions

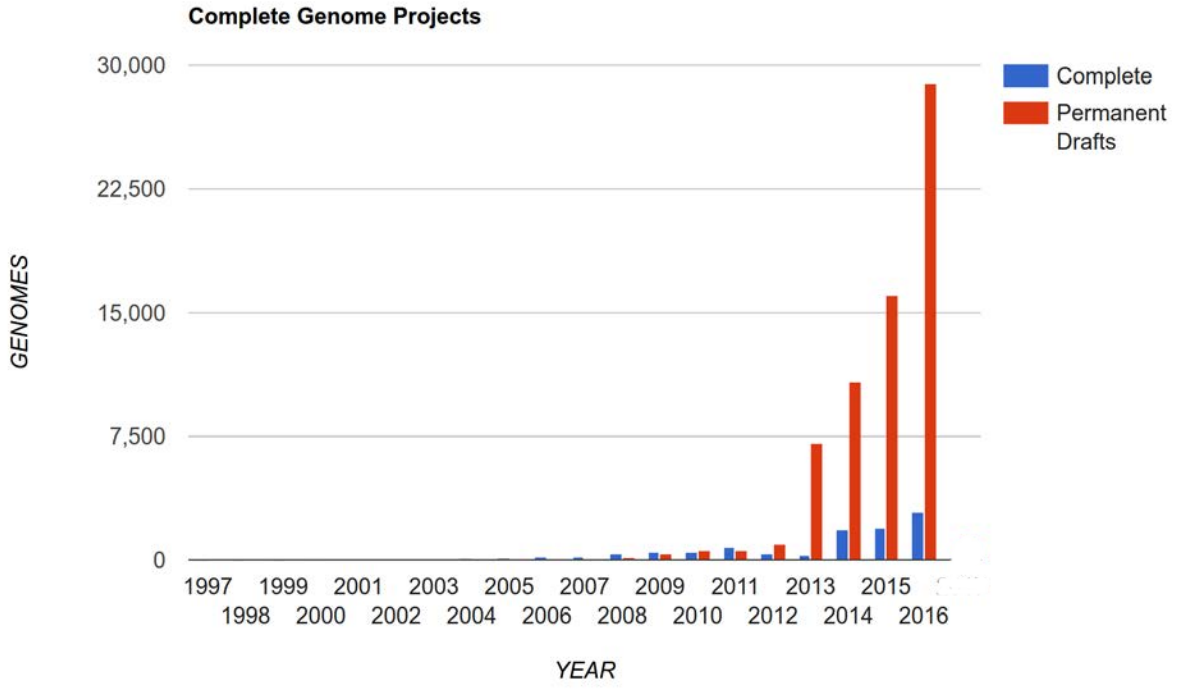

Figure 1.3: Number of finished and unfinished genomes in gold database (https://gold.jgi.doe.gov/statistics). Unfinished (or "draft") genomes may be represented as a set of widely fragmented sequences.

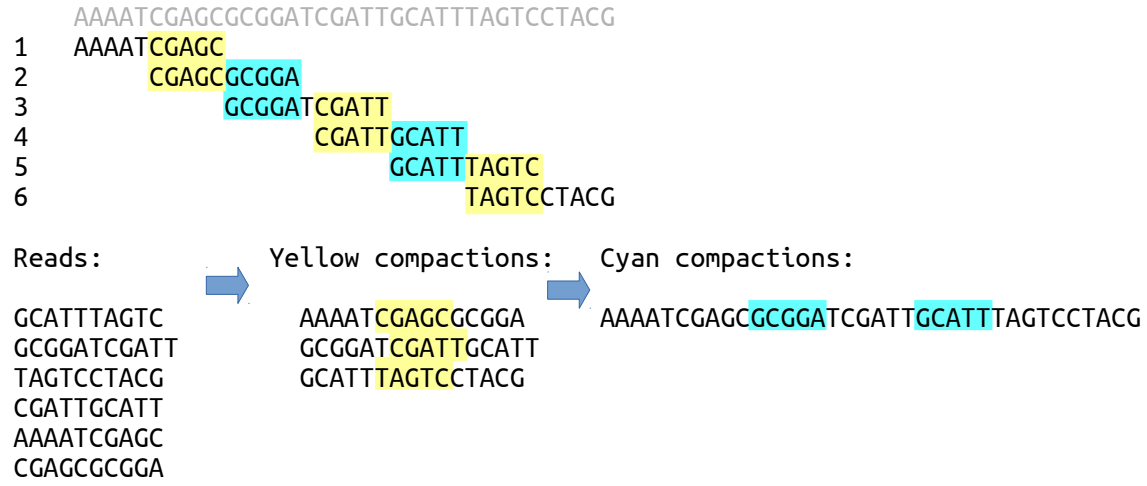

Figure 1.4: Intuition of how assembly is possible. The first set of sequences on the left are the reads to be assembled. Overlaps between those reads are computed before we can compact the reads according to the large overlaps found. Compaction between two reads AO and OB according to the overlap O is obtained by concatenating AO with the suffix B excluding O (AAAATCGAGC and CGAGCGCGGA become AAAATCGAGCGCGGA) . In this toy example, in a first step, we compact the yellow overlaps and obtain longer sequences. In the second step we compact the cyan overlaps and get the original sequenced genome. Double strand aspect of the DNA is not considered here.

Genome coverage	% Genome not sequenced	% Genome sequenced
0.25	78	22
0.5	61	39
0.75	47	53
1	37	63
2	14	86
5	0.6	99.4
10	0.0005	99.995

Table 1.1: Expected missing fraction of the genome according to the sequencing depth. Source: adapted from `http://www.genome.ou.edu/poisson_calc.html`.

bases, a sequencing of 4 millions reads of 100 base pairs will present a coverage of 100X because it contains 100 times more nucleotides than the genome. A high coverage is an important factor for at least 3 reasons. The higher the coverage:

- The larger the overlaps between reads are (on average)
- The less chance we have to miss regions of the genome because they are not sequenced (Table 1.1)
- The easier it is to deal with sequencing errors

With a high degree of redundancy we will be able via statistical methods to detect stochastic errors and remove them. Some assembly strategies try to correct reads before assembling them.

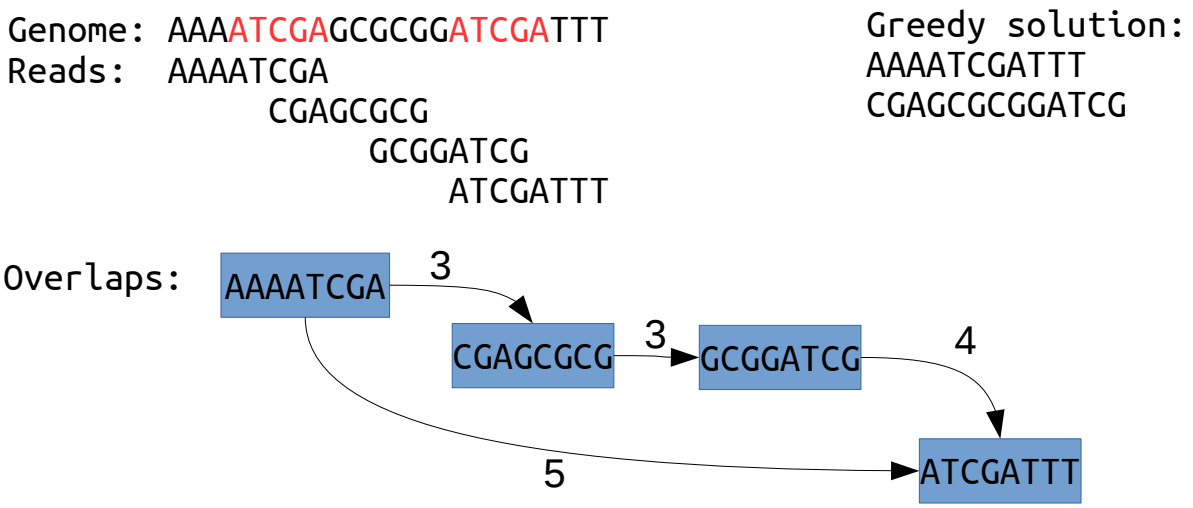

Figure 1.5: Example of greedy assembly. The overlaps between the first and last reads (AAAATCGA and ATCGATTT) is the largest and therefore compacted first in a greedy manner. This first compaction produces AAAATCGATTT. Then the largest overlap is between the two other reads (CGAGCGCG and GCGGATCG) that are compacted into CGAGCGCGGATCG. The assembler produced two misassembled sequences that are not the shortest common superstring.

We can distinguish three main families of assemblers: "Greedy", "Overlap Layout Consensus" and "de Bruijn graph " as detailed in the three following sections.

1.2.1 Greedy

This family of assemblers is the conceptually simpler. The idea is to find a shortest common superstring (SCS) of a set of sequences. Given a set of reads, a SCS is a string T of minimal size such that every read is a substring of T. Since finding the shortest common superstrings is an NP-complete problem [16], greedy assemblers, as their name suggests, apply greedy heuristics. The heuristic performs the compaction of the largest overlap if a read overlaps with several reads. The algorithm can be outlined by:

- Index reads
- Select two reads with the largest overlap
- Merge the two reads
- Repeat

The result is of course not guaranteed to be optimal as the greedy strategy may induce assembly errors, especially around repeated sequences (Figure 1.5). Popular Sanger assembler such as TIGR [17] or CAP3 [18] were greedy and broadly used because of their efficiency. This strategy was also reintroduced later to handle very short reads (around 25 to 50 bases) and implemented in assemblers like Ssake [19] and Vcake [20]. Those tools produce acceptable results on simple genomes. On more redundant genomes, such approaches produce too much assembly errors and other techniques are now favored. We can also criticize the model of the shortest common superstring, as in the presence

Figure 1.6: Toy example of the shortest common superstrings of a sequencing dataset. In this example, the genome contains a repeat in light and dark red. When the shortest common superstrings are computed we observe that none of the two SCS match with the genome because they are both shorter. In this example the SCS computation do not lead to the genome reconstruction.

of repeated sequences, multiple SCS can exist and genome may not be one of the SCS (Figure 1.6).

1.2.2 Overlap Layout Consensus

The overlap layout consensus paradigm core notion is the overlap graph. This framework is the most general among the three paradigms as we can argue that all assemblers use an overlap graph implicitly. The objective is to know how all reads can be positioned in relation to each others, to represent those connections in a graph and to consider all overlaps (not only maximal ones) to produce a solution. We know how reads can be ordered by knowing how they overlap. The overlap graph is a graph where reads are nodes, connected if they overlap significantly (Figure 1.7). The algorithm can be outlined by:

- Overlap: calculate pairwise overlaps between reads
- Layout: look for a parsimonious solution (as a generalized Hamiltonian path visiting each node at least once while minimizing the total string length)
- Consensus: merging reads, using redundancy to correct sequencing errors

The first OLC assembler was Celera [21] and was designed to handle Sanger sequences. Celera uses a BLAST-like [22] approach to compare each read to the others and to find significant overlaps. Then it compacts the overlaps presenting no ambiguity (Figure 1.8) and tries to apply heuristics on the complex cases involving repeats. The final sequences

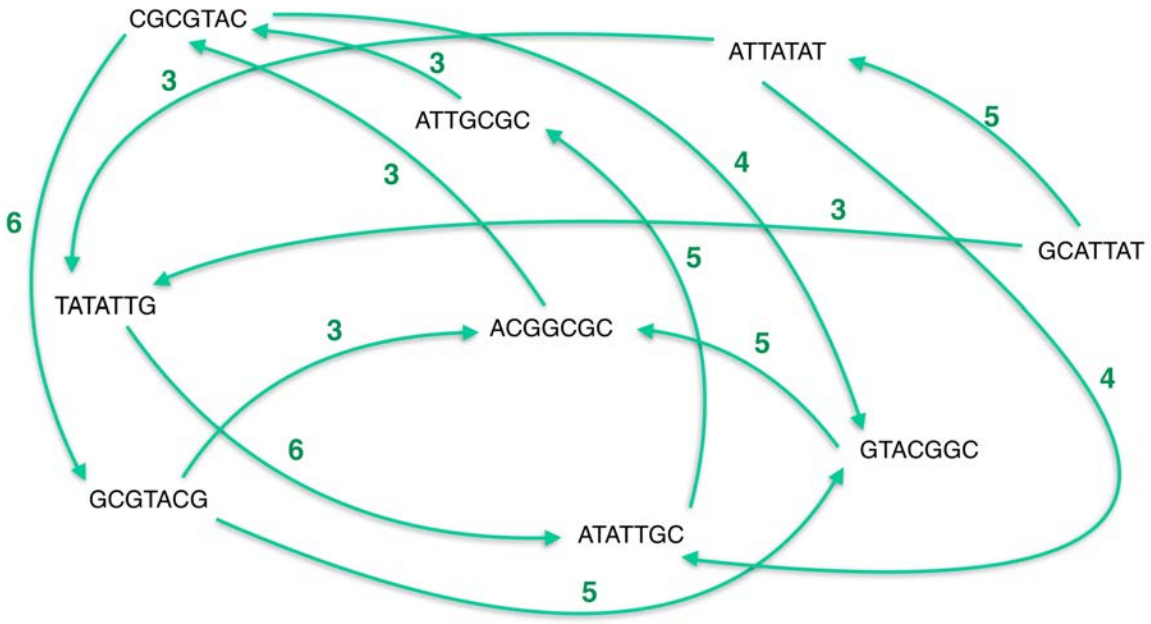

Figure 1.7: Overlap graph toy example where only overlaps of size 3 or more are considered.

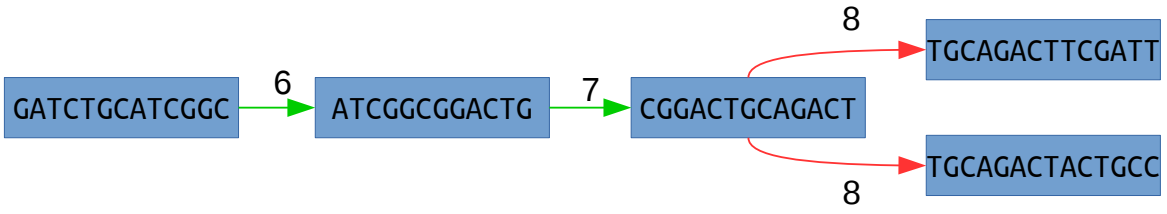

Figure 1.8: Example of non ambiguous compactions in green and unsafe compactions in red. The green compactions are the only possible choices, thus there is no ambiguity on which compaction should be performed. But the third read could be compacted with two other reads, the two compactions are indistinguishable. Choosing one compaction over the other could lead to assembly error.

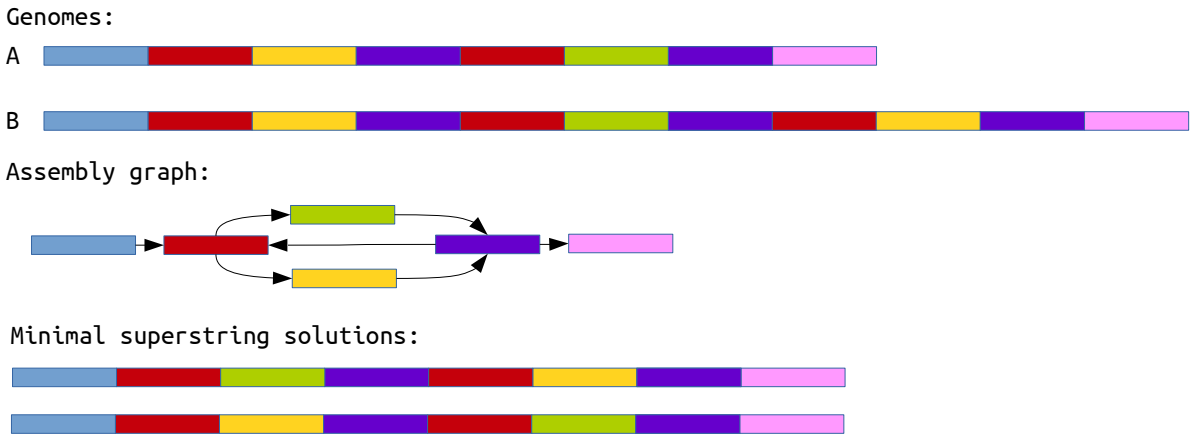

Figure 1.9: How repeated sequences can result in unsolvable graph. Two genomes A and B are sequenced, the reads covering their different regions are compacted into a simplified assembly graph of unambiguous sequences represented by blocks. Two repeats are present in red an purple. Several remarks can be made. First, the two genome A and B share the same simplified assembly graph structure despite being different. Secondly, several minimal solution may exist for a given assembly graph. Thirdly, because of repeats, the genome B will not be a minimal solution of the given assembly graph.

are produced via a consensus to remove most sequencing errors. For comparison, in the toy example of Figure 1.5, the OLC approach would achieve to solve the assembly by considering even non maximal overlaps and to produce the correct path, composed of strictly less nucleotides than the greedy solution.

The fact is that, with either paradigm, a perfect assembly is in most cases impossible to obtain. Sometimes the information available is not sufficient to make sound choices. In those cases the parsimonious strategy of no choosing between two indistinguishable possibilities is applied (Figure 1.8). This results into fragmented assemblies constituted of consensus sequences that are supposed to be genome substrings. We call those sequences "contigs" for contiguous consensus sequence [23]. In the example of the Figure 1.9 an assembly graph is created from the reads information. The graph can be simplified by compacting reads that overlap unambiguously into contigs. Assembly can become complex in multiple ways. First, different assemblies can be proposed from this graph even considering only minimal solutions. The green and yellow contigs are interchangeable in the two minimal solutions of Figure 1.9. Secondly, different genomes can share very similar assembly graphs. In Figure 1.9, both genomes A and B would be represented by the same simplified assembly graph. Thirdly, sometimes the solution is not a minimal substring: the genome B is not generated as minimal solution of the assembly graph. The most parsimonious way is therefore to output the proposed contigs represented by the colored blocks. To give orders of magnitude of the fragmentation of a genome into contigs we can look at published assemblies. A *E. coli* genome has almost a hundred contigs `https://www.ncbi.nlm.nih.gov/assembly/GCF_002099405.1`, a *C. elegans* genome count more than 5,000 contigs `https://www.ncbi.nlm.nih.gov/assembly/GCA_000939815.1/`.

But even by applying parsimonious rules, assemblers may make mistakes and produce assembly errors (misassemblies). Some tools, such as QUAST [24], are designed

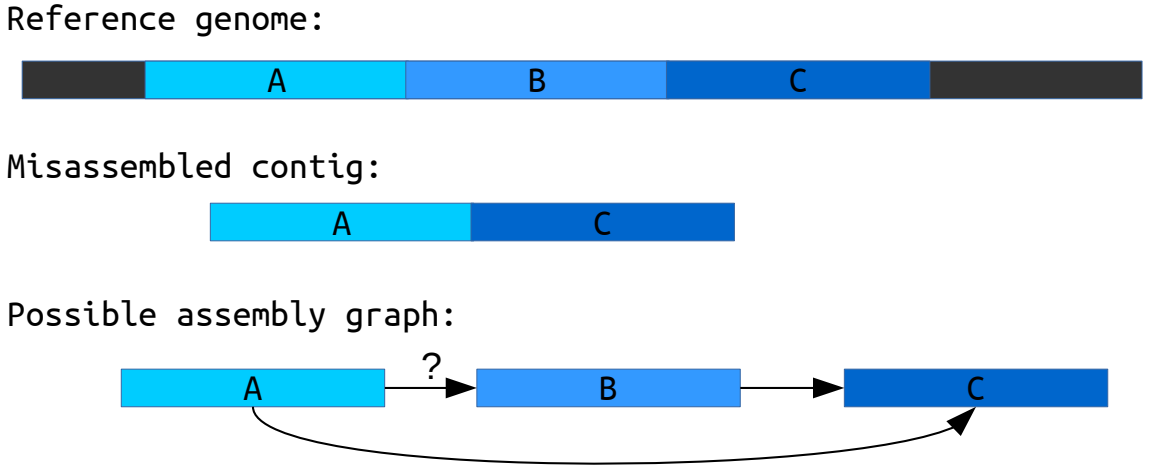

Figure 1.10: A classic error of assembly: the relocation. In the example the assembler produced a contig AC where the genome sequence is ABC. This error may be explained by different scenario, like a missing edge between A and B or a wrong choice of the assembler.

to evaluate the quality of an assembly by mapping the contigs on a reference genome and providing a classification and quantification of the different misassembly types. The classic misassembly is the relocation, when the assembler merges two sequences that are not consecutive in the genome into a contig. An example of relocation is shown on Figure 1.10: the assembler outputs the contig AC where the actual genome sequence is ABC. If the size of B is below a threshold (1 kilobase by default for QUAST) it is considered as a minor mistake as the contigs will still be mapped in their entirety on the reference genome and B will be considered as a deletion error. If B is larger this may be problematic as it induces a chimeric genome structure.

QUAST detects two other types of error (Figure 1.11).

- The translocation, where two (or more) parts of a contig come from different chromosomes
- The inversion, when a part of the contig is the reverse complement of the actual genome sequence

Such errors may appear for various reasons and may be due to the data or the assembler strategy. The connection AB may have been missed, leading the assembler to produce AC or a heuristic choice may have chosen the path AC over the path AB.

Users want assemblers both to produce long contigs and as few misassemblies as possible. The overlap graph is able to produce less errors than the greedy approaches by considering a more global information. But it may lead to very heavy structures with huge numbers of nodes and edges. In order to cope with this problem, a new improved model, the String graph [25], has been proposed. The essential conceptual difference between overlap graphs and string graphs is the transitive reduction of edges. When A overlaps B and C and B overlaps C, the edge A to C is removed because it can be "deduced" by transition from A to B and B to C (Figure 1.12). Later, [26] and [27] proposed further computational improvements to compute and represent a string graph.

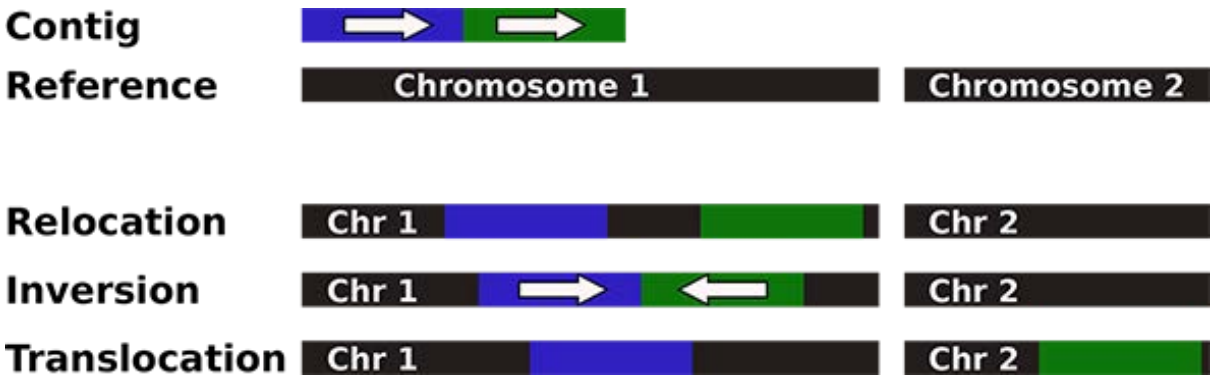

Figure 1.11: Assembly error types defined by QUAST. From `http://quast.bioinf.spbau.ru/manual.html`.

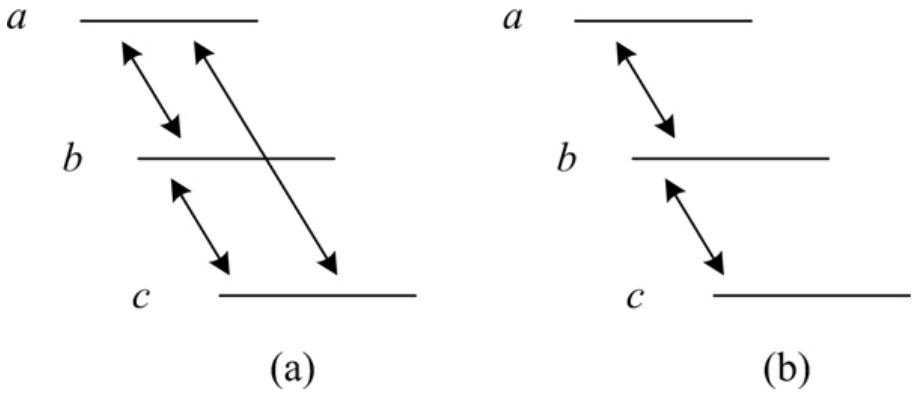

Figure 1.12: Transitive reduction of edges, the main difference between overlap graph and string graph. In the left part the connection between a and c can be deduced by the connections ab and bc and is removed in the right part.

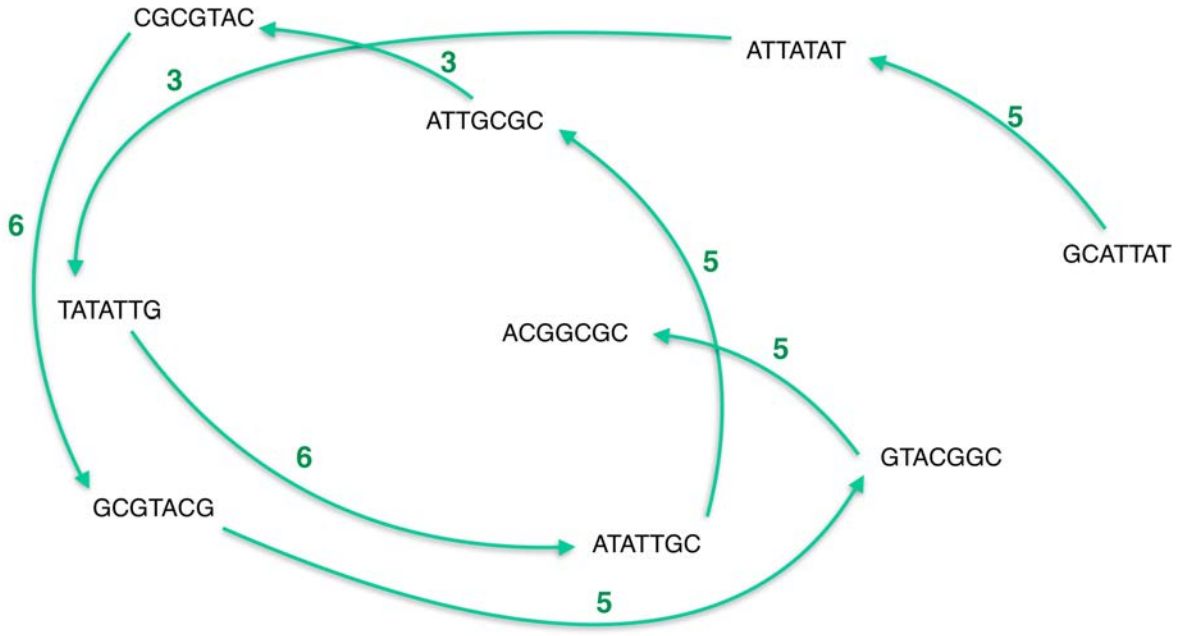

Figure 1.13: The previous overlap graph (Figure 1.7) toy example made into a string graph by removing transitively-inferrible edges.

This paradigm was used a lot with long Sanger sequences and for relatively small genomes. Because of the cost of the pairwise overlaps computation, the OLC is too time consuming on high number of short reads from NGS. Thus, other solutions had to be found to be able to deal with the amount of reads to assemble large genomes.

1.2.3 De Bruijn graphs

De Bruijn graph usage The de Bruijn graph is a directed graph representing overlaps between sequences of symbols, named after Nicolass Govert de Bruijn [28]. Given an alphabet σ of m symbols, a k dimensional de Bruijn graph has the following properties.

- m^k vertices produced by all words of length k from the alphabet σ
- Two vertices A and B are connected by an edge from A to B if and only if the $k-1$ suffix of A is equal to the $k-1$ prefix of B.

This graph has interesting properties and several applications in networking [29], hashing [30] and bioinformatics for genome assembly. Even if the graph used for assembly is called a de Bruijn graph, it is not exactly a de Bruijn graph as defined above.

The first application of the de Bruijn graph in genome assembly was introduced into the EULER assembler [31] in order to tackle assembly complexity. The idea was to consider a partial de Bruijn graph on the alphabet (A,C,T,G) constructed only with the vertices whose words of length k, called kmers, appeared in the sequencing data. The intuition of this approach is the following (Figure 1.14):

- A read is represented as a path in the graph
- Reads that overlap with more than k nucleotides will share some kmers
- Extracting paths of such graph will produce assembled reads

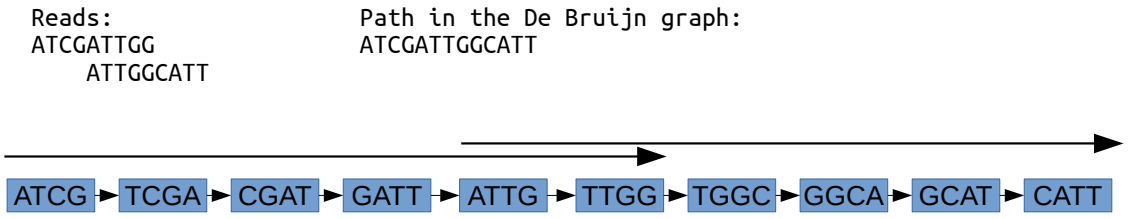

Figure 1.14: Intuition of how the de Bruijn graph is able to merge overlapping reads. In this figure, the two read share an overlap of 5 bases. In the de Bruijn graph they share 2 kmer and are part of a path of the graph whose sequence is the concatenation of the two reads.

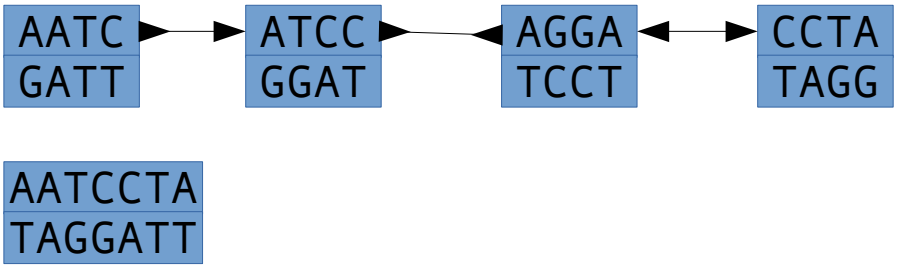

Figure 1.15: Classic representation of canonical kmers and their different kinds of overlaps. For each kmer, the (lexicographically) smaller kmer between the sequence and its reverse complement is chosen as representative and is called the canonical kmer. Since a kmer and its reverse complement are indistinguishable, several types of k-1 overlap have to be considered in a de Bruijn graph. The first one is the forward forward (FF), that comes from canonical to canonical. The second one is the forward reverse(FR), from the canonical to the reverse. The third one is the reverse forward (RF) from the reverse to the canonical.

Another technical point comes from the fact that the sequencing data is not stranded. As previously mentioned, since a read can be extracted from a strand or another, a sequence and its reverse-complement should be considered as the same object. In order to handle this, the principle of canonical kmer is used. A kmer w is called "canonical" if and only if it is smaller lexicographically than its reverse complement v. The kmers that are canonical are just inserted in the graph when the reverse complement of non canonical kmers are inserted instead of them. This modification leads to a change in the definition of how two kmers overlap and therefore are connected in the graph. In this graph four connections are possible (Figure 1.15):

- Suffix $\rightarrow$ prefix
- Suffix $\rightarrow$ reverse prefix
- Prefix $\rightarrow$ suffix
- Prefix $\rightarrow$ reverse suffix

This is rather a technical problem than an algorithmic one. Practical and theoretical results show that the difficulty of assembly is not due to this stranding problem [32]. For the sake of clarity we will ignore this property in most cases.

De Bruijn graph and overlap graph The de Bruijn graph theoretically achieves the same tasks than the overlap graph, while being conceptually simpler and much more efficient for the three reasons detailed in the following:

- No alignment
- Abstracted coverage
- No consensus

The Figure 1.14 shows how the de Bruijn graph finds (exact) overlaps of length superior to $k-1$ between two reads. The de Bruijn graph does not explicitly compact reads together. However, selecting long paths from the de Bruijn graph is very similar to compacting overlapping reads in the OLC.

The de Bruijn graph became widely used when the shorts reads from NGS appeared, as it was better suited than the OLC to handle this kind of sequencing data. The OLC approach did not scale well on the high number of sequences generated by NGS. The use of the de Bruijn graph is very interesting for short read assembly for its ability to deal with the high redundancy of such sequencing in a very efficient way. Indeed a kmer presents dozens of time in the sequencing dataset appears only once in the graph. This makes the de Bruijn graph structure not very sensible to the high coverage, unlike the OLC. The de Bruijn graph was first proposed as an alternative structure [31] because it was less sensible to repeats. Repeats that were problematic in the OLC, creating very complex and edges heavy zones, are collapsed in the de Bruijn graph.

Redundancy The high redundancy in the sequencing data can also be used to efficiently filter the sequencing errors. The first highly used de Bruijn graph short reads assembler was Velvet [33]. It introduced core notions of de Bruijn graph assembly, as the idea of error filtering based on kmer abundances. With a high coverage, we expect a high abundance for most kmers. A kmer with very low amount of occurrences is therefore very likely not a genomic kmer (a kmer present in the genome) but rather an erroneous one (not present in the genome and coming from a sequencing error). By admitting in the de Bruijn graph only kmers with a coverage above a threshold, we can get rid of most erroneous kmers almost without losing genomic kmers, but the ones that has an unexpectedly low abundance. Kmers whose abundance are over the abundance threshold (or solidity threshold) are called "solid". In Figure 1.16 we can see that sequencing errors generate a huge number of low abundance kmers.

De Bruijn graph patterns De Bruijn graph assembly basically consists in graph simplification by applying heuristics on knowns patterns (Figure 1.17). They rely on path exploration, applying different strategies to handle recurring motifs. After graph simplification, they output the long simple paths as their contigs (Figure 1.18).

The simplest pattern is the tip (or dead end), a short path in the graph that is not extensible because its last kmer has no successor. If the tip is short, shorter than the read length for example, then it is very likely to be due to a sequencing error. But if it is large it could just be the beginning or an end of a chromosome. A basic assembly step is to recognize those short tips and remove them from the graph to simplify it and to allow longer contigs.

Another frequent pattern is the bubble. A bubble arises when several paths start from a kmer and end in another kmer. A sequencing error can create a bubble if the

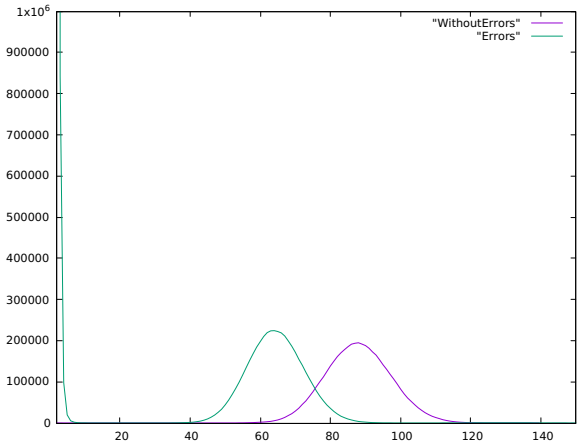

Figure 1.16: Comparison of kmer spectra with and without sequencing errors. The two kmer spectra are computed from simulated reads with 1% and 0% errors respectively from *E. coli* with a coverage of 100X. We observe two similar curves but the green one presents a huge quantity of low abundance kmers due to the presence of sequencing errors.

sequencing error is positioned k nucleotides apart from the end and the start of a read.

Another source of bubbles can be heterozygosity. For organisms with (at least) a pair of homologous chromosomes, heterozygosity defines the scenario of owning two different versions of a sequence (called alleles) on each chromosome. A well known example is blood type:, for instance an AB individual is heterozygous. A diploid (polyploid) genome is a genome containing two (multiple) complete sets of chromosomes. Within a diploid (polyploid) genome, heterozygosity can happen at various positions. When a diploid (polyploid) genome is sequenced, two (multiple) close sequences are simultaneously sequenced and the original allele of each read is unknown. When the two sequences are almost identical with some minor differences, this creates bubbles in the de Bruijn graph.

A last important source of bubbles are the quasi-repeats. When two almost identical sequences appear in the genome, it can create the X pattern specific of the repeats (Figure 1.19), with bubbles inside the central repeated sequence (Figure 1.20).

Those patterns highly depend on the size of k. The k parameter, called the "order" of the de Bruijn graph, is a key factor in de Bruijn graph assembly. For a given dataset, distinct orders can lead to the construction of extremely dissimilar graphs.

Repeats The first point about the size of k is that the smallest k is, the more complex the graph will become. With a small k, the probability of a kmer (or a $k-1$mer) to be present at multiple positions in the genome is high. If this happens, then the two occurrences of the repeated kmer are collapsed in the graph and create a "X" structure (Figure 1.19). When such a motif is encountered, the assembler has no mean *a priori* to know how to continue and stop. A k too small may results into a profusion of such scheme and produce a fragmented assembly. In a similar way, a repeat of size $k-1$ can also create an edge between unrelated kmers called "spurious edge". In other words, the smaller the k value is, the less significant the connections between kmers are. To see how the size of k affects the graph complexity, we present in Table 1.2 the number of repeated kmers in the de Bruijn graph created from reference genomes according to the

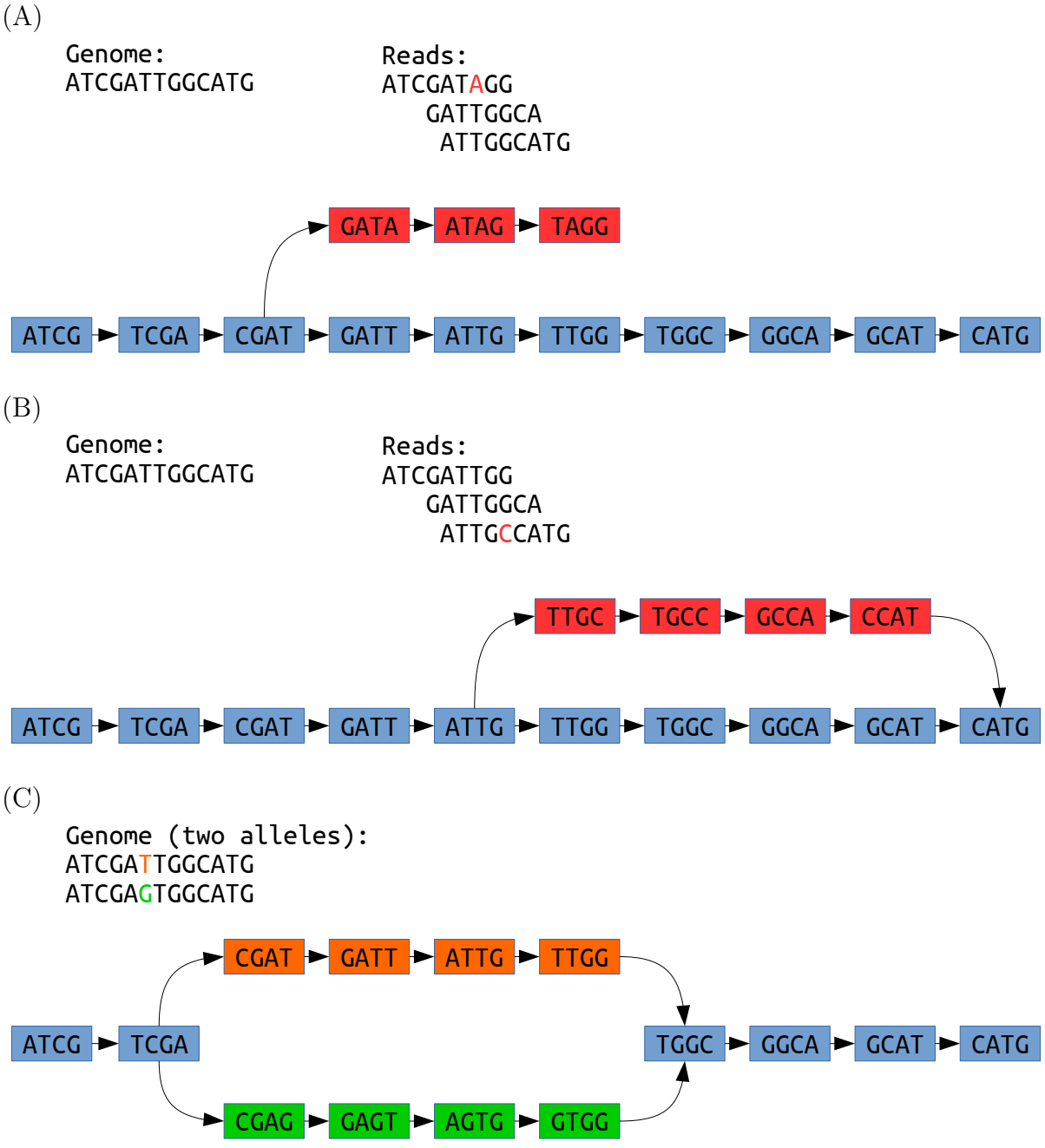

Figure 1.17: Patterns in a de Bruijn graph with $k = 4$. The first pattern (A) is a tip generated by a sequencing error at the end of a read. The second one (B) is a bubble generated by a sequencing error in the middle of a read. The third one (C) is a bubble generated by an actual variation in the sequenced genome.

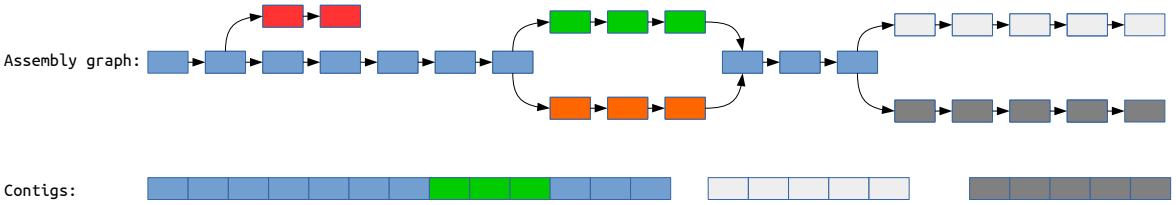

Figure 1.18: Example of contigs generation. In this toy example, the tip in red is removed, the bubble is crushed and the green path is chosen over the orange one. But the assembler is not able to choose between the two gray paths to extend its contig so it stopped. The two gray paths are output as contigs with the large blue one.

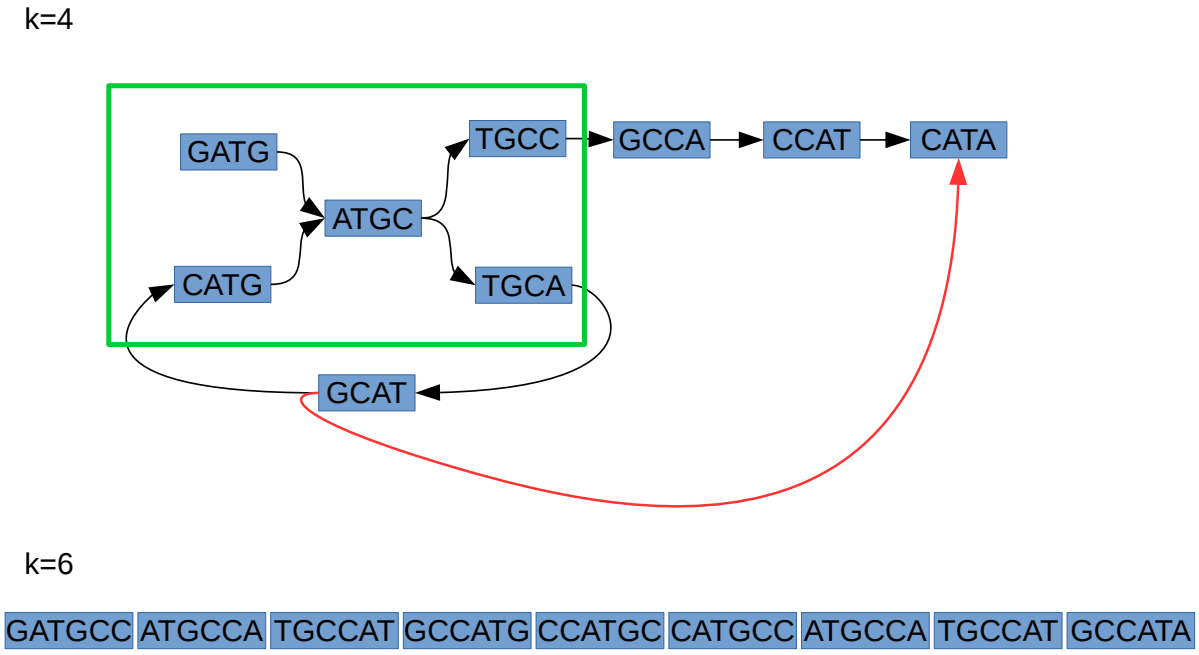

Figure 1.19: Example of spurious edge and X pattern due to repeats of size k and $k-1$. With k=4 we got a repeat of size 4 that creates the X pattern framed in green. A repeat of size 3 also create a spurious edge between GCAT and CATA. With k=6 the graph is linear since no repeat of size 5 exist.

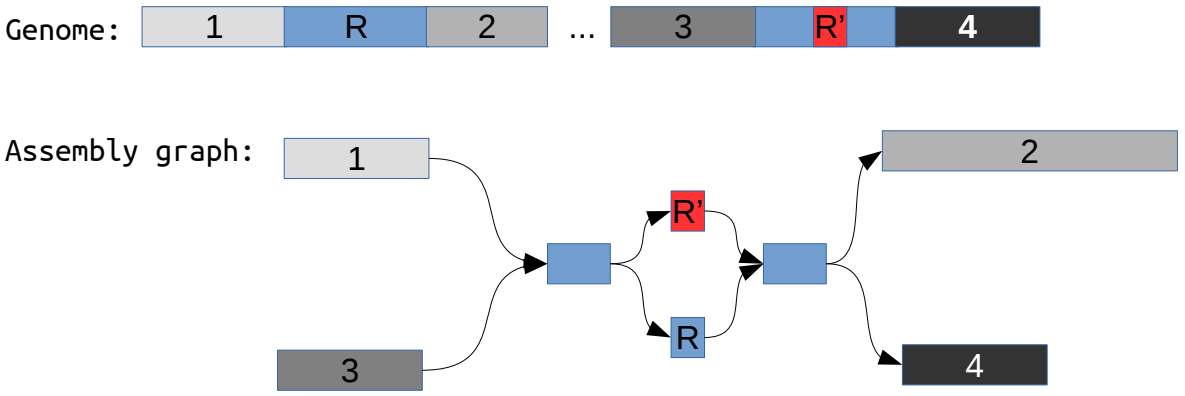

Figure 1.20: Example of quasi-repeats creating a bubble. The two regions R and R' are almost identical, but the small difference between the two occurrences create a bubble inside the X pattern.

Reference genome	kmer size	Repeated kmer
E. coli	11	976,822
E. coli	15	104,501
E. coli	21	33,745
E. coli	31	30,273
E. coli	61	25,575
E. coli	101	22,345
E. coli	201	17,985
E. coli	301	15,421
C. elegans	15	14,299,107
C. elegans	21	3,290,873
C. elegans	31	2,475,913
C. elegans	61	1,892,518
C. elegans	101	1,524,266
C. elegans	201	1,052,332
C. elegans	301	824,254
Human	15	250,505,977
Human	21	148,690,202
Human	31	122,846,758
Human	61	102,800,268
Human	101	92,294,300
Human	201	85,392,911
Human	301	81,190,289

Table 1.2: Number of kmers that appear multiple times in the reference genome according to the size of k.

size of k.

We observe that a low k value creates a de Bruijn graph with many repeated kmers that will create X patterns and therefore a complex graph. This can be explained by

Reference genome	kmer size	Percent unique kmer
E. coli	9	1.1
E. coli	11	33.2
E. coli	15	97.7
E. coli	21	99.3
C. elegans	11	0.6
C. elegans	15	76.4
C. elegans	21	96.4
C. elegans	31	97.4
Human	15	25
Human	21	93.4
Human	31	95.1

Table 1.3: Proportion of kmer that appear a single time in the reference genome according to the size of k.

the fact that small kmers are expected to appear multiple times even on a random word. There are 262,144 different 9mers (without accounting for reverse-complements) while the *E. coli* genome is a word of almost 5 millions nucleotides. We expect most 9mers to be present multiple times in the genome. The Table 1.3 confirms this fact. With $k = 11$ (4,194,304 11mers) most kmers are still repeated but with $k = 15$ and higher (1,073,741,824 15mers), most kmers are unique (present only one time in the genome). In practice we need to use a k such that 4^k is order of magnitude larger than the genome size in order to get a de Bruijn graph with mostly unique kmers. A higher value reduces the number of repeated kmer, with a decreasing efficiency because of the existence of large repeats in genomes. *C. elegans* reference genome presents almost a million repeats larger than 300 nucleotides.

Overlap detection The second point about kmer size is that a de Bruijn graph will only (implicitly) detect overlaps larger than $k-1$ between reads. A high k value imposes that reads have to share large overlaps, and a high redundancy may be necessary to do so. A too large k value will create holes in the de Bruijn graph (Figure 1.21).

Error removal The third point is about sequencing errors. A read of size L has $L-k+1$ kmers. Thus a high k value means less kmers per reads. For example a read of length 100 has 70 31mers or 40 61mers. With a higher k value, the abundance of all kmers is lower and it is more difficult to differentiate genomic from erroneous kmers, based on their abundance. Another problem is that sequencing errors "destroy" kmers (Figure 1.22). When $L \leq 2 * k - 1$ a sequencing error can make erroneous all kmers from a read . The Table 1.4 shows the effect of the errors and the kmer size on the presence/absence of the genomic kmers without any kmer filtering. 6,716 missed kmers can seem like a low rate, around 1/1000 of the genomic kmers. But each of these "holes" fragments the assembly as seen in Figure 1.21. We observe that reads with a reduced error rate (substitutions) allows for the usage of a higher kmer size with the same coverage.

Given those effects summarized in Table 1.5, the size of k should be chosen according

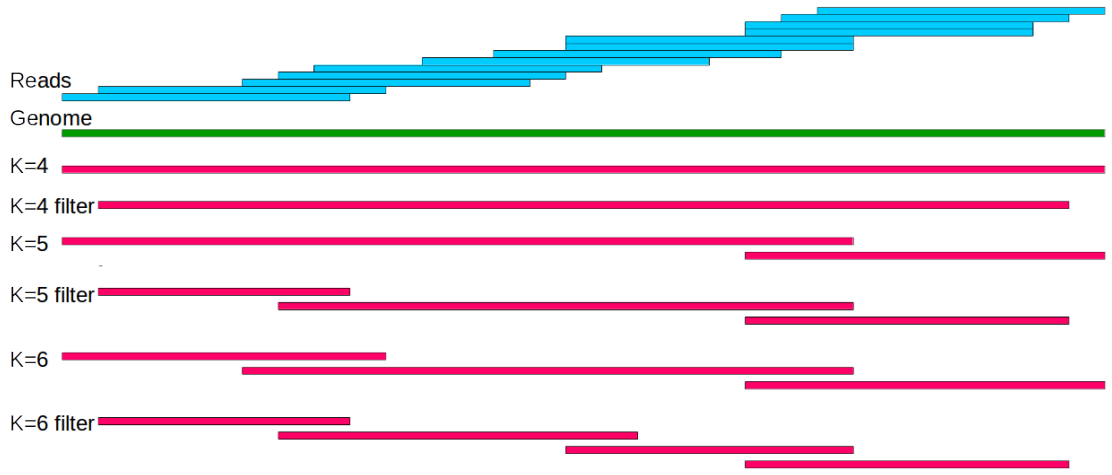

Figure 1.21: How the use of high values of k and solidity thresholds can fragment the de Bruijn graph. The red lines represent the paths of the de Bruijn graph according to the k value and the filter usage. The filter consists in removing the unique kmers. We can see that using a high k value can fragment the assembly because no large enough overlap exists. Unique kmers removal can also create holes when removing errorless kmers. Using a too large k size or a too high solidity threshold creates holes in the de Bruijn graph and results into a fragmented assembly.

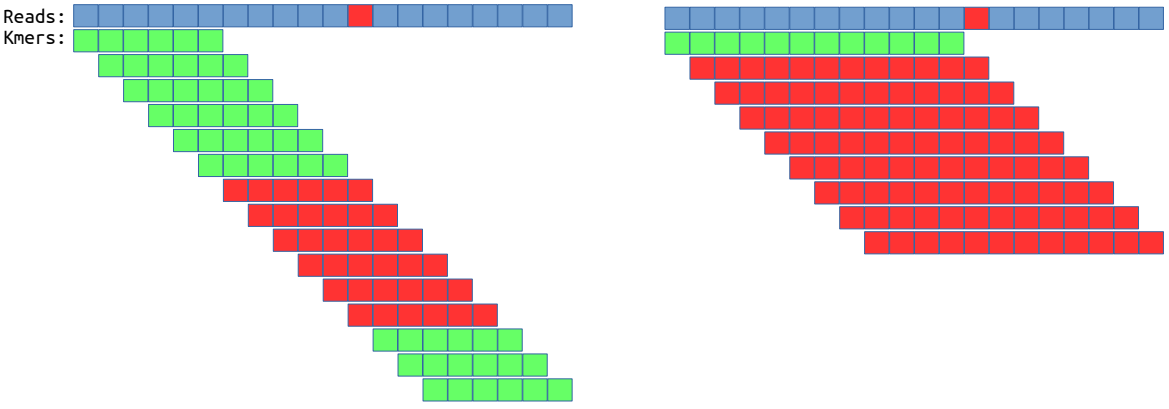

Figure 1.22: How sequencing errors can produce many erroneous kmers. Red positions are sequencing errors and kmer in red are therefore erroneous kmers while green kmers are genomic. In the left figure, $k = 8$ thus the read has 15 kmers. Because of the sequencing error, 6 of them are erroneous. In the left figure, $k = 12$ thus the read has only 9 kmers. Because of the sequencing error 8 of them are erroneous. If the sequencing error was one base to the left, all kmer would have been erroneous.

Error rate	kmer size	Missed kmers
1%	31	3
1%	61	6,716
1%	91	1,374,153
0.1%	31	2
0.1%	61	53
0.1%	91	293,675

Table 1.4: Number of genomic kmers not present in a set of 100 base pairs simulated reads with a coverage of 30X from *E. coli*, according to the error rate.

	Low k	High k
Benefits	Detect even small overlap between reads kmers are more covered (more kmer by reads)	Less repeats in the graph Less spurious edges
Drawbacks	More repeats appear in the graph More spurious edges	Only large overlap are detected kmers are less covered

Table 1.5: Effect of k parameter on a de Bruijn graph.

mainly to the coverage of the sequencing dataset and to its error rate. Some methods concentrate on automated selection of this crucial factor, as kmerGenie [34] in order to produce the best assembly possible given a read set.

Assembly quality Ideally we would want to use the highest possible kmer size in order to reduce the number of repeated kmers and spurious edges. To do so without having a graph abounding with holes, multiple solutions exist:

- Get a huge coverage
- Correct sequencing errors
- Use several kmers size

The first solution has no real drawback, but is not satisfying because of the supplementary expenditure. Even if sequencing costs are lowering, sequencings still represent important investments. In some cases the bottleneck may be the quantity of DNA material accessible, that would make impossible to get more sequencing.

The second solution, used by many assemblers [33, 35, 36, 37, 38], is to use a preprocessing to correct sequencing errors. Those techniques use the redundancy of the read set to correct errors based on statistical methods. It presents the interest to use reads with less errors and therefore containing longer errorless sequences. Read correction step can be criticized as it may modify correct bases or may correct errors incorrectly.

The third solution, used in state of the art assembly approaches (called multi-k approaches), is to make use of multiple kmer sizes. The idea of those approaches is to get the graph connectivity from the low kmer sizes, and the low amount of repeats from the high kmer sizes. The precise method may vary among tools but it generally follows the

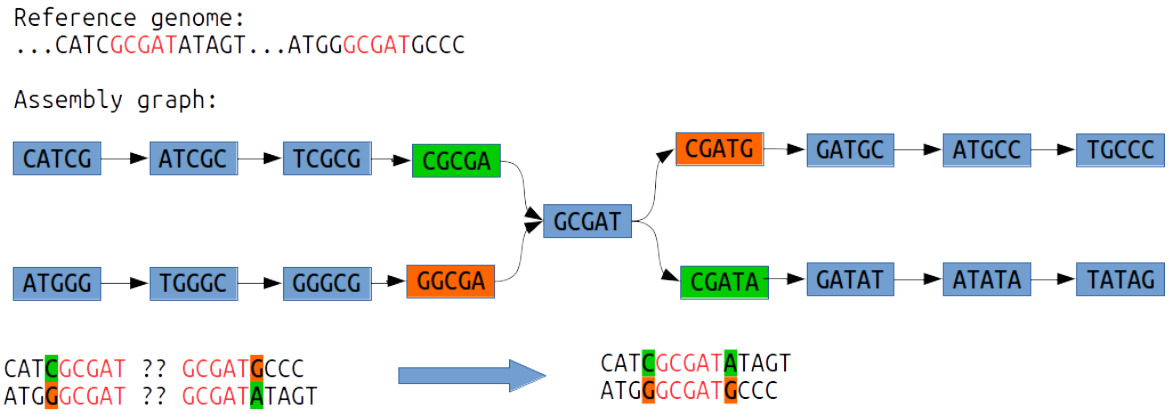

Figure 1.23: Solving a repeat consists in the determination of the possible contexts around the repeat and therefore limits the fragmentation. To solve the presented toy repeat, the assembler must link the green parts and the orange parts together.

pattern described by IDBA [39] in order to construct the best possible de Bruijn graph:

1. Start with a low k
2. Create a de Bruijn graph from the reads with k
3. Produce contigs using this graph
4. Raise the value of k
5. Create a de Bruijn graph from the reads and the contigs with k
6. Repeat from step 3 until $k \leq k_{max}$

This method allows the use of large k values even if the coverage is not sufficient everywhere. The assembly step with a low k value detects the possible compactions between reads that share low overlaps. The large kmer size allows to "solve" some repeats as presented in Figures 1.23 and 1.19. Those approaches are still not perfect since complex or poorly covered regions can be absent from the contigs. Another problem of those techniques is that they may be computationally challenging since they perform several assembly steps, in a sequential way (time consuming) or simultaneously (memory consuming).

A plethora of de Bruijn graph based assemblers have been developed, ABYSS [40], SOAPdenovo [41], IDBA [39], ALLPATHSLG [42], gossamer [43], minia [44], SPADES [36]. Each brings to the table extremely different properties and trade-off between resources needed, efficiency and assembly quality.

But even the best combinations of those strategies are not able to use the full read length. Despite being conceptually simpler than OLC, the de Bruijn graph is inferior in terms of information usage because $k \leq read\ size$. Even if we use a very large k value, we may encounter repeats with $k < repeats\ size < read\ size$ that could be solved by a string graph but not by the de Bruijn graph. The multi-k de Bruijn graphs tend to

come close to the string graph (various overlap sizes detection, usage of a large part of the reads) but also tend to lose their performances advantages.

"Genome assembly" core messages:

- Assemblers goal is to order the reads and concatenate them into larger sequences while removing the sequencing errors
- In most cases, the assembler is not able to completely recover the genome and will output "contigs" that are sequences supposed to be substring of the genome larger than the reads
- Greedy approaches are efficient but are not suited to large or complex genomes as they handle repeats very poorly, resulting into erroneous assembly.
- Overlap graph is a framework where the overlaps between the reads are computed and put in a graph where reads are nodes, connected if they overlap.
- The de Bruijn graph is a more restricted framework where the words of length k from the reads are nodes, connected if they share an overlap of $k-1$.
- OLC and de Bruijn graph approaches select paths from their graphs using heuristics in order to produce contigs based on the graph structure
- The overlap graph approaches relies on heavy data structures while the de Bruijn graph, conceptually simpler, scales better on large datasets

1.3 Assembly hardness

Assemblers are supposed to produce the largest possible contigs while making as few assembly errors as possible from the available read information. Here we describe the challenges that genome assembly may present and the limitations of existing approaches.

1.3.1 Repeats

As we have seen, one of the problems of assembly is the repeated sequences. In fact repeats may be the fundamental problem in genome assembly. An analysis of the complexity of the assembly problem according to the type of data available has been made in [45]. The principal conclusion of the paper is that repeats larger than reads are impossible to solve. In fact, for most genomes, the perfect assembly is not achievable using only NGS reads. It may seem odd that such large repeats exist. The fact is that genomes are absolutely not random sequences. When we compare Table 1.6 with Table 1.2, we see that there is no large repeated kmers in random sequences whereas *E. coli* genome has thousands of repeated 301mers. The conclusion is that NGS reads sequences are not enough to produce a complete assembly. [46] has shown the effort necessary to finish short reads assembly. Therefore longer range information is required in order to solve the large repeats and produce more continuous assembly.

Reference genome	kmer size	Repeated kmer
Random Genome (*E. coli*)	15	84,148
Random Genome (*E. coli*)	21	28
Random Genome (*E. coli*)	31	0
Random Genome (*C. elegans*)	15	34,050,959
Random Genome (*C. elegans*)	21	17,953
Random Genome (*C. elegans*)	31	0

Table 1.6: Number of repeats of size k in a random genome of the same size than *E. coli* or *C. elegans*.

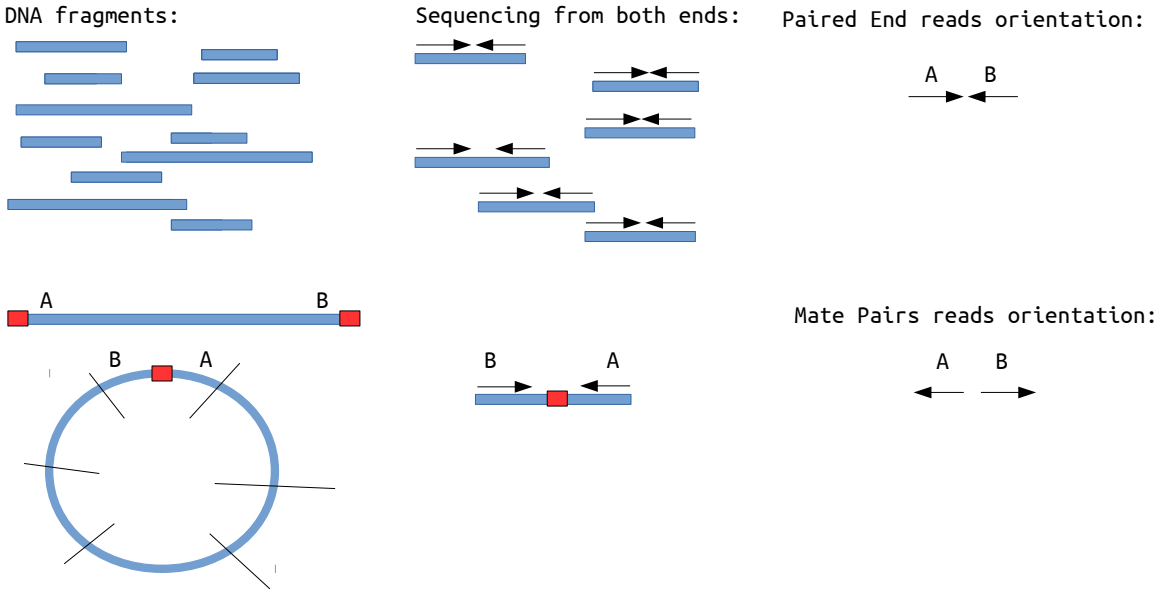

Figure 1.24: Differences between Paired End sequencing and Mate Pairs sequencing. In Paired End, small DNA fragment are selected and sequenced from both end. For Mate Pairs, longer fragment are selected, both ends are marked and connected into a circular sequence. The sequence is then cut and the marked part is sequenced from both and generates an opposite sequencing orientation.

1.3.2 Scaffolding

As denoted before, most assemblers use the reads sequences to produce contigs, that are a set of fragments of the genome. In order to improve such genome drafts, other informations and techniques can be used. It is possible via sequencing techniques to obtain short reads that are associated by pairs coming from related positions of the genome. There are two main sequencing techniques called "Paired End" and "Mate Pairs" sequencing (Figure 1.24). In both cases, we get a pair a short reads with the same characteristics than short reads described previously. The additional information is that we have an estimation of the distance covered by the read pair, called the fragment size, since we know that both come from the same DNA fragment. For example, a paired end sequencing of 2*250 base pairs with an fragment size of 800 will produce pairs of reads of 250 bases pair reads spaced by 300 nucleotides in the genome (on average). The paired end sequencing can handle fragment size up to 2,000 bases while mate pair can produce fragment size from 2 to 20 kilo-bases. Another difference is that in Paired End sequencing, the first fragment is read forward and the second is reversed while in Mate Pairs this is the opposite, as shown in Figure 1.24. This can be explained by the difference between the two protocols. In paired end sequencing, fragments of the desired size are selected and sequenced from both ends. In mate pairs sequencing long fragments end are marked, circularized, fragmented, and sequences with marker are sequenced from both ends.

The standard way to use those linking information is to align paired reads on the contigs and use the pair of reads that map on different contigs to order and orient them with respect to each other. This task is frequently called scaffolding, because contigs are arranged together into "scaffolds", based on estimations of the distance between them (Figure 1.25). A complete assembly workflow is summarized in Figure 1.26.

[47] formally defined the problem and showed its NP-completeness and therefore its potential intractability. Most scaffolders try to optimize the number of satisfied links in order to produce scaffolds. The main problem of scaffolds is that they may contain "holes". Scaffolds are basically ordered contigs. When the different contigs used are not overlapping, the scaffolder usually estimates the distance between them and fills the "holes" with 'N' characters to inform that the sequences here are unknown. The scaffolders may be able to order large contigs, but can fail to find the contigs separating them. This may be the case if those contigs are too small or simply if they were not output by the contig generation step. Since the scaffold approach is based on mapping the reads on the contigs, very short contigs (shorter than reads) cannot be considered this way because of multiple mapping problems (when a read maps on several contigs).

Recent scaffolders like SSPACE [48] try to extend contigs using the pairs information earlier to connect them in order to reduce the potential gaps in the later phase. Some standalone tools called gap-giller, as MindTheGap [49], are specifically designed to fill such holes. Other kinds of information start to be used for scaffolding such as long reads from third generation sequencing [50] and 3C [51] a sequencing technology liking reads coming from the same chromosome.

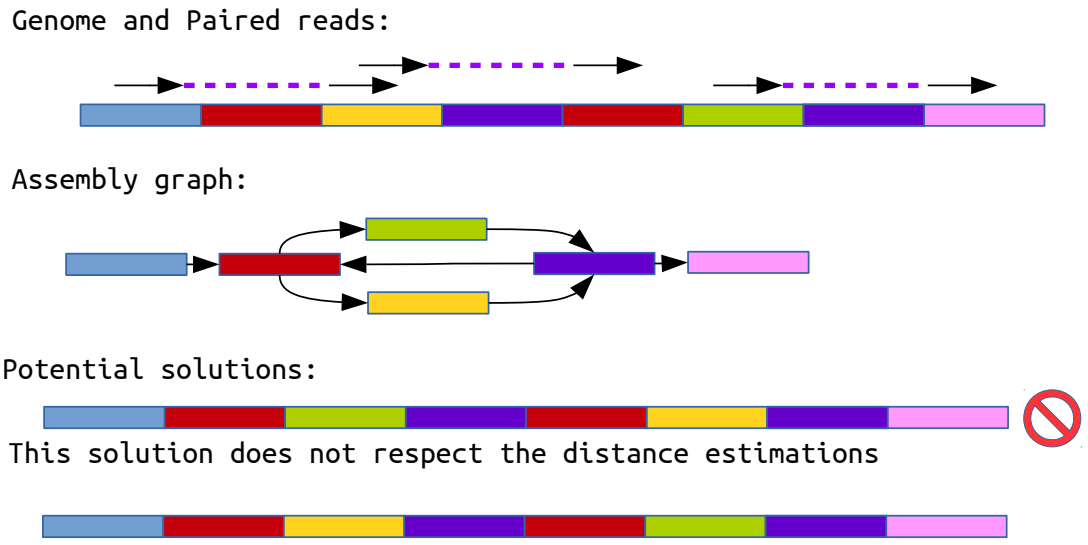

Figure 1.25: How the paired reads information can be used to solve some complex scenario. Paired reads show that blue and yellow contigs are separated by one contig, as yellow and red and green and pink. Using this information, we can eliminate solutions where those distances are not correct. The first solution does not respect any links (Blue and yellow are too far apart as green and pink and yellow and read are too close) and is therefore rejected. In this example the solution respecting the link is equal to the genome.

1.3.3 Multiple genomes

The assembly problem is defined as a sequence reconstruction. In many cases we are interested in assembling multiple genomes at once. We briefly introduced heterozygosity and highlighted that for many species, as human, two sets of chromosomes, or more, are sequenced. We will refer to a complete set of chromosomes coming from an individual as an haplotype. A human has therefore two haplotypes because it owns two versions of each chromosome (but the sexual ones). Intuitively, we are sequencing two similar genomes at once. We can define the heterozygosity rate by the ratio of the edit distance between the two alleles over the size of the region. Regions identical across haplotypes will be called homozygous while regions with variations will be called heterozygous. Since for many model species such as the human the heterozygosity rate is very low (around 0.1% for the human), most assemblers do not try to assemble both sequences and rather ignore the heterozygosity information. They usually produce a sequence presenting a mix of the different haplotypes as it is shown in Figure 1.27. In an assembly graph such variants may appear as bubbles and most assemblers will select a path through the bubble without trying to conserve haplotype information. Some tools try to assemble separately the different haplotypes when possible as Platanus [52]. In such cases, all homozygous regions can be considered as repeats as they appear identically in two different chromosomes. But other scenarii can force users to assemble multiple genomes at once.

Some species cannot be grown and sequenced in wet labs, therefore their whole environment (sea water for instance) is picked and sequenced, then we must assemble the

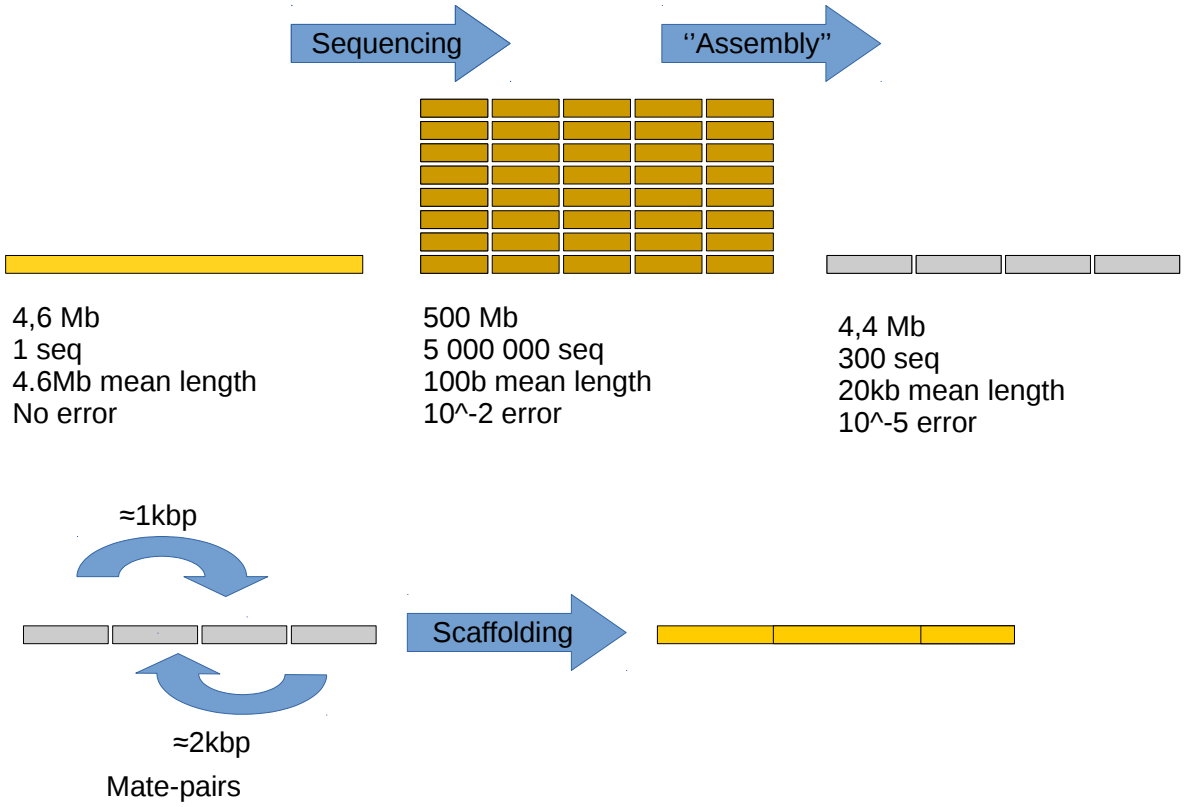

Figure 1.26: Summary of assembly steps. The global picture of an assembly of *E. coli* from simulated reads with SPAdes. Some refer to assembly for contigs generation and scaffolding steps while other call the contigs generation step as assembly and separate it from the scaffolding.

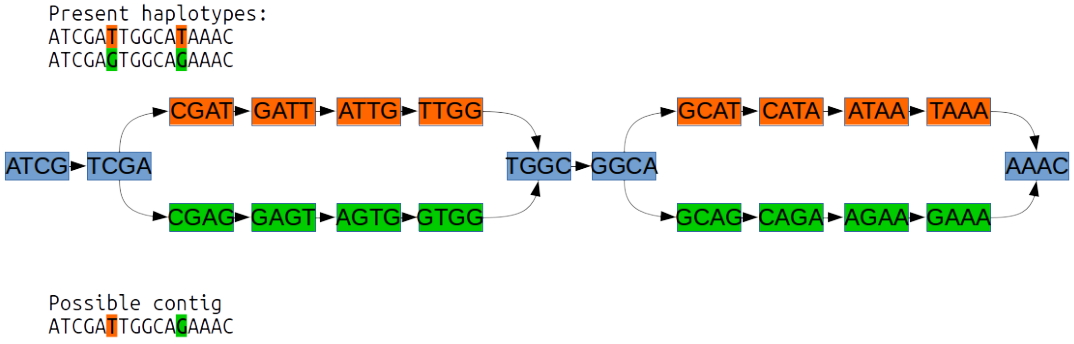

Figure 1.27: Example on how an assembler may crush haplotypes. The assembler detects two bubbles and crushes them, choosing one path over another. The resulting contigs may contain sequences from both haplotype mixed.

whole data (meta-genomic). Some assemblers are designed to assemble meta-genome as meta-IDBA [53] or meta-velvet [54]. As their names suggest, they are usually based on a regular genome assembler adapted to fit to the proposed scenario. In some cases, a single species is sequenced but with several individuals having different, however closely related, genomes (pool-seq). Since repeats assess the toughness of assembly, we can sort scenarii by their apparent hardness:

- Single haploid genomes (some repeats)
- Heterozygous genomes (each homozygous regions is a repeat)
- Pool-seq, multiple related genomes (genome sized repeat)
- Meta-genomic, multiple non-related genomes (genome size repeats, some repeat shared among species)

It seems obvious that perfect assembly and distinguishing closely related genomes are currently out of reach. But this gives objectives to be met in the assembly field.

"Assembly challenges" core messages:

- Finished assembly is often impossible because of repeats longer than the reads
- Other kinds of data may be used to order contigs in larger sequences called "scaffolds"
- Polyploid, meta-genome and pool-seq assembly are even harder than regular assembly because of systematic repeats

1.4 Outline of the thesis

In this introduction, we highlighted three "challenges" of the assembly process of NGS data:

- The high amount of resources required
- The fragmentation of produced assembly
- The hardness to assemble complex (large and/or repeat-rich) genomes

To address those issues, we will present new approaches based on the de Bruijn graph. In the first chapter we present new ways to represent and construct the de Bruijn graph in order to make it more scalable and more efficient. In the second chapter we introduce new methods to use the de Bruijn graph as a reference and show the interest of such structures over a set of fragmented contigs. In the third chapter we propose techniques to construct high order de Bruijn graphs, allowing the use of the reads information, as the string graph does, without losing the de Bruijn graph efficiency.

Chapter 2

Handling assembly

In this chapter we will assess the computational aspect of assembly. We show why the scalability of methods used for genome assembly can be critical (Section 1). We provide an overview of the state of the art of efficient methods and structures to address such problems for both overlap graph (Section 2) and de Bruijn graph (Section 3) frameworks. Then we present our theoretical and practical contributions. We show that de Bruijn graph assembly may be done using a Navigational Data Structure, a novel model that we introduce, which shows lower theoretical memory bounds. We also propose a new proof of concept assembler that shows practical resources improvement over state of the art tools (Section 4). Those implementations are based on the idea to enumerate and index simple paths of the graph. We argue that such enumeration is a bottleneck in many assembly methods and we provide resources efficient methods to answer this need (Section 5). To do so, we make use of minimal perfect hashing functions as very efficient indexes and provide a new method to compute such functions on very large sets of keys (Section 6).

2.1 The assembly burden

In the last section we described the sequencing data but we gave no clue about the typical size of a genome. The fact is that this size can vary a lot among living organisms (Figure 2.2). Though we can indicate some orders of magnitude on genomes size:

- Virus : Thousands base pairs
- Bacteria: Millions base pairs
- Mammals : Billions base pairs

Some species, yet to be sequenced, are expected to present order of magnitude larger genome (Paris japonica, Polychaos dubium) [55] with presumed hundred of billions base pairs. The struggle for assembling such genomes may not be seen directly. Since high coverages are needed for assembly, sequencing datasets represent huge amount of information. Most assemblies typically rely on a mean coverage ranging from 30X to 100X and higher. Consequently, assemblers have to handle millions of reads, even for bacterial genomes. Larger genomes can count hundreds of millions base pairs for most prokaryotes, and up to billions for some mammals or plants genomes. Such sets can reach billions of reads and represent hundreds of gigabytes or even terabytes of data.

Dealing with this tremendous amount of information require either to use huge computational resources or to conceive specific algorithms and data structures designed for resource efficiency. As sequencing cost continues to decrease, sequencing very large genomes becomes affordable, but assembly of such genome is barely possible. If the running time is an important concern, it is usually not the source of intractability. Most assemblers rely on very large graph structure and index, that can require terabytes of RAM on large datasets. Most of the time, such memory requirement is more concerning than running time, as large scale clusters may not be easily accessible. Besides facing the challenge to produce correct assemblies, the future assemblers will have to handle larger and larger datasets, in order to deal with large genomes or meta-genomes while providing a high throughput to follow the sequencing rate. As sequencing costs are dropping (Figure 2.1), the computational resources necessary to treat them is becoming the financial bottleneck.

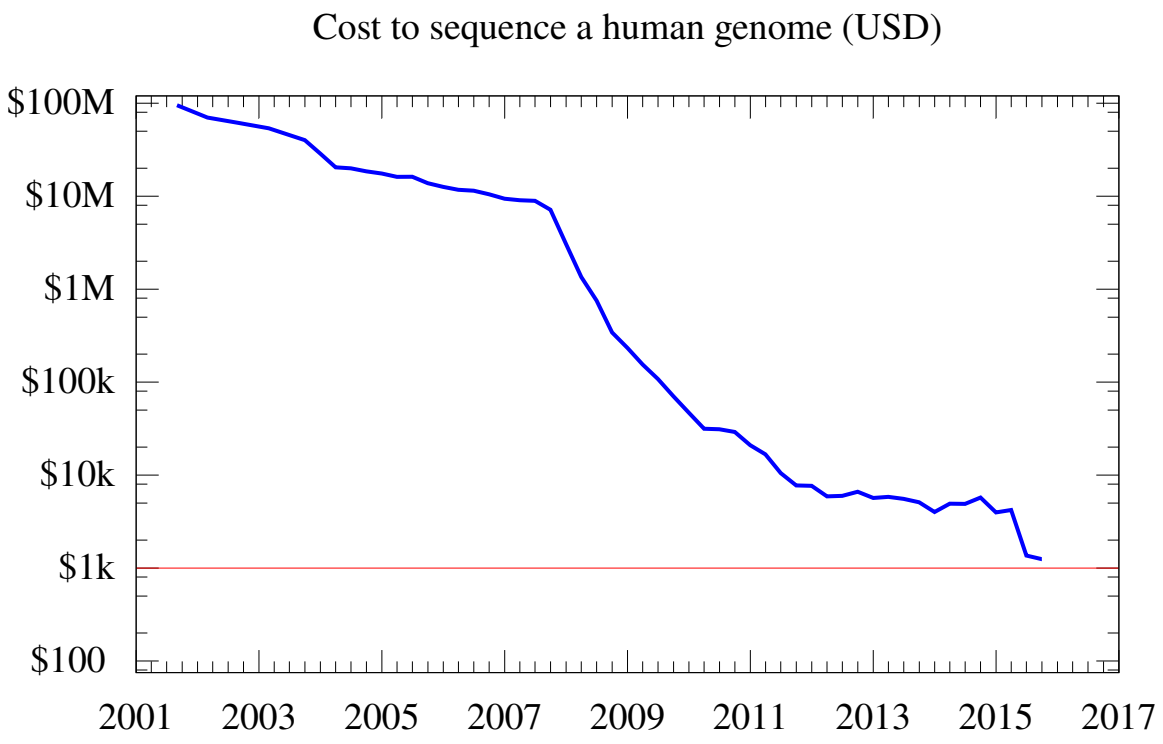

Figure 2.1: Evolution of the cost of human sequencing. From `https://upload.wikimedia.org/wikipedia/commons/e/e7/Historic_cost_of_sequencing_a_human_genome.svg`.

organism	genome size (base pairs)
model organisms	
model bacteria *E. coli*	4.6 Mbp
budding yeast *S. cerevisiae*	12 Mbp
fission yeast *S. pombe*	13 Mbp
amoeba *D. discoideum*	34 Mbp
nematode *C. elegans*	100 Mbp
fruit fly *D. melanogaster*	140 Mbp
model plant *A. thaliana*	140 Mbp
moss *P. patens*	510 Mbp
mouse *M. musculus*	2.8 Gbp
human *H. sapiens*	3.2 Gbp
viruses	
hepatitis D virus (smallest known animal RNA virus)	1.7 Kb
HIV-1	9.7 kbp
influenza A	14 kbp
bacteriophage λ	49 kbp
Pandoravirus salinus (largest known viral genome)	2.8 Mbp
organelles	
mitochondria - *H. sapiens*	16.8 kbp
mitochondria – *S. cerevisiae*	86 kbp
chloroplast – *A. thaliana*	150 kbp
bacteria	
C. ruddii (smallest genome of an endosymbiont bacteria)	160 kbp
M. genitalium (smallest genome of a free living bacteria)	580 kbp
H. pylori	1.7 Mbp
Cyanobacteria S. elongatus	2.7 Mbp
methicillin-resistant *S. aureus* (MRSA)	2.9 Mbp
B. subtilis	4.3 Mbp
S. cellulosum (largest known bacterial genome)	13 Mbp
archaea	
Nanoarchaeum equitans (smallest parasitic archaeal genome)	490 kbp
Thermoplasma acidophilum (flourishes in pH<1)	1.6 Mbp
Methanocaldococcus (Methanococcus) jannaschii (from ocean bottom hydrothermal vents; pressure >200 atm)	1.7 Mbp
Pyrococcus furiosus (optimal temp 100°C)	1.9 Mbp
eukaryotes - multicellular	
pufferfish *Fugu rubripes* (smallest known vertebrate genome)	400 Mbp
poplar *P. trichocarpa* (first tree genome sequenced)	500 Mbp
corn *Z. mays*	2.3 Gbp
dog *C. familiaris*	2.4 Gbp
chimpanzee *P. troglodytes*	3.3 Gbp
wheat *T. aestivum* (hexaploid)	16.8 Gbp
marbled lungfish *P. aethiopicus* (largest known animal genome)	130 Gbp
herb plant *Paris japonica* (largest known genome)	150 Gbp

Figure 2.2: Genome size for a variety of selected organisms. From `http://book.bionumbers.org/how-big-are-genomes/` .

This fast evolving sequencing environment pushes assembly designers to conceive increasingly efficient dedicated methods and data structures. In this chapter, we will present a state of the art of such techniques with a focus on the scalability of the de Bruijn graph assembly.

"The assembly burden" core messages:

- Genomes may count billions of bases
- Sequencing datasets can be composed of billions of reads and represent terabytes of data
- Genome assembly is a High Performance Computing issue
- Efficient and dedicated structures are needed especially to reduce the memory usage
- Decreasing sequencing costs raise the need to efficient assembly

2.2 Overlap graph scalability

The problem of assembly scalability appeared with the overlap graphs and large genomes. Tools such as TIGR [17] or CAP3 [18] can be considered as brute force approaches and were suitable only for small genomes. When the concern about repeat misassemblies arose, the overlap graph approaches have become preferred. The core operation of the overlap graph is to find all overlaps between reads from pairwise alignment. Therefore standard overlap graph assembly methods have a worst time complexity quadratic with the number of reads. This fact may be moderated as only significant overlaps are considered. Heuristics are used in order to quickly find significant alignment. Tools mainly use BLAST [22], or other aligners following the "seed and extend" paradigm. The principle of the seed and extend paradigm is to look for words in common between the query and the reference (called seeds or anchors) and to try to extend the alignment from the shared words. Unlike Smith-Waterman algorithm [56], such approaches are not guaranteed to find all optimal alignments. But they are dramatically faster while providing almost identical results. Using such anchoring heuristics implies that finding the overlaps of a read is roughly linear in the number of similar reads (according to the method and parameter used) instead of being linear in the total number of reads.

However, with high coverage, one read presents a high amount of high quality overlaps (Figure 2.3). This will impact the running time of each query trying to find the overlaps shared by a read. Such alignment based approaches were quite efficient with Sanger sequencing, since the number of reads remained relatively low. With a very low error rate, low coverage is sufficient to complete correct assemblies, around 10X or 20X. Since the sequences are also longer (around thousands of base pairs), the number of Sanger reads is lower than the number of NGS reads for a given coverage. In practice the number of reads, even for large genome assembly, was not above the million. The number of overlaps needed to be stored in order to work on the graph could still be challenging. Furthermore, repeated regions resulted in dense zones (called hub) with very high numbers of overlaps. Celera [21] and Arachne [57] improved the running time by masking repeated regions to

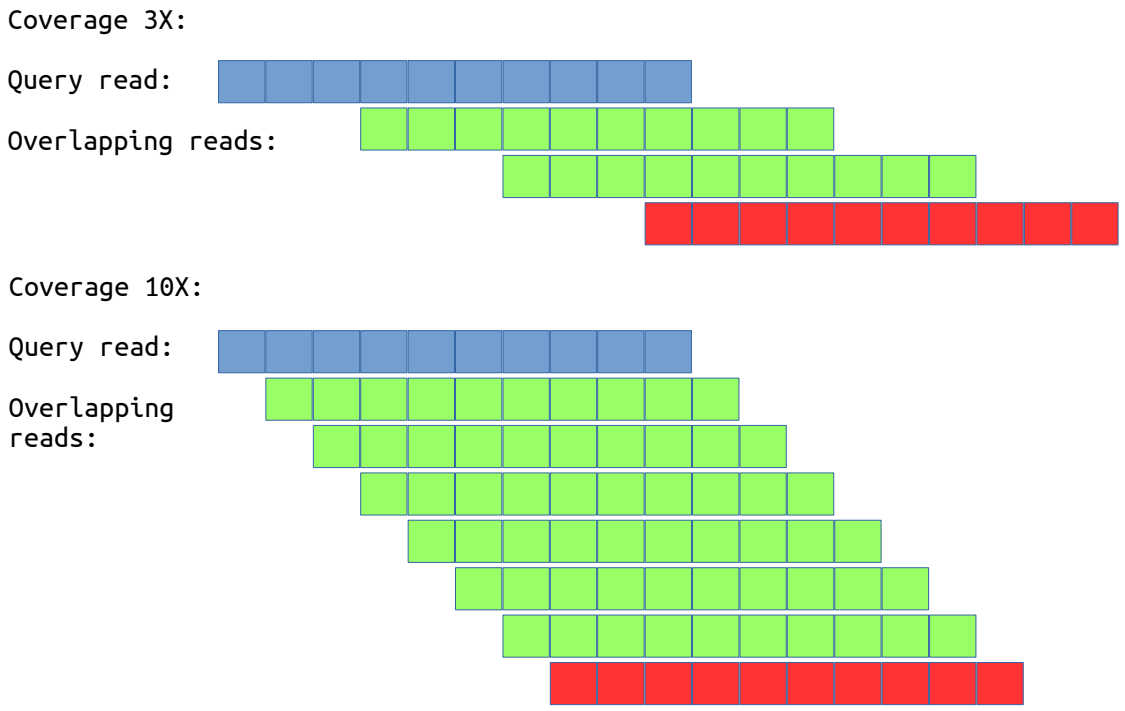

Figure 2.3: How the coverage affects the number of overlaps in a overlap graph. On this toy example, with a 3X uniform coverage, a read shares an overlap superior to 4 bases with 2 other reads in green. With a coverage of 10X each read shares a significant overlap with 6 other reads. The reads in red are overlapping but not detected because of a small overlap with the query read.

simplify the overlap graph. Later, the string graph [25] was proposed to abstract the redundancy present in the sequencing data, as does the de Bruijn graph, using transitive reduction. The assembler BOA [25] was designed to assemble Sanger reads with such string graphs. However with NGS technologies, deeper sequencing was affordable and helped to cope with the error rate. This raised up the number of reads by orders of magnitude, themselves leading to orders of magnitude higher numbers of overlaps to handle. The overlap graph and string graph need enormous resources to process such datasets. Thus de Bruijn graphs were preferred to handle this kind of data.

However, because of the fact that the string graph should theoretically lead to better assembly by using the whole reads instead of kmers, several contributions were proposed in order to allow the string graph to scale up with the tremendous number of short reads. The introduction of the very popular FM-index structure [58] in bioinformatics allowed memory efficient indexing of large DNA sequences. It was first used to index references and enabled very efficient read mapping (fast and stringent local alignment of short reads on a reference). Notable examples of such tools based on a FM-index are Bowtie [59] and BWA [60]. [26] used the same idea to improve the string graph performances. The rationale is to index the reads with a FM-index and to use the research operation to detect overlaps between sequences. The memory usage of the assembly of a human genome has been extrapolated by the author to 700GB, which is not intractable but still very expensive. The time required for assembly is also tractable, days for 100 mega-bases genome, but this is yet very slow compared to most de Bruijn graph assemblers. A more efficient implementation called SGA [27] proposed a distributed FM-index construction. Using multiple small indexes allows for the construction of a global FM-index using less than 60 GB of RAM for a human sequencing dataset of more than 1.2 billions reads. The efficient use of the FM-index allowed the string graph to scale up to large genomes with moderate memory usage, but the computational resources remained high. The human genome assembly of SGA used more than 1500 CPU hours over 6 days.

"Overlap graph scalability" core messages:

- Overlap graphs and string graphs usually rely on memory heavy data structure
- Compressed data structures may be used but still lead to very long assembly processes
- String graphs grow roughly with the size of the dataset when de Bruijn graph grow with the size of the sequenced genome

2.3 The scaling story of the de Bruijn graph representation

The main difference between the overlap graph and the de Bruijn graph is the memory usage. The overlap graph grows linearly with the size of the sequencing data. The de Bruijn graph memory usage can (with kmer filtering) be linear in the size of the genome, since we only have roughly $GenomeSize$ different kmers in the genome, estimated through their abundances. A very low abundance threshold would keep too much kmers and

therefore use too much memory. However, even with a very low threshold the number of non genomic kmers is hardly very high (Table 2.1). Thus the memory usage used by a de Bruijn graph is rather function of the size of the genome than of the size of the sequencing dataset.

Largest genomes still lead to graphs with billions of nodes and edges, that are not easy to represent with a decent amount of memory. In this section we concentrate on the problem of storing a huge de Bruijn graph in memory. We present here an overview of the main milestones and a state of the art of the data structures used to represent a de Bruijn graph. We describe the first representations that were based on classical graph implementation and exhibit very high memory footprints. Next we introduce the work of Conway and Bromage that proposed a lower bound on the necessary memory to represent a de Bruijn graph. Then we see how the use of external memory highly impacted the memory usage of most assemblers. And finally we present very efficient techniques based on probabilistic data structures. We also present several works on the scaling of de Bruijn graph assembly on a massive number of cores.

2.3.1 Generic graphs

The earliest representations of a de Bruijn graph were based on classical and generic graph representations and were not memory efficient. The first graph representations

Reference genome	Abundance Threshold	Erroneous kmers	Missing kmer
E. coli	1	89,044,977	5
E. coli	2	3,889,707	6
E. coli	3	200,173	11
E. coli	4	15,198	11
E. coli	5	1,913	13
E. coli	6	312	16
E. coli	7	33	20
E. coli	8	0	21
C. elegans	1	1,909,728,320	6
C. elegans	2	89,087,143	7
C. elegans	3	7,294,417	11
C. elegans	4	2,003,946	15
C. elegans	5	1,049,812	21
C. elegans	6	643,918	26
C. elegans	7	425,435	26
C. elegans	8	293,457	75
C. elegans	9	208,539	213
C. elegans	10	150,939	589

Table 2.1: Number of erroneous and missing kmers in a de Bruijn graph after an abundance filtering. On simulated reads with a 100X coverage, 1% error rate and k=51 from the *E. coli*. The *E. coli* genome presents 4,554,207 genomic 51mers. The *C. elegans* genome presents 100,286,051 genomic 51mers.

were based on hash tables indexing kmers associated with informations allowing graph traversal, as in EULER [31] or Velvet [33] assemblers. The use of a dynamic hash table for indexing millions of elements rapidly lead to very high memory usage. Velvet's paper reports a memory usage of 2GB for a Streptococcus genome of 2.2 millions bases. The proposed solution considered for scaling was the use of disk in order to store the graph structure. However this kind of solutions was never really put in practice since disk accesses are very slow compared to RAM accesses. Thus such methods would present orders of magnitude longer running time. Such data structure is adapted to bacterial genomes assembly but impracticable on larger one.

In order to address this scalability issue, Abyss [40] introduced several solutions. First the indexing method changed, using an interesting property of the de Bruijn graph: the numbers of fathers and of sons of a node is bounded by four (the size of the nucleic alphabet). In Abyss, all kmers are indexed and the value associated is just eight bits coding the presence or absence of its eight possible neighbors. Secondly, the main cost of a hash table is usually the pointers used for its internal structure. Instead of this, they used open addressing hash tables that represent a low overhead of a few bits by elements. Thirdly they proposed the distribution of the graph on several machines. The idea of their distributed de Bruijn graph is to use a first hash to decide on which hashtable/machine the kmer will be inserted. On large genome, memory usage was still high and some informations had to be transferred via the network between machines. Still, Abyss was the first assembler scalable enough to assemble a human genome. Today some assemblers still require Terabyte level of memory to assemble large genomes as ABYSS that assembled the white spruce using 4.3TB of memory distributed on large memory servers [61] or DiscovarDenovo that used almost 2 terabytes for a human assembly [62]. The main reason is their too memory expensive indexing methods. In the following we present several works that assess the problem of designing smaller data structures to make large genomes assembly more tractable.

2.3.2 Theoretical limits

Conway and Bromage [63] analyzed the theoretical minimal number of bits necessary to encode a de Bruijn graph. They first noted that only the nodes need to be encoded, since edges could simply be deduced from the presence of nodes. Thus a de Bruijn graph can be defined as its set of kmers. Therefore the minimal number of bits needed to encode a de Bruijn graph of n kmers is

$$bits = log\binom{4^k}{n}$$

That is $\Omega(n.log(n))$ if $4^k > n$. The paper gives the example of a human genome with 5 billions 25-mer that needs at least 12 GB of memory. A regular bit array that may contain all possible kmers would be impossible to store for k larger than 20. Such bit array would be extremely sparse. They therefore propose the use of succinct data structures to represent such arrays that can be extremely compressed based on Elias-Fano encoding (Sarray) [64]. They provide an implementation of this technique in a tool called Gossamer [43]. It achieved a memory usage close to their proposed bound and a proof of concept assembly of a human genome.

2.3.3 Kmer Counting

One may be surprised by the apparent contradiction between the assertion that a de Bruijn graph assembler use memory according to the number of genomic kmer and the fact that previously presented tools showed so high memory usage. The problem is that such tools have to deal with all kmers from the reads, not just the genomic ones. Because of sequencing errors, the number of different kmers before filtering is huge. For example a simulated 100X coverage dataset of *E. coli* with 1% error rate contains 79,917,279 erroneous kmers for only 4,554,207 genomic ones. In order to avoid the indexing of a huge amount of erroneous kmer, a new category of tools called kmer counters has been proposed. The goal of a kmer counter is to associate to each kmer of a dataset its abundance, in order to keep only the solid ones for the assembly, using the lowest amount of resources. Jellyfish [65] proposed the use of an open addressing hash table [66]. The use of this pointer-less table allows the counting operation to use order of magnitude less memory.

DSK [67] proposed a disk based algorithm with an extremely low memory footprint and achieved to count the kmers of a human genome using only 4 GB of memory where JellyFish used 70GB. Although the use of disk operations slowed the process, using 18 hours instead of 3.5 for Jellyfish. However the use of Solid State Drive (SSD) allows DSK to be as fast as jellyfish. KMC2 [68] improved the DSK paradigm by compacting some kmers together reducing the disk footprint. This approach enables a great speedup and is even faster than JellyFish.

These algorithms allowed the filtering of a huge part of the erroneous kmers with a low memory usage. Following assemblers could rely on such techniques in order to work on a set of mostly genomic kmers and therefore present order of magnitude lower memory usage.

2.3.4 Probabilistic de Bruijn graphs

Several approaches proposed the use of probabilistic structures in order to leverage the memory footprint. SparseAssembler [69] or LightAssembler [70] propose a sub-sampling of kmers based on the fact that most genomic kmers will appear a high number of times. They index only a subsample, storing 1 out of g read kmers. It ensues that most genomic kmer are still contained in the sub-sample, while the number of erroneous kmer is roughly divided by g. A more aggressive sub-sampling can be done on high coverage. LightAssembler proposes by default $g = 3$ with 25X and a sub-sampling up to $g = 25$ for 280X. Even if those parameters are conservative, such techniques are still more likely to lose kmers than exact approaches because they highly rely on uniform coverage and error distribution. Such techniques may be criticized for not using the whole information present in their data and possibly missing genomic kmers.

Another proposition was to index the kmers thanks to a Bloom filter [71]. The Bloom filter is a probabilistic set presenting false positives but no false negatives, and having the property of using very low memory. A query to a Bloom filter is of the type "Is this kmer in the set ?". If the answer is "no", then we are sure that the kmer is not in the set, meaning there is no false negative. If the answer is "yes", then the queried kmer is likely to be in the set. A query on a kmer not inserted in the set may return "yes". Such errors

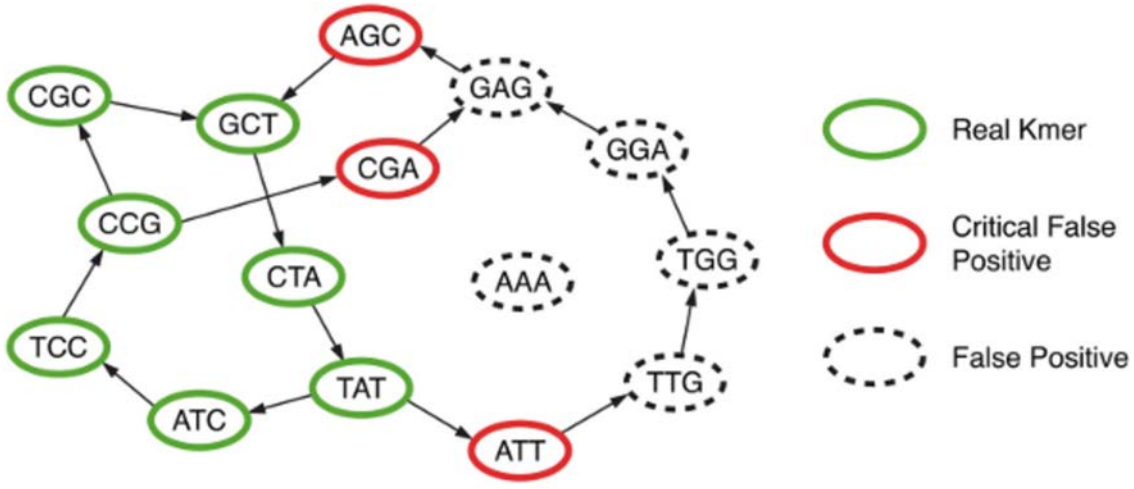

Figure 2.4: How Minia represent an exact de Bruijn graph storing a set of critical false positive. Source `http://minia.genouest.org/`. The Bloom filter creates false positives in black and red. False positives not connected to the graph do not matter since they will not be queried. Only false positives that are connected to the graph are indexed. This way an assembly on the real kmer can be performed.

are false positives. The probability of such an error to appear can be controlled by using more memory, a dozen of bits per kmer can lead to a false positive rate around 1%. In the context of assembly, kmers are usually inserted into the Bloom filter during an indexing phase and the filter is queried during the assembly step. Most of the false positives are harmless to the assembly process, most of them are kmers disconnected from the graph and will never be visited or queried. But in some rare cases, some false positive kmers can be connected to the graph and induce errors or fragmentations.

An improvement was proposed in Minia [72], that stores those "critical false positives" in order to obtain a lossless de Bruijn graph representation(Figure 2.4). This structure was itself improved using several Bloom filters in order to store the critical false positives [73]. Minia successfully assembled a human genome with less than 6 GB and less than one day with one processor. On the same dataset, Gossamer used more than 30 GB and 50 hours and AByss more than 300 GB and 15 hours.

2.3.5 Navigational data structures

One may be surprised to see that Minia achieved the assembly of a human genome with only 6 GB (less than 16 bits per kmer) where the theoretical limit was supposed to be the double. The Conway and Bromage theoretical limit was the number of bits necessary to index a set of kmer. The lower memory usage of Minia is due to the fact that it relies on a data structure specific to assembly. Minia structure may answer wrongly to membership query, because the storing of the critical false positives are only designed to block the assembler to "exit" from the de Bruijn graph. The Minia structure is therefore not an exact representation of the set of kmers which explains why it is able to use less memory

than the theoretical limit. Minia takes advantage of the fact that during the assembly step, kmer membership queries are not random as we are interested in the existence of a neighbor in order to go through the paths of the graph. The membership operation itself is thus not primordial. Minia is able to answer correctly to neighbor-ship queries (membership of neighbors of a node), used for assembly, with a lower memory usage than needed to index exactly all kmers. We call such representation a "navigational data structure" as introduced by [74] as opposed to a "membership data structure". This paper introduces the notion of Navigational Data Structure (NDS) and shows that it can be used to perform de Bruijn graph assembly. The interest of such structures is the lower bound on their sizes, a NDS needs at least 3.24 bits per kmer to represent a de Bruijn graph. Future assemblers could rely on implementation of such data structure to provide highly reduced memory usage.

2.3.6 Massively parallel assembly

In previous sections we talked mostly about the memory bottleneck of assembly. But the time required to perform the assembly step can also be a limitation in many analyses, especially for large genomes or large collections of genomes.

Several contributions were proposed in order to offer fast genome assembly thought the use of massive multi-core servers. Since most assembly algorithms consist mainly in graph traversal operations, it is difficult to really increase the throughput by optimizing these operations since they essentially rely on memory accesses. The main way to improve the wall clock time is to perform the graph traversal operations in parallel. Some assemblers are able to use multiple threads to accelerate their processing but the gain is usually limited. In its benchmark, HipMer [75] performed a human assembly in less than one hour on a thousand cores server while other parallel assemblers like Abyss were not able to make an efficient use of such architectures and were more than ten time slower. The challenge in order to enable massive parallelization is to limit the threads communications. The question of memory access is also critical, when the massive parallelization requires the use of multiple machines communicating via very slow network accesses. Massively parallel assemblers rely on efficient graph partitioning strategies following the divide and conquer paradigm in order to improve data locality and to limit synchronization messages between threads dealing with a local task. If tools like HipMer or SWAP [76, 77] are able to use thousands of cores, those optimizations lead to a very high memory usage and more generally very high resource consumption.

"The scaling story of the de Bruijn graph representation" core messages:

- Regular data structures like hash table do not scale on large genome
- Efficient use of external memory to filter erroneous kmer help assemblers to index mostly genomic kmer
- Memory efficient and specific data structures can lead to moderate memory usage to assemble large genomes

Reference genome	# Unitig	N50	# kmer
E. coli	1,520	67,344	4,567,544
C. elegans	127,106	13,605	96,501,920
Human	2,755,964	4,967	2,768,098,045

Table 2.2: Statistics on unitigs created from references genomes with k=63. The N50 value is a metric to evaluate an assembly. Given a set of sequences of varying lengths, the N50 length is defined as the length N for which 50% of all bases in the sequences are in a sequence of length $L < N$. Intuitively, we can cover 50% of the assembly with sequences larger than the N50 value.

2.4 Efficient de Bruijn graph representation

In this section we describe our contribution in the paper "On the representation of de Bruijn graphs" [74]

2.4.1 Compacted de Bruijn graph

In practice, an assembly de Bruijn graph can often be decomposed into a set of long simple paths. A simple path is a path whose nodes exhibit in and out degrees of one, except the first and last nodes. A maximal simple path is a simple path that is not included in a larger simple path. A nodes-disjoint set of such maximal simple paths that cover the de Bruijn graph can represent its set of kmers (Figure 2.5). We call the simple paths from such a set "unitigs". But each unitig composed of p kmers can be represented as a string of length $p + k - 1$. We can therefore represent a de Bruijn graph of n kmers with less than

$$2(n + (k-1)\#unitigs)\ bits$$

$2n$ bits for the genome sequences and an overhead of $2(k-1)$ bits for each unitig. This graph of unitigs is called the compacted de Bruijn graph. Unitigs construction is often the first step of the assembly process. The unitig set can be considered as a safe assembly, since graph modification heuristics are applied on the unitigs in order to get longer contigs sequences. Two points can be observed in Table 2.2:

- A genome presents orders of magnitude less unitigs than kmers
- Unitigs can represent quite large sequences

Those points show the interest to consider unitigs instead of kmers as nodes of the graph.

2.4.2 De Bruijn graph construction

The question asked here is "Can we take advantage of the unitig representation in order to propose a data structure that achieves a memory consumption close to 2 bits per kmers?" The paper [74] propose a data structure called "DBGFM" to represent a De Bruin graph in low memory. The idea is to index the de Bruijn graph unitigs in a FM-index [58]. The

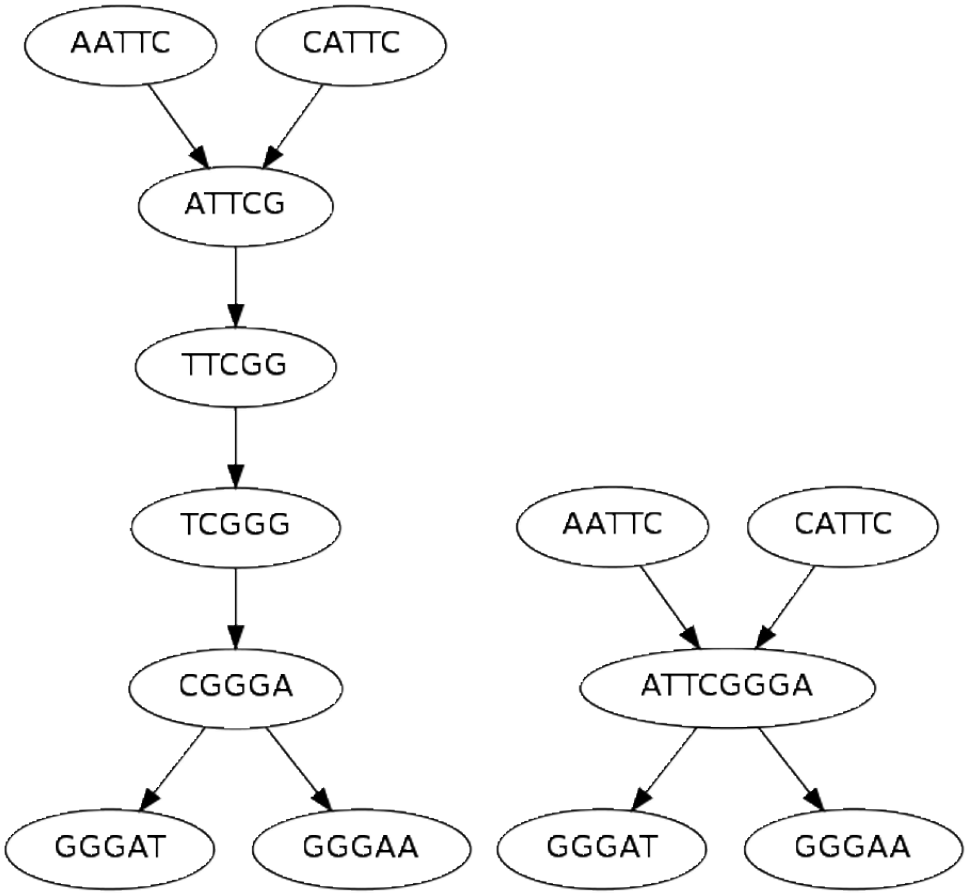

Figure 2.5: Toy example of a de Bruijn graph (left) and the corresponding compacted de Bruijn graph (right). The nodes of the compacted de Bruijn graph are no longer kmers but sequences of length $\geq k$ and called unitigs.

FM-index is a lightweight full-text index based on the Burrows-Wheeler transform [78]. Such a structure can answer membership and neighborhood queries. It allows to count the number of occurrences of a pattern q in $\mathcal{O}(|q|)$. We can therefore perform membership queries of a kmer in $\mathcal{O}(|k|)$. But the main interest is that we can access the in-neighbors of a node in constant time by querying the symbols preceding a kmer. Hence, DBGFM is a membership data structure that also provides fast in-neighbors query.

The DBGFM structure was integrated into the ABySS assembler as a proof of concept of low memory assembly. But straightforward compacted de Bruijn graph construction requires to store the de Bruijn graph itself in memory. This de Bruijn graph compaction step would therefore become the bottleneck of the assembly workflow. To fulfill this need, we proposed a new external memory based algorithm in order to compute the compacted de Bruijn graph with low memory called BCALM.

BCALM algorithm is based on the idea of minimizer [79]. The l minimizer of a string u is the smallest $l$mer that is a substring of u according to a total ordering of the strings e.g. lexicographical. We define the left minimizer Lmin(u) and the right minimizer Rmin(u) as the minimizer of the $k-1$ prefix of u and the minimizer of the $k-1$ suffix of u respectively. We use the binary relation $u \to v$ between two strings that denotes an exact suffix-prefix overlap of length $k-1$ between u and v. The use of minimizers is motivated by the following property: For two strings u and v, if $u \to v$ then Rmin(u)=Lmin(v)

For describing the BCALM algorithm we will rely on the notions of compatibility introduced in the next paragraph.

Given a set of strings S, we say that $(u, v) \in S^2$ are *compactable* in a set $V \subseteq S$ if $u \to v$ and, $\forall w \in V$, if $w \to v$ then $w = u$ and if $u \to w$ then $w = v$. The compaction operation is defined on a pair of compactable strings u, v in S. It replaces u and v by a single string $w = u \cdot v[k+1 \ldots |v|]$ where '$\cdot$' is the string concatenation operator. We say that two strings (u, v) are *m-compactable* in V if they are compactable in V and if $m = Rmin(u) = Lmin(v)$. The m-compaction of a set V is obtained by applying the compaction operation as much as possible in any order to all pairs of strings that are m-compactable in V. It is easy to show that the order in which strings are compacted does not lead to different m-compactions. Compaction is a useful notion because a simple way to obtain the unitigs is to greedily perform compaction as long as possible.

2.4.3 Unitig enumeration in low memory: BCALM

The BCALM algorithm is described in Figure 2.6, and a step by step execution is presented in Figure 2.7. The first step consists into placing the input kmers into different files F_m according to their minimizers. Then, each file is processed starting from the file with the smaller minimizer in increasing order. Each file F_m is loaded in memory and all possible m-compaction among the sequences of the file are applied. The idea is that each file will be small enough to be compacted efficiently using low amounts of memory. Each sequence of the file are thereafter placed in the output file or in another file to be eventually further compacted. The rules of choosing which file to write a sequence is based on its minimizers. If both the left and right minimizers are below m, the sequence is output as a unitig in the output file. Otherwise we write the sequence in the file $F_{m'}$ where m' is the smallest minimizer bigger than m of the sequence. Finally the file F_m is discarded and the next file is processed.

1: **Input:** Set of kmers S, minimizer size $\ell < k$
2: **Output:** Sequences of all simple paths in the de Bruijn graph of S
3: Perform a linear scan of S to get the frequency of all l-mers (in memory)
4: Define the ordering of the minimizers, given by their frequency in S
5: Partition S into files F_m based on the minimizer m of each k-mer
6: **for** each file F_m in increasing order of m **do**
7: $C_m \leftarrow m$-compaction of F_m (performed in memory)
8: **for** each string u of C_m **do**
9: $B_{min} \leftarrow$ min(Lmin (u),Rmin (u))
10: $B_{max} \leftarrow$ max(Lmin (u),Rmin (u))
11: **if** $B_{min} \leq m$ and $B_{max} \leq m$ **then**
12: Output u
13: **else if** $B_{min} \leq m$ and $B_{max} > m$ **then**
14: Write u to $F_{B_{max}}$
15: **else if** $B_{min} > m$ and $B_{max} > m$ **then**
16: Write u to $F_{B_{min}}$
17: **end if**
18: **end for**
19: Delete F_m
20: **end for**

Figure 2.6: BCALM: Enumeration of all maximal simple paths in the De Bruijn graph

The idea of the algorithm is to perform the m-compactions for each m in order to perform all compactions. The intuition of the rule to place sequences in the correct file is that, if the suffix (or the prefix) has a overlap with a minimizer n superior to m, then it will eventually be compacted in the file F_n. If both minimizers are below m, no further compactions can be made on the sequence and it is therefore an unitig. If both minimizers are above m then the smallest is chosen, as the algorithm treats minimizers in increasing order.

Several implementation details allow this algorithm to be practical. Reverse complements are handled as presented before by identifying each kmer with its reverse complement and letting the minimizer be the smallest $l$mer in both of them. To avoid the creation of too many files, we encode several virtual files in one physical one, this allowed to use $l = 10$ in our experiments. We also avoid to load in memory the whole sequences of a file as only the $k-1$ prefix and suffix are used by the compaction detection algorithm. For a fixed input S, the number of strings in a file F_m depends on the minimizer length l and the ordering of minimizers. When l increases, the number of $k-1$mers in S that share a given minimizer decreases. Thus, increasing l yields less strings per file, which decreases the memory usage. We realized that, when highly-repeated $l$mers are less likely to be chosen as minimizers, the sequences are more evenly distributed among files. We therefore perform in-memory $l$mer counting (line 3) to obtain a sorted frequency table of all $l$mers. Each $l$mer is then mapped to its rank in the frequency array, to create a total ordering of minimizers (line 4). Our experiments showed a drastic improvement over lexicographic ordering (Table 2.3).

A proof of correctness of this algorithm is showed in [74].

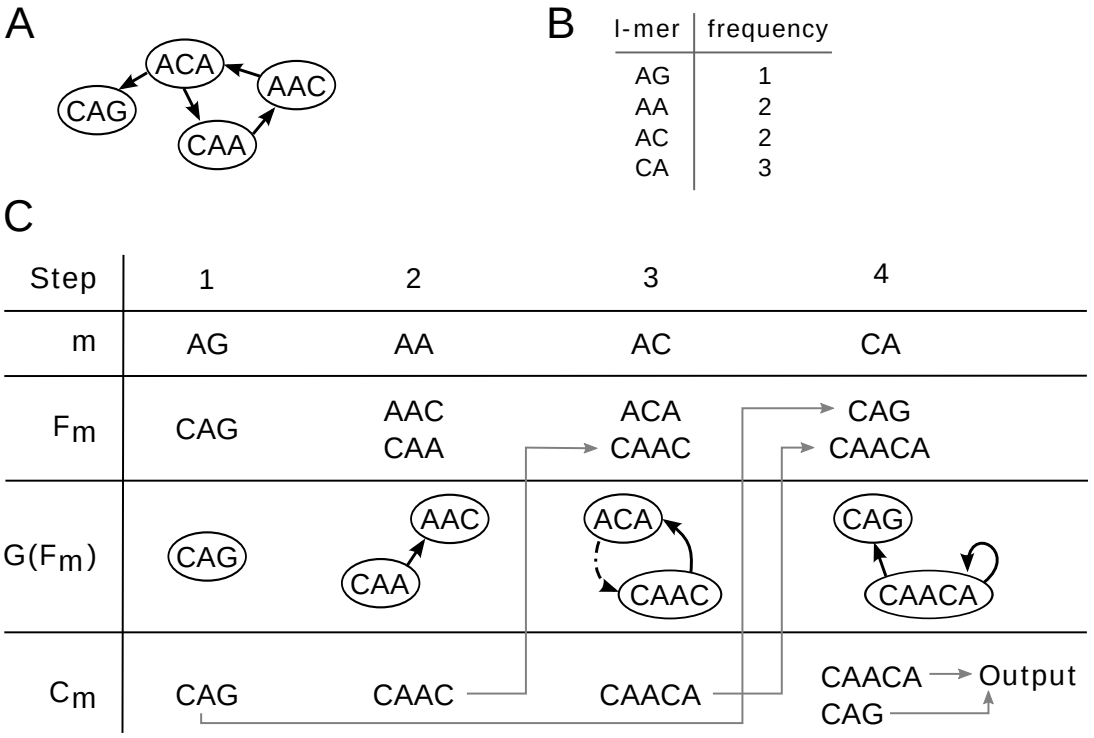

Figure 2.7: An execution of th BCALM algorithm on a toy example. (Panel A) The set S = ACA, AAC, CAA, CAG of k-mers is represented as a de Bruijn graph (k = 3). (Panel B) The frequencies of each l-mer present in the kmers is given in increasing order ($l = 2$). (Panel C) Steps of the algorithm; initially, each set F m contains all kmers having minimizer m. For each minimizer m (in order of frequency), perform the m-compaction of F_m and store the result in C_m ; the grey arrows indicate where each element of C_m is written to, either the file of a higher minimizer or the output. The row $G(F_m)$ shows a graph where solid arrows indicate m-compactable pairs of strings in F_m. The dash-dot arrow in Step 3 indicates that the two strings are compactable in F_m but not m-compactable; in fact, they are not compactable in S.

Ordering used	Lexicographical	Uniformly random	$l$mer frequency
Memory usage	840 MB	222 MB	19 MB

Table 2.3: Memory usage of BCALM with three different minimizer orderings: lexicographical, uniformly random, and according to $l$mer frequencies. The dataset used is the human chromosome 14 with $k = 55$ and $l = 8$

2.4.4 Assembly in low memory using BCALM

We tested the efficiency of our algorithm in two datasets. The first dataset is 36 million 155bp Illumina human chromosome 14 reads (2.9 GB compressed fastq) from the GAGE benchmark [80]. The second dataset is 1.4 billion Illumina 100bp reads (54 GB compressed) from the NA18507 human genome (SRX016231). We first processed the reads with kmer counting software. We used DSK for this operation for its low memory usage. We set k to 55 and applied a solidity theshold of 5 for chr14 and 3 for the whole genome.

First we evaluate (Table 2.3) the memory usage of BCALM on the chr14 dataset, according to the minimizer ordering used. We observe that the lexicographical order do not perform well. The lexicographical smallest $l$mer is A^l, which is abundant in the human chromosomes, resulting in a large F_{m0} and a high memory usage. A second attempt has been made with a uniformly random ordering. If it performs better than the lexicographical one, it is still sensible to repeated $l$mer in the first files resulting in high memory usage. Finally, we ordered the $l$mers according to their frequency in the dataset. This resulted in a memory usage of 19 MB, a 40-fold improvement over the initial lexicographical ordering. The running times of all three orderings were comparable.

We also evaluated the effect of the size of the minimizer on the BCALM performances in Table 2.4. Large l generally lead to small memory usage, however we did not see much improvement past $l = 8$ on this dataset.

Using BCALM, we evaluated the performances of the low memory assembly workflow with DSK, BCALM and DBGFM in Table 2.5. For the whole genome dataset, BCALM used only 43 MB of memory to compact a set of $2.5 * 10^9$ 55mers and output 40 million sequences of total length 4.6 billions base pairs. DBGFM represented these paths in an index of size 1.5 GB. The overall construction time, including DSK, was roughly 24 hours. In comparison, a subset of this dataset was used to construct the data structure of Salikhov et al [73] in 30.7 hours.

We also compared the memory usage of the proposed workflow with the other state of the art approaches in Table 2.6, showing the interest the proposed low memory assembly workflow.

Those results show that indexing unitigs instead of kmers can lead to a memory efficient assembly steps. The BCALM algorithm, allows an memory efficient construction of the unitig set by relying on external memory.

Minimizer size l	2	4	6	8	10
Memory usage	9,879 MB	3,413 MB	248 MB	19 MB	19 MB
Running time	27m19s	22m2s	20m5s	18m39s	21m4s

Table 2.4: Memory usage and wall clock time of BCALM with increasing values of minimizer sizes l on the chr14 data. With the system of virtual files, these values of l require respectively 4,16,64,256 and 1024 physical files. The used ordering is the one based on $l$mer counts.

Dataset	DSK	BCALM	DBGFM
Chromosome 14	43 MB	19 MB	38 MB
	25 mins	15 mins	7 mins
Whole human genome	1.1 GB	43 MB	1.5 GB
	5 h	12 h	7 h

Table 2.5: Running times (wall-clock) and memory usage of DSK, BCALM and DBGFM construction on the human chromosome 14 and whole human genome datasets ($k = 55$ and $\ell = 10$ for both).

	DBGFM	Salikhov *et al.*	Conway & Bromage
chr14	38.0 MB	94.9 MB	> 875 MB
Full human dataset	1,462 MB	2,702 MB	$> 22,951$ MB

Table 2.6: Memory usage of de Bruijn graph data structures, on the human chromosome 14 and whole human genome datasets ($k = 55$ for both). We did not run the algorithm of Conway and Bromage because our machine does not have sufficient memory for the whole genome. Instead, we report the theoretical size of their data structure, assuming that it would be constructed from the output of DSK. As described in [81], this gives a lower bound on how well their implementation could perform.

"Efficient de Bruijn graph representations" core messages:

- Unitigs (maximal simple paths) are great abstractions to represent the de Bruijn graph
- Unitigs can be constructed with a low memory usage
- Indexing unitigs with a compressed index achieves very low memory assembly (less than 1.5GB for a human assembly)

2.5 Efficient de Bruijn graph construction

In this section we describe our contribution in the paper "Compacting de Bruijn graphs from sequencing data quickly and in low memory " [82]

We introduced an algorithm to construct the de Bruijn graph with very low memory usage relying on external memory. But the approach does not exhibit straightforward parallel patterns to exploit and is quite slow. In the proof of concept human assembly, BCALM took 12 hours to complete for a total running time of 24 hours for the whole workflow. As we said before, the kmer counting step tool reached a very high throughput and an extremely reduced memory footprint. The graph compaction step has therefore become the bottleneck for assembly. We showed before that unitigs could be used as atoms for assembly instead of kmers and it appears that most assemblers rely on them or even, in some cases, explicitly compute such sets of sequences [83, 40]. Some papers proposed solutions for fast graph compaction but are mostly based on parallel graph traversal and no memory efficient method have been proposed. This is why we propose a fast and parallel algorithm for graph compaction presenting a low memory footprint called BCALM2.

2.5.1 BCALM2 algorithm

We give here a high level overview of the BCALM2 algorithm (Figure 2.8). As in BCALM the first step of the algorithm is to distribute the input set of kmer into files. In BCALM2 a kmer can be inserted in two different files, if it has different left and right minimizers. The second step will consist into performing compactions on all the files in parallel according to the compaction procedure described in Figure 2.9. The last stage will need to glue back together the kmers that were duplicated in two files. The Figure 2.12 shows an execution of BCALM2 on a toy example.

Since the size of the files are small, the compactions can be made with a in-memory algorithm in a parallel way. The resulting sequences are written on disk and are processed in the third stage when all compactions are finished. At the end of the compaction step, we notice that some kmers exist in two copies in the sequences produced. Such kmers are always the prefix or the suffix of the compacted sequences. We record the sequences ends that have a doubled kmers has "lonely" as they should be reunited in the third stage. The other sequences are output as unitigs.

In the last stage, we process the sequences that need to be reunited with the Reunite procedure (Figure 2.10). The goal of Reunite is to associate the sequences sharing a suffix or prefix kmer and to glue them together (Figure 2.11) by reuniting the two occurrences of a doubled kmer. The lonely mark are transfered to the produced sequences and this process is repeated until a sequence has a lonely end. After the reunite operation, there is no duplicated kmer left and the output correspond to the unitig set. In order to be efficient, the reunite operation partitions the sequences in order to process the partitions in a parallel and memory efficient way.

A proof of correctness of this algorithm is available in [82]

2.5.2 Implementation details

In this section we describe optimizations and important implementation details. As in BCALM, we do not use the lexical ordering for minimizers but the frequency based minimizer order described previously. Files used in the algorithm are virtual files organized into groups, in order to introduce natural checkpoints in BCALM 2 in between parallel sections. BCALM2 iterates sequentially through the groups, but parallelizes the processing within a group. The For loop at line 1 of Figure 2.8 is executed in parallel within a

Input: the set of k-mers K.

1: **for all parallel** $x \in K$ **do**
2: Write x to $F(\text{Lmin}(x))$.
3: **if** $\text{Lmin}(x) \neq \text{Rmin}(x)$ **then**
4: Write x to $F(\text{Rmin}(x))$.
5: **end if**
6: **end for**
7: **for all parallel** $i \in \{1, \ldots, 4^\ell\}$ **do**
8: Run $CompactBucket(i)$
9: **end for**
10: $Reunite()$

Figure 2.8: BCALM2(K)

1: Load $F(i)$ into memory.
2: $U \leftarrow i$-compaction of $F(i)$.
3: **for all** strings $u \in U$ **do**
4: Mark u's prefix as "lonely" if $i \neq \text{Lmin}(u)$.
5: Mark u's suffix as "lonely" if $i \neq \text{Rmin}(u)$.
6: **if** u's prefix and suffix are not lonely **then**
7: Output u.
8: **else**
9: Place u in the Reunite file
10: **end if**
11: **end for**

Figure 2.9: CompactBucket(i)

Input: the set of strings R from the Reunite file.

1: UF $\leftarrow$ Union find data structure whose elements are the distinct k-mer extremities in R.
2: **for all parallel** $u \in R$ **do**
3: **if** both ends of u are lonely **then**
4: $UF.union(suf_k(u), pre_k(u))$
5: **end if**
6: **end for**
7: **for all parallel** classes C of UF **do**
8: $P \leftarrow$ all $u \in R$ that have a lonely extremity in C
9: **while** $\exists u \in P$ that does not have a lonely prefix **do**
10: Remove u from P
11: Let $s = u$
12: **while** $\exists\ v \in P$ such that $suf_k(s) = pre_k(v)$ **do**
13: $s \leftarrow Glue(s, v)$
14: Remove v from P
15: **end while**
16: Output s
17: **end while**
18: **end for**

Figure 2.10: Reunite()

Input: strings u and v, such that $suffix_k(u) = prefix_k(v)$.

1: Let $w = u \odot^k v$.
2: Set lonely prefix bit of w to be the lonely prefix bit of u.
3: Set lonely suffix bit of w to be the lonely suffix bit of v.
4: **return** w

Figure 2.11: Glue(u, v)

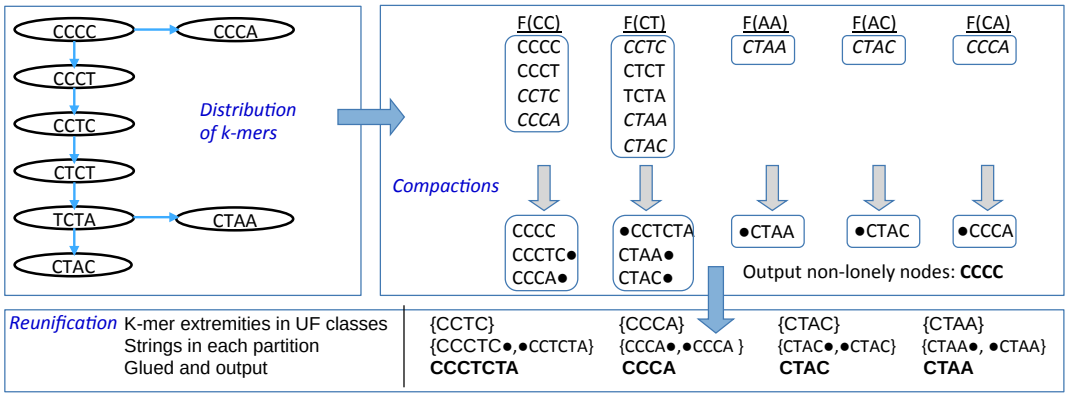

Figure 2.12: Execution of BCALM2 on a small example, with k = 4 and l = 2. On the top left, we show the input de Bruijn graph. The maximal unitig corresponds to the path from CCCT to TCTA (spelling CCCTCTA), and to the kmers CCCC, CCCA, CTAC, CTAA. In this example, minimizers are defined using a lexicographic ordering of lmers. In the top right, we show the contents of the files. Only five of the files are non-empty, corresponding to the minimizers CC, CT, AA,AC and CA. The doubled kmers are italicized. Below that, we show the set of strings that each i-compaction generates. For example in the file CC, the kmers CCCT and CCTC are compacted into CCCTC, however CCCC and CCCT are not compactable because CCCA is another out-neighbor of CCCC. The lonely ends are denoted by •. In the bottom half we show the execution steps of the Reunite algorithm. Nodes in bold are output

group, with each thread given a subset of K. Kmers are distributed only to those files that are in the group, with other buckets being ignored. After the kmers are distributed, files from a group are compacted in parallel. The CompactBucket routines are independent of each other, and hence we run CompactBucket(i) in parallel using all available processors. After BCALM2 finishes processing a group, it moves on to the next group. For the Reunite operation, we used a minimal perfect hash function of all distinct kmer extremities in order to perform the necessary operations. The possibly high amount of key to index lead us to the conception of a new method to construct such functions that will be described in the next section. The presented algorithm takes as input a set of kmers, but in our implementation, BCALM2 is based on the GATB [84] library and make use of its kmer counting operation to handle directly read files. In this kmer counter, kmers are divided into partitions according to their minimizer, then each partition is counted independently. We modified the GATB kmer counting algorithm so that partition files correspond exactly to file groups.

2.5.3 Large genome de Bruijn graphs

We evaluated BCALM2 performances and its comparison with other tool for compacting a De Bruijn graph. We used two human datasets from GAGE [80] and two larger datasets from the spruce and pine sequencing projects [61, 85].

In Figure 2.13 we show how BCALM2 is affected by the changes in the parameters k and l. It shows that BCALM2 has almost identical running times for l from 6 to 10.

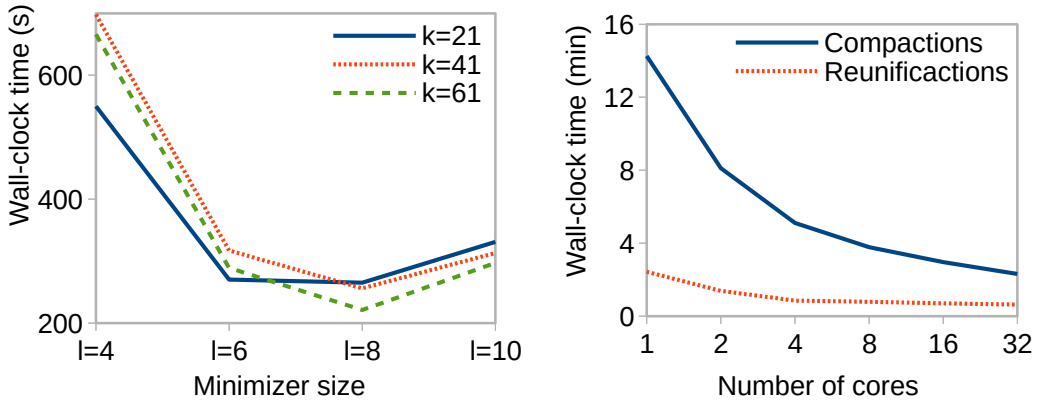

Figure 2.13: BCALM2 wall-clock running times with respect to parameters ℓ (left) and k (using 4 cores) and number of cores (using $k = 55$ and $\ell = 8$), on the chromosome 14 dataset (right).

Short minimizer sizes create fewer files and therefore limit parallel operations. We also show how BCALM2 scales with multiples cores. We observe that the compactions scale in a nearly linear manner with the number of thread used. There remains overhead related to disk operations.

We also compared the BCALM2 performances with BCALM, ABYSS and Meraculous2 graph compaction steps in Table 2.7. We observe that BCALM2 is the fastest available tool while using order of magnitude less memory but compared to BCALM. We further evaluated BCALM 2 on two very large sequencing datasets: Illumina reads from the 20 Gbp Picea glauca genome (8.5 billion reads, 152–300 bp each, 1.1 TB compressed FASTQ, SRA056234), and Illumina paired-end reads from the 22 Gbp Pinus taeda genome (9.4 billion reads, 128–154 bp each, 1.2 TB compressed FASTQ, SRX016231). The k-mer counting step took around a day and less than 40 GB of memory for each dataset.

Table 2.8 shows the performance of BCALM 2 on these two datasets, as well as unitigs statistics. Graph construction of the spruce dataset previously required 4.3 TB of memory and 2 days on a 1380-core cluster [61], while the assembly of the pine dataset previously required 800 GB of memory and 3 months on a single machine [85]. Although we used the same sequencing datasets, several parameters differ between these previous reports and our results (e.g. k value, abundance cutoff, and whether reads were error-corrected). Hence, run time, memory usage, and unitigs statistics cannot be directly compared. However, it seems reasonable to infer that BCALM 2 would remains 1–2 orders of magnitude more efficient in time and memory.

Dataset	BCALM2	BCALM	ABySS-P	Meraculous 2
Chr 14	5 mins 400 MB	15 mins 19 MB	11 mins 11 GB	62 mins 2.35 GB
Whole human	1.2 h 2.8 GB	12 h 43 MB	6.5 h 89 GB	16 h * unreported *

Table 2.7: Running times (wall-clock) and memory usage of compaction algorithms for the human datasets. For BCALM2 and BCALM we used $k = 55$, and $\ell = 8$ and $\ell = 10$, respectively; abundance cutoffs were set to 5 for Chr 14 and 3 for whole human. We used 16 cores for the parallel algorithms ABySS, Meraculous 2, and BCALM2. Meraculous 2 aborted with a validation failure due to insufficient peak k-mer depth when we ran it with abundance cutoffs of 5. We were able to execute it on chromosome 14 with a cutoff of 8, but not for the whole genome. The exact memory usage was unreported there but is less than < 1 TB. Meraculous 2 was executed with 32 prefix blocks.

Dataset	Loblolly pine	White spruce
Distinct k-mers ($\times 10^9$)	10.7	13.0
Num threads	8	16
CompactBucket() time	4 h 40 m	3 h 47 m
CompactBucket() mem	6.5 GB	6 GB
Reunite file size	85 GB	140 GB
Reunite() time	4 h 32 m	3 h 08 m
Reunite() memory	31 GB	39 GB
Total time	9 h 12 m	6 h 55 m
Total max memory	31 GB	39 GB
Unitigs ($\times 10^6$)	721	1200
Total length	32.3 Gbp	49.0 Gbp
Longest unitig	11.2 Kbp	9.0 Kbp

Table 2.8: Performance of BCALM2 on the loblolly pine and white spruce datasets. The kmer size was 31 and the abundance cutoff for kmer counting was 7.

"Efficient de Bruijn graph construction" core messages:

- Graph construction is the bottleneck of most assemblers
- Unitigs can be constructed in parallel and in low memory
- Very large genomes can be addressed without huge need of RAM

2.6 Indexing large sets

We presented the interest of working on unitigs for assembly and how to efficiently compute them. We proposed DBGFM, a data structure able to index the set of unitigs using less memory than previously best known data structures and fast enough to allow efficient assembly. Since DBGFM is more powerful than a NDS because it is a membership data structure, can we propose a better structure that is just a NDS?

We are interested in a structure using less memory and providing faster access to neighbors. We remark that the unitig sequence representation is close to the 2 bits per kmer limit. Each unitig represents an overhead of $2 * (k-1)$ nucleotides to add to the 2 bits per kmer. Raw unitig sequences is not an interesting structure in practice because a query would be linear. Indexing the last and first $k-1$mers of each unitig and associating to each $k-1$mer the unitigs that start and end with it could lead to an NDS. The cost of such a structure would be at least $8 * log(\#unitigs)$ bits per unitigs, as a $k-1$mer can be associated to up to 8 unitigs ($log(\#unitigs)$ bits are needed to encode the indice of an unitig), plus the indexing cost.

A better scheme would be to index the first and last kmer of each unitig and to associate to each indexed kmer its unique unitig. Indeed a kmer appears in at most one unitig due to the definition of the compacted de Bruijn graph. This way, an unitig can know its sons by querying the index with the four possible kmers. In order to avoid performing useless queries, we can add 8 bits to each unitig to precise which neighbors exist. This leads to a structure of at least $2 * log(\#unitigs)$ per unitig (to encode the the unitig indice twice) bits plus 8 bits from the proposed optimization plus the cost of indexing. This kind of structure would allow constant and fast queries, with a very low memory overhead. A regular hashing would be costly in memory and in query time. The best solution to keep a low memory usage would be an open addressing hash table that would offer a $2 * log(\#unitigs) * (1/loadFactor)$ bits. In practice the load factor is set below 0.8. Practically we can consider that the unitig identifier would be represented by a 32 bits integer, thus the open addressing would cost roughly one byte by unitig and would produce long queries when collisions occur. Can we do better ?

One may argue that we do not use the fact that our set of keys is static and that we could use a Minimal Perfect Hash Function (MPHF). Such structures use very few memory (as low as 3 bits per key) and guarantee constant query time and in practice very fast access (hundred of nanoseconds). Beside, many applications in bioinformatics could benefit from an efficient and lightweight structure to associate information to large sets of keys as kmers or unitigs: coverage, origin of the sequence or any properties. But MPHF are costly to construct for very large sets of keys, both in running time and in memory usage. We therefore present a technique able to efficiently construct efficient

MPHF from huge sets of keys with low memory and time.

This method is presented in the paper "Fast and scalable minimal perfect hashing construction for massive key sets" [86] included at the end of the chapter.

This paper observes that most methods to construct MPHF are designed to provide the smallest structures. However, most methods use way more memory during construction than the size of the MPHF itself. Generally, available implementations showed very high running time and memory requirement on large sets of keys, making them impracticable to index billions of objects. The paper therefore proposes a new implementation of a simple algorithm providing efficient MPHF that scales on very large sets.

The algorithm used is pretty straightforward. First keys are hashed into a bit array. Second the positions in the bit array that received an unique key are marked with a one and the corresponding keys are removed from the key set. The remaining keys are hashed in a second bit array and the process is repeated until there is not key left. At the end of the algorithm each key is associated to a unique position in one of the bit array. To find this position the key is hashed to the different bit array until a one is found.

The main interests of the method are the amount of memory needed for construction that is similar to the size of the MPHF and the parallel construction allowing a very short running time. BBHASH constructed a MPHF from a billion key dataset in a minute with 2GB of RAM were the best know MPHF library constructs it in more than 20 minutes using more than 18GB. BBHASH also achieved to construct a trillion key MPHF on a 750GB machine. This experiment is, to our knowledge, the largest MPHF constructed.

Applications BBhash MPHF has been used in several bioinformatics tools and was integrated in the GATB [84] library. It is used to associate information to kmers such as coverage or origin. The unification operation of BCALM2 also relies on BBhash MPHF structure. Another tool, called SRC [87, 88], integrated the BBhash MPHF in order to propose an efficient data structure to index reads for read set comparisons and read coverage estimation.

"Indexing large sets" core messages:

- We can build MPHF from huge sets of keys with low resources
- Such MPHF are used for assembly and also for various applications in bioinformatics

Fast and scalable minimal perfect hashing for massive key sets

Antoine Limasset[1], Guillaume Rizk[1], Rayan Chikhi[2], and Pierre Peterlongo[1]

1 **IRISA Inria Rennes Bretagne Atlantique, GenScale team, Campus de Beaulieu 35042 Rennes, France**
2 **CNRS, CRIStAL, Université de Lille, Inria Lille - Nord Europe, France**

Abstract

Minimal perfect hash functions provide space-efficient and collision-free hashing on static sets. Existing algorithms and implementations that build such functions have practical limitations on the number of input elements they can process, due to high construction time, RAM or external memory usage. We revisit a simple algorithm and show that it is highly competitive with the state of the art, especially in terms of construction time and memory usage. We provide a parallel C++ implementation called *BBhash*. It is capable of creating a minimal perfect hash function of 10^{10} elements in less than 7 minutes using 8 threads and 5 GB of memory, and the resulting function uses 3.7 bits/element. To the best of our knowledge, this is also the first implementation that has been successfully tested on an input of cardinality 10^{12}. Source code: https://github.com/rizkg/BBHash

1998 ACM Subject Classification H.3.1 E.2

Keywords and phrases Minimal Perfect Hash Functions, Algorithms, Data Structures, Big Data

Digital Object Identifier 10.4230/LIPIcs...

1 Introduction

Given a set S of N elements (*keys*), a minimal perfect hash function (`MPHF`) is an injective function that maps each key of S to an integer in the interval $[1, N]$. In other words, an `MPHF` labels each key of S with integers in a collision-free manner, using the smallest possible integer range. A remarkable property is the small space in which these functions can be stored: only a couple of bits per key, independently of the size of the keys. Furthermore, an `MPHF` query is done in constant time. While an `MPHF` could be easily obtained using a key-value store (e.g. a hash table), such a representation would occupy an unreasonable amount of space, with both the keys and the integer labels stored explicitly.

The theoretical minimum amount of space needed to represent an `MPHF` is known to be $\log_2(e)N \approx 1.44N$ bits [10, 14]. In practice, for large key sets (billions of keys), many implementations achieve less than $3N$ bits per key, regardless of the number of keys [2, 9]. However no implementation comes asymptotically close to the lower bound for large key sets. Given that `MPHF`s are typically used to index huge sets of strings, e.g. in bioinformatics [6, 7, 8], in network applications [12], or in databases [5], lowering the representation space is of interest. We observe that in many of these applications, `MPHF`s are actually used to construct static dictionaries, i.e. key-value stores where the set of keys is fixed and never updated [6, 8]. Assuming that the user only queries the `MPHF` to get values corresponding to keys that are guaranteed to be in the static set, the keys themselves do not necessarily need to be stored in memory. However the associated values in the dictionary typically do need to be stored, and

they often dwarf the size of the MPHF. The representation of such dictionaries then consists of two components: a space-efficient MPHF, and a relatively more space-expensive set of values. In such applications, whether the MPHF occupies 1.44 bits or 3 bits per key is thus arguably not a critical aspect.

In practice, a significant bottleneck for large-scale applications is the construction step of MPHFs, both in terms of memory usage and computation time. Constructing MPHFs efficiently is an active area of research. Many recent MPHF construction algorithms are based on efficient peeling of hypergraphs [1, 3, 4, 11]. However, they require an order of magnitude more space during construction than for the resulting data structure. For billions of keys, while the MPHF itself can easily fit in main memory of a commodity computer, its construction algorithm requires large-memory servers. To address this, Botelho and colleagues [4] propose to divide the problem by building many smaller MPHFs, while Belazzougui *et al.* [1] propose an external-memory algorithm for hypergraph peeling. Very recently, Genuzio *et al.* [11] demonstrated practical improvements to the Gaussian elimination technique, that make it competitive with [1] in terms of construction time, lookup time and space of the final structure. These techniques are, to the best of our knowledge, the most scalable solutions available. However, when evaluating existing implementations, the construction of MPHFs for sets that significantly exceed a billion keys remains prohibitive in terms of time and space usage.

A simple idea has been explored by previous works [6, 12, 16] for constructing PHFs (Perfect Hash Functions, non minimal) or MPHFs using arrays of bits, or fingerprints. However, it has received relatively less attention compared to other hypergraph-based methods, and no implementation is publicly available in a stand-alone MPHF library. In this article we revisit this idea, and introduce novel contributions: a careful analysis of space usage during construction, and an efficient, parallel implementation along with an extensive evaluation with respect to the state of the art. We show that it is possible to construct an MPHF using almost as little memory as the space required by the final structure, without partitioning the input. We propose a novel implementation called *BBhash* ("Basic Binary representAtion of Successive Hashing") with the following features:

- construction space overhead is small compared to the space occupied by the MPHF,
- multi-threaded,
- scales up to to very large key sets (tested with up to 1 trillion keys).

To the best of our knowledge, there does not exist another usable implementation that satisfies any two of the features above. Furthermore, the algorithm enables a time/memory trade-off: faster construction and faster query times can be obtained at the expense of a few more bits per element in the final structure and during construction. We created an MPHF for ten billion keys in 6 minutes 47 seconds and less than 5 GB of working memory, and an MPHF for a trillion keys in less than 36 hours and 637 GB memory. Overall, with respect to other available MPHF construction approaches, our implementation is at least two orders of magnitudes more space-efficient when considering internal and external memory usage during construction, and at least one order of magnitude faster. The resulting MPHF has slightly higher space usage and faster or comparable query times than other methods.

2 Efficient construction of minimal perfect hash function

2.1 Method overview

Our MPHF construction procedure revisits previously published techniques [6, 12]. Given a set F_0 of keys, a classical hash function h_0 maps keys to an integer in $[1, |F_0|]$. A bit array A_0 of

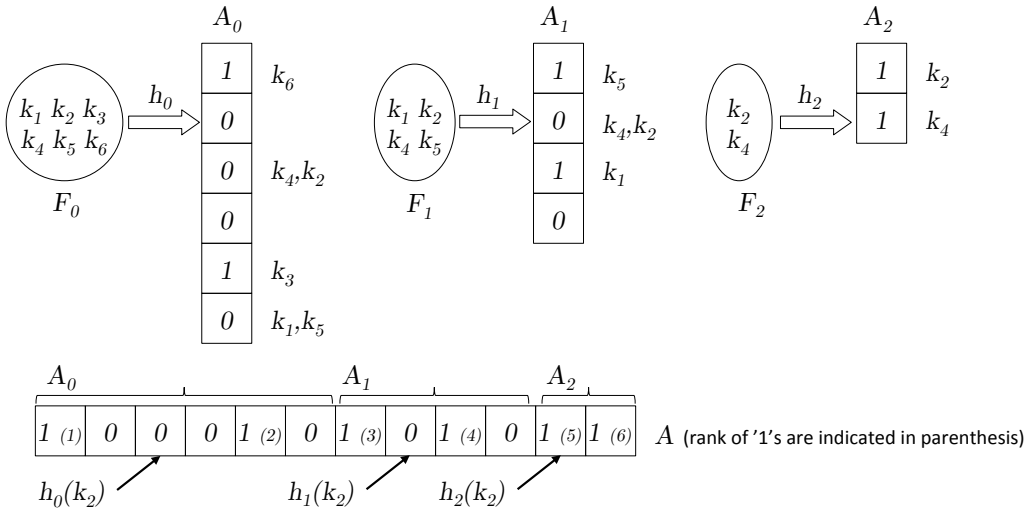

Figure 1 `MPHF` construction and query example. The input is a set F_0 composed of $N = 6$ keys (k_1 to k_6). All keys are hashed using a hash function h_0 and are attempted to be placed in an array A_0 at positions given by the hash function. The keys k_3 and k_6 do not have collisions in the array, thus the corresponding bits in A_0 are set to '1'. The other keys from F_0 that are involved in collisions are placed in a new set F_1. In the second level, keys from F_1 are hashed using a hash function h_1. Keys k_1 and k_5 are uniquely placed while k_2 and k_4 collide, thus they are then stored in the set F_2. With the hash function h_2, the keys from F_2 have no collision, and the process finishes. The `MPHF` query operation is very similar to the construction algorithm. Let A be the the concatenation of A_0, A_1, A_2 (see bottom part of the figure). To query k_2, the key is first hashed with h_0. The associated value in A_0 is '0', so k_2 is then hashed with h_1. The value associated in A_1 is again '0'. When finally hashed with h_2, the value associated in A_2 is '1' and thus the query stops here. The index returned by the `MPHF` is the rank of this '1' (here, 5) in A. In this example, the `MPHF` values returned when querying k_1, k_2, k_3, k_4, k_5 and k_6 are respectively 4,5,2,6,3, and 1.

size $|F_0|$ is created such that there is a 1 at position i if and only if exactly one element of F_0 has a hash value of i. We say that there is a *collision* whenever two keys in F_0 have the same hash value. Keys from F_0 that were involved in a collision are inserted into a new set F_1. The process repeats with F_1 and a new hash function h_1. A new bit array A_1 of size $|F_1|$ is created using the same procedure as for A_0 (except that F_1 is used instead of F_0, and h_1 instead of h_0). The process is repeated with $F_2, F_3, \ldots$ until one of these sets, $F_{\text{last}+1}$, is empty.

We obtain an `MPHF` by concatenating the bit arrays $A_0, A_1, \ldots, A_{\text{last}}$ into an array A. To perform a query, a key is hashed successively with hash functions $h_0, h_1, \ldots$ as long as the value in A_i ($i \geq 0$) at the position given by the hash function h_i is 0. Eventually, by construction, we reach a 1 at some position of A for some $i = d$. We say that the *level* of the key is d. The index returned by the `MPHF` is the rank of this one in A. See Figure 1 for an example.

2.2 Algorithm details

2.2.1 Collision detection

During construction at each level d, collisions are detected using a temporary bit array C_d of size $|A_d|$. Initially all C_d bits are set to '0'. A bit of $C_d[i]$ is set to '1' if two or more keys from

F_d have the same value i given by hash function h_d. Finally, if $C_d[i] = 1$, then $A_d[i] = 0$. Formally:

$$C_d[i] = 1 \Rightarrow A_d[i] = 0;$$
$$(h_d[x] = i \text{ and } A_d[i] = 0 \text{ and } C_d[i] = 0) \Rightarrow A_d[i] = 1 \,(\text{and } C_d[i] = 0)\,;$$
$$(h_d[x] = i \text{ and } A_d[i] = 1 \text{ and } C_d[i] = 0) \Rightarrow A_d[i] = 0 \text{ and } C_d[i] = 1.$$

2.2.2 Queries

A query of a key x is performed by finding the smallest d such that $A_d[h_d(x)] = 1$. The (non minimal) hash value of x is then $(\sum_{i<d} |F_i|) + h_d(x)$.

2.2.3 Minimality

To ensure that the image range of the function is $[1, |F_0|]$, we compute the cumulative rank of each '1' in the bit arrays A_i. Suppose, that d is the smallest value such that $A_d[h_d(x)] = 1$. The minimal perfect hash value is given by $\sum_{i<d}(weight(A_i) + rank(A_d[h_d(x)])$, where $weight(A_i)$ is the number of bits set to '1' in the A_i array, and $rank(A_d[y])$ is the number of bits set to 1 in A_d within the interval $[0, y]$, thus $rank(A_d[y]) = \sum_{j<y} A_d[j]$. This is a classic method also used in other MPHFs [3].

2.2.4 Faster query and construction times (parameter γ)

The running time of the construction depends on the number of collisions on the A_d arrays, at each level d. One way to reduce the number of collisions, hence to place more keys at each level, is to use bit arrays (A_d and C_d) larger than $|F_d|$. We introduce a parameter $\gamma \in \mathbb{R}$, $\gamma \geq 1$, such that $|C_d| = |A_d| = \gamma|F_d|$. With $\gamma = 1$, the size of A is minimal. With $\gamma \geq 2$, the number of collisions is significantly decreased and thus construction and query times are reduced, at the cost of a larger `MPHF` structure size. The influence of γ is discussed in more detail in the following analyses and results.

2.3 Analysis

Proofs of the following observations and lemma are given in the Appendix.

2.3.1 Size of the `MPHF`

The expected size of the structure can be determined using a simple argument, previously made in [6]. When $\gamma = 1$, the expected number of keys which do not collide at level d is $|A_d|e^{-1}$, thus $|A_d| = |A_{d-1}|(1 - e^{-1}) = |A_0|(1 - e^{-1})^d$. In total, the expected number of bits required by the hashing scheme is $\sum_{d\geq 0} |A_d| = N \sum_{d\geq 0}(1 - e^{-1})^d = eN$, with N being the total number of input keys ($N = |F_0|$). Note that consequently the image of the hash function is also in $[1, eN]$, before minimization using the rank technique. When $\gamma \geq 1$, the expected proportion of keys without collisions at each level d is $|A_d|e^{-\frac{1}{\gamma}}$. Since each A_d no longer uses one bit per key but γ bits per key, the expected total number of bits required by the `MPHF` is $\gamma e^{\frac{1}{\gamma}} N$.

2.3.2 Space usage during construction

We analyze the disk space used during construction. Recall that during construction of level d, a bit array C_d of size $|A_d|$ is used to record collisions. Note that the C_d array is only

needed during the d-th level. It is deleted before level $d+1$. The total memory required during level d is $\sum_{i\leq d}(|A_i|)+|C_d| = \sum_{i<d}(|A_i|)+2|A_d|$.

▶ Lemma 1. For $\gamma > 0$, the space of our MPHF is $S = \gamma e^{\frac{1}{\gamma}} N$ bits. The maximal space during construction is S when $\gamma \leq \log(2)^{-1}$, and $2S$ bits otherwise.

A full proof of the Lemma is provided in the Appendix.

3 Implementation

We present *BBhash*, a C++ implementation available at http://github.com/rizkg/BBHash. We describe in this section some design key choices and optimizations.

3.1 Rank structure

We use a classical technique to implement the rank operation: the ranks of a fraction of the '1's present in A are recorded, and the ranks in-between are computed dynamically using the recorded ranks as checkpoints.

In practice 64 bit integers are used for counters, which is enough for realistic use of an MPHF, and placed every 512 positions by default. These values were chosen as they offer a good speed/memory trade-off, increasing the size of the MPHF by a factor 1.125 while achieving good query performance. The total size of the MPHF is thus $(1+\frac{64}{512})\gamma e^{\frac{1}{\gamma}} N$.

3.2 Parallelization

Parallelization is achieved by partitioning keys over several threads. The algorithm presented in Section 2 is executed on multiple threads concurrently, over the same memory space. Built-in compiler functions (e.g. *sync_fetch_and_or*) are used for concurrent access in the A_i arrays. The efficiency of this parallelization scheme is shown in the Results section, but note that it is fundamentally limited by random memory accesses to the A_i arrays which incur cache misses.

3.3 Hash functions

The MPHF construction requires classical hash functions. Other authors have observed that common hash functions behave practically as well as fully random hash functions [2]. We therefore choose to use xor-shift based hash functions [13] for their efficiency both in terms of computation speed and distribution uniformity [15].

3.4 Disk usage

In the applications we consider, key sets are typically too big to fit in RAM. Thus we propose to read them on the fly from disk. There are mainly two distinct strategies regarding the disk usage during construction: 1/ during each level d, keys that are to be inserted in the set F_{d+1} are written directly to disk. The set F_{d+1} is then read during level $d+1$ and erased before level $d+2$; or 2/ at each level all keys from the original input key file are read and queried in order to determine which keys were already assigned to a level $i<d$, and which would belong to F_d. When the key set becomes small enough (below user-defined threshold) it is loaded in ram to avoid costly re-computation from scratch at each level.

The first strategy obviously provides faster construction at the cost of temporary disk usage. At each level $d>0$, two temporary key files are stored on disk: F_d and F_{d+1}. The highest disk

usage is thus achieved during level 1, i.e. by storing $|F_1|+|F_2| = |F_0|((1-e^{-1/\gamma})+(1-e^{-1/\gamma})^2)$ elements. With $\gamma = 1$, this represents $\approx 1.03N$ elements, thus the construction overhead on disk is approximately the size of the input key file. Note that with $\gamma = 2$ (resp. $\gamma = 5$), this overhead diminishes and becomes a ratio of ≈ 0.55 (resp. ≈ 0.21) the size of the input key file.

The first strategy is the default strategy proposed in our implementation. The second one has also been implemented and can be optionally switched on.

3.5 Termination

The expected number of unplaced keys decreases exponentially with the number of levels but is not theoretically guaranteed to reach zero in a finite number of steps. To ensure termination of the construction algorithm, in our implementation a maximal number D of levels is fixed. Then, the remaining keys are inserted into a regular hash table. Value D is a parameter, its default value is $D = 25$ for which the expected number of keys stored in this hash table is $\approx 10^{-5}N$ for $\gamma = 1$ and becomes in practice negligible for $\gamma \geq 2$, allowing the size overhead of the final hash table to be negligible regarding the final `MPHF` size.

4 Results

We evaluated the performance *BBhash* for the construction of large `MPHF`s. We generated files containing various numbers of keys (from 1 million to 1 trillion keys). In our tests, a key is a binary representation of a pseudo-random positive integer in $[0; 2^{64}]$. Within each file, each key is unique. We also performed a test where input keys are strings (n-grams) to ensure that using integers as keys does not bias results. Tests were performed on a cluster node with a Intel© Xeon© CPU E5-2660 v3 2.60GH 20-core CPU, 256 GB of memory, and a mechanical hard drive. Except for the experiment with 10^{12} keys, running times include the time needed to read input keys from disk. Note that files containing key sets may be cached in memory by the operating system, and all evaluated methods benefit from this effect during `MPHF` construction. We refer to the Appendix for the specific commands and parameters used in these experiments.

We first analyzed the influence of the γ value (the main parameter of *BBhash*), then the effect of using multiple threads depending on the parallelization strategy. Second, we compared *BBhash* with other state-of-the-art methods. Finally, we performed an `MPHF` construction on 10^{12} elements.

4.1 Influence of the γ parameter

We report in Figure 2 (left) the construction times and the mean query times, as well as the size of the produced `MPHF`, with respect to several γ values. The main observation is that $\gamma \geq 2$ drastically accelerates construction and query times. This is expected since large γ values allow more elements to be placed in the first levels of the `MPHF`; thus limiting the number of times each key is hashed to compute its level. In particular, for keys placed in the very first level, the query time is limited to a single hashing and a memory access. The average level of all keys is $e^{(1/\gamma)}$, we therefore expect construction and query times to decrease when γ increases. However, larger γ values also incur larger `MPHF` sizes. One observes that $\gamma > 5$ values seem to bring very little advantage at the price of higher space requirements. A related work used $\gamma = 1$ in order to minimize the `MPHF` size [6]. Here, we

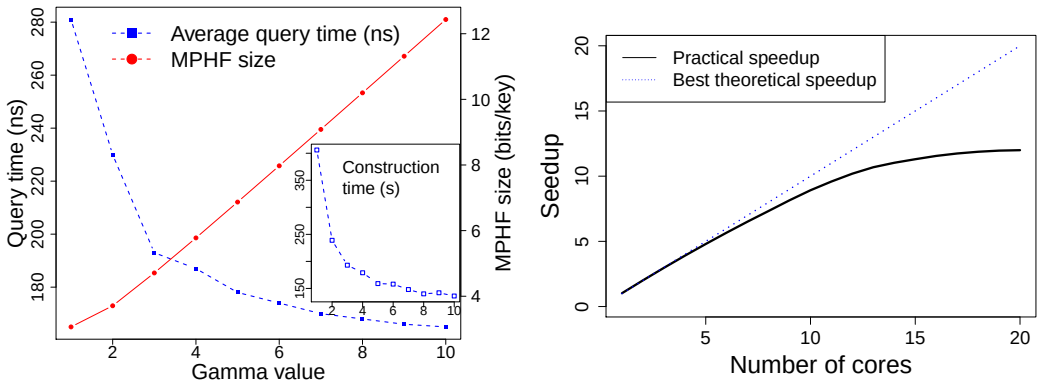

Figure 2 Left: Effects of the gamma parameter on the performance of *BBhash* when run on a set composed of one billion keys, when executed on a single CPU thread. Times and `MPHF` size behave accordingly to the theoretical analysis, respectively $O(e^{(1/\gamma)})$, and $O(\gamma e^{(1/\gamma)})$. Right: Performance of the *BBhash* construction time according to the number of cores, using $\gamma = 2$.

argue that using γ values larger than 1 has significant practical merits. In our tests, we often used $\gamma = 2$ as it yields an attractive time/space trade-off during construction and queries.

4.2 Parallelization performance

We evaluated the capability of our implementation to make use of multiple CPU cores. In Figure 2 (right), we report the construction times with respect to the number of threads. We observe a near-ideal speed-up with respect to the number of threads with diminishing returns when using more than 10 threads, which is likely due to cache misses that induce a memory access bottleneck.

In addition to these results, we applied *BBhash* on a key set of 10 billion keys and on a key set of 100 billion keys, again using default parameters and 8 threads. The memory usage was respectively 4.96GB and 49.49GB, and the construction time was respectively 462 seconds and 8913 seconds, showing the scalability of *BBhash*.

4.3 Comparisons with state of the art methods

We compared *BBhash* with state-of-the-art `MPHF` methods. CHD (http://cmph.sourceforge.net/) is an implementations of the compressed hash-and-displace algorithm [2]. EMPHF [1] is based on random hypergraph peeling, and the HEM [4] implementation in EMPHF is based on partitioning the input data. Sux4J is a Java implementation of [11]. We did not include other methods cited earlier because they do not provide an implementation [12, 16] or the software integrates a non-minimal perfect hash function that is not stand-alone [6]. However single-threaded results presented in [16] show that construction times and `MPHF` sizes are comparable to ours, query times are significantly longer, and no indication is provided about the memory usage during construction. Our benchmark code is available at https://github.com/rchikhi/benchmphf.

Figure 3 shows that all evaluated methods are able to construct `MPHF`s that contain a billion elements, but only *BBhash* scales up to datasets that contain 10^{11} elements and more. Overall, *BBhash* shows consistently better time and memory usage during construction.

We additionally compared the resulting `MPHF` size, i.e. the space of the data structure

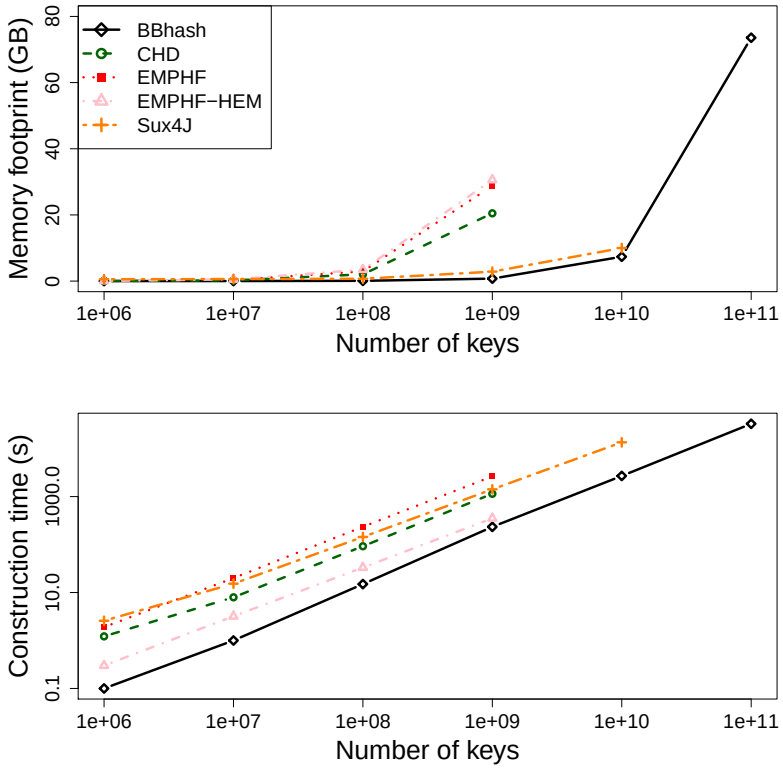

Figure 3 Memory footprint and construction time with respect to the number of keys. All libraries were run using default parameters, including $\gamma = 2$ for *BBhash*. For a fair comparison, *BBhash* was executed on a single CPU thread. Except for Sux4J, missing data points correspond to runs that exceeded the amount of available RAM. Sux4J limit comes from the disk usage, estimated at approximately 4TB for 10^{11} keys.

returned by the construction algorithm, and the mean query time across all libraries on a dataset consisting of a billion keys (Table 1). `MPHF`s produced by *BBhash* range from 2.89 bits/key (when $\gamma = 1$ and ranks are sampled every 1024 positions) to 6.9 bits/key (when $\gamma = 5$ and a rank sampling of 512). The 0-0.8 bits/key size difference between our implementation and the theoretical space usage of the *BBhash* structure size is due to additional space used by the rank structure. We believe that a reasonable compromise in terms of query time and structure size is 3.7 bits/key with $\gamma = 2$ and a rank sampling of 512, which is marginally larger than the `MPHF` sizes of other libraries (ranging from 2.6 to 3.5 bits/key). As we argued in the Introduction, using one more bit per key seems to be a reasonable trade-off for performance.

Construction times vary by one or two orders of magnitude across methods, *BBhash* being the fastest. With default parameters ($\gamma = 2$, rank sampling of 512), *BBhash* has a construction memory footprint 40× to 60× smaller than other libraries except for Sux4j, for which *BBhash* remains 4× smaller. Query times are roughly within an order of magnitude ($179 - 1037$ ns) of each other across methods, with a slight advantage for *BBhash* when $\gamma \geq 2$. Sux4j achieves an attractive balance with low construction memory and query times, but high disk usage. In our tests, the high disk usage of Sux4j was a limiting factor for the construction of very large `MPHF`s.

Method	Query time (ns)	MPHF size (bits/key)	Const. time* (s)	Const. memory**	Disk. usage (GB)
BBhash $\gamma = 1$	271	3.1	60 (393)	3.2 (376)	8.23
BBhash $\gamma = 1$ minirank	279	2.9	61(401)	3.2 (376)	8.23
BBhash $\gamma = 2$	216	3.7	35 (229)	4.3 (516)	4.45
BBhash $\gamma = 2$ nodisk	216	3.7	80 (549)	6.2 (743)	0
BBhash $\gamma = 5$	179	6.9	25 (162)	10.7 (1,276)	1.52
EMPHF	246	2.9	2,642	247.1 (29,461)†	20.8
EMPHF HEM	581	3.5	489	258.4 (30,798)†	22.5
CHD	1037	2.6	1,146	176.0 (20,982)	0
Sux4J	252	3.3	1,418	18.10 (2,158)	40.1

Table 1 Performance of different MPHF algorithms applied on a key set composed of 10^9 64-bits random integers, of size 8GB. Each time result is the average value over three tests. The 'nodisk' row implements the second strategy described in Section 3.4, and the 'minirank' row samples ranks every 1024 positions instead of 512 by default. *The column "*Const. time*" indicates the construction time in seconds. In the case of *BBhash*, the first value is the construction time using eight CPU threads and the second value in parenthesis is the one using one CPU thread. **The column "*Const. memory*" indicates the RAM used during the MPHF construction, in bits/key and the total in MB in parenthesis. † The memory usages of EMPHF and EMPHF HEM reflect the use of memory-mapped files (mmap scheme).

Note that EMPHF, EMPHF HEM and Sux4j implement a disk partitioning strategy, that could in principle also be applied to others methods, including ours. Instead of creating a single large MPHF, they partition the set of input keys on disk and construct many small MPHFs independently. In theory this technique allows to engineer the MPHF construction algorithm to use parallelism and lower memory, at the expense of higher disk usage. In practice we observe that the existing implementations that use this technique are not parallelized. While EMPHF and EMPHF HEM used relatively high memory in our tests (around 30 GB for 1 billion elements) due to memory-mapped files, they also completed the construction successfully on another machine that had 16 GB of available memory. However, we observed what appears to be limitations in the scalability of the scheme: we were unable to run EMPHF and EMPHF HEM on an input of 10 billion elements using 256 GB of memory. Regardless, we view this partitioning technique as promising but orthogonal to the design of efficient "monolithic" MPHFs constructions such as *BBhash*.

4.4 Performance on an actual dataset

In order to ensure that using pseudo-random integers as keys does not bias results, we ran *BBhash* using strings as keys. We used n-grams extracted from the Google Books Ngram dataset[1], version 20120701. On average the n-gram size is 18. We also generated random words of size 18. As reported in Table 2, we obtained highly similar results to those obtained with random integer keys.

[1] http://storage.googleapis.com/books/ngrams/books/datasetsv2.html

Dataset	Query time (ns)	MPHF size (bits/key)	Const. time (s)
10^8 Random strings	325	3.7	35
10^8 Ngrams	296	3.7	37

Table 2 Performance of *BBhash* ($\gamma = 2$, 8 threads) when using ASCII strings as keys.

4.5 Indexing a trillion keys

We performed a very large-scale test by creating an MPHF for 10^{12} keys. For this experiment, we used a machine with 750 GB of RAM. Since storing that many keys would require 8 TB of disk space, we instead used a procedure that deterministically generates a stream of 10^{12} pseudo-random integers in $[0, 2^{64} - 1]$. We considered the streamed values as input keys without writing them to disk. In addition, key sets of cardinality below 20 billion (2% of the input) were stored in memory to avoid re-computation from scratch at each subsequent level. Thus, the reported computation time should not be compared to previously presented results as this experiment has no disk accesses. The test was performed using $\gamma = 2$, 24 threads.

Creating the MPHF took 35.4 hours and required 637 GB RAM. This memory footprint is roughly separated between the bit arrays ($\approx$ 459 GB) and the memory required for loading 20 billion keys in memory ($\approx$ 178 GB). The final MPHF occupied 3.71 bits per key.

5 Conclusion

We have proposed a resource-efficient and highly scalable algorithm for constructing and querying MPHFs. Our algorithmic choices were motivated by simplicity: the method only relies on bit arrays and classical hash functions. While the idea of recording collisions in bit arrays to create MPHFs is not novel [6, 12], to the best of our knowledge *BBhash* is the first implementation that is competitive with the state of the art. The construction is particularly time-efficient as it is parallelized and mainly consists in hashing keys and performing memory accesses. Moreover, the additional data structures used during construction are provably small enough to ensure a low memory overhead during construction. In other words, creating the MPHF does not require much more space than the resulting MPHF itself. This aspect is important when constructing MPHFs on large key sets in practice.

Experimental results show that *BBhash* generates MPHFs that are slightly larger to those produced by other methods. However *BBhash* is by far the most efficient in terms of construction time, query time, memory and disk footprint for indexing large key sets (of cardinality above 10^9 keys). The scalability of our approach was confirmed by constructing MPHFs for sets as large as 10^{12} keys. To the best of our knowledge, no other MPHF implementation has been tested on that many keys.

A time/space trade-off is achieved through the γ parameter. The value $\gamma = 1$ yields MPHFs that occupy roughly $3N$ bits of space and have little memory overhead during construction. Higher γ values use more space for the construction and the final structure size, but they achieve faster construction and query times. Our results suggest that $\gamma = 2$ is a good time-versus-space compromise, using 3.7 bits per key. With respect to hypergraph-based methods [1, 3, 4, 11], *BBhash* offers significantly better construction performance, but the resulting MPHF size is up to 1 bit/key larger. We however argue that the MPHF size, as long as it is limited to a few bits per key, is generally not a bottleneck as many applications use MPHFs to associate much larger values to keys. Thus, we believe that this work will unlock many high performance computing applications where the possibility to index billions keys

and more is a huge step forward.

An interesting direction for future work is to obtain more space-efficient MPHFs using our method. We believe that a way to achieve this goal is to slightly change the hashing scheme. We would like to explore an idea inspired by the CHD algorithm for testing several hash functions at each level and selecting (then storing) one that minimizes the number of collisions. At the price of longer construction times, we anticipate that this approach could significantly decrease the final structure size.

Acknowledgments

This work was funded by French ANR-12-BS02-0008 Colib'read project. We thank the GenOuest BioInformatics Platform that provided the computing resources necessary for benchmarking. We thank Djamal Belazzougui for helpful discussions and pointers.

References

1 Djamal Belazzougui, Paolo Boldi, Giuseppe Ottaviano, Rossano Venturini, and Sebastiano Vigna. Cache-oblivious peeling of random hypergraphs. In *Data Compression Conference (DCC), 2014*, pages 352–361. IEEE, 2014.

2 Djamal Belazzougui, Fabiano C Botelho, and Martin Dietzfelbinger. Hash, displace, and compress. In *European Symposium on Algorithms*, pages 682–693. Springer, 2009.

3 Fabiano C Botelho, Rasmus Pagh, and Nivio Ziviani. Simple and space-efficient minimal perfect hash functions. In *Algorithms and Data Structures*, pages 139–150. Springer, 2007.

4 Fabiano C Botelho, Rasmus Pagh, and Nivio Ziviani. Practical perfect hashing in nearly optimal space. *Information Systems*, 38(1):108–131, 2013.

5 Chin-Chen Chang and Chih-Yang Lin. Perfect hashing schemes for mining association rules. 48(2):168–179, 2005. doi:10.1093/comjnl/bxh074.

6 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.

7 Yupeng Chen, Bertil Schmidt, and Douglas L Maskell. A hybrid short read mapping accelerator. *BMC Bioinformatics*, (1):67. doi:10.1186/1471-2105-14-67.

8 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.

9 Zbigniew J Czech, George Havas, and Bohdan S Majewski. Perfect hashing. *Theoretical Computer Science*, 182(1):1–143, 1997.

10 Michael L Fredman and János Komlós. On the size of separating systems and families of perfect hash functions. *SIAM Journal on Algebraic Discrete Methods*, 5(1):61–68, 1984.

11 Marco Genuzio, Giuseppe Ottaviano, and Sebastiano Vigna. Fast scalable construction of (minimal perfect hash) functions. In V. Andrew Goldberg and S. Alexander Kulikov, editors, *Experimental Algorithms: 15th International Symposium, SEA 2016, St. Petersburg, Russia, June 5-8, 2016, Proceedings*, pages 339–352. Springer International Publishing, Cham, 2016. doi:10.1007/978-3-319-38851-9_23.

12 Yi Lu, Balaji Prabhakar, and Flavio Bonomi. Perfect hashing for network applications. In *2006 IEEE International Symposium on Information Theory*, pages 2774–2778. IEEE, 2006.

13 George Marsaglia et al. Xorshift rngs. *Journal of Statistical Software*, 8(14):1–6, 2003.

14 Kurt Mehlhorn. On the program size of perfect and universal hash functions. In *Foundations of Computer Science, 1982. SFCS'08. 23rd Annual Symposium on*, pages 170–175. IEEE, 1982.

15 Michael Mitzenmacher and Salil Vadhan. Why simple hash functions work: exploiting the entropy in a data stream. In *Proceedings of the nineteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 746–755. Society for Industrial and Applied Mathematics, 2008.

16 Ingo Müller, Peter Sanders, Robert Schulze, and Wei Zhou. *Retrieval and Perfect Hashing Using Fingerprinting*, pages 138–149. Springer International Publishing, Cham, 2014. doi:10.1007/978-3-319-07959-2_12.

Appendix

Proofs of MPHF size and memory required for construction

MPHF size with $\gamma = 1$.

$$\begin{aligned}\sum_{d\geq 0}|A_d| &= N\sum_{d\geq 0}(1-e^{-1})^d \\ &= N\frac{1}{1-(1-e^{-1})} && \text{as } \lim_{d\to+\infty}(1-e^{-1})^d = 0 \\ &= eN\end{aligned}$$

◀

MPHF size using any $\gamma \geq 1$. With $\gamma \geq 1 : |A_d| = \gamma|A_{d-1}|(1-e^{\frac{-1}{\gamma}}) = \gamma|A_0|(1-e^{\frac{-1}{\gamma}})^d = \gamma N(1-e^{\frac{-1}{\gamma}})^d$

$$\text{Thus, } \sum_{d\geq 0}|A_d| = \gamma N\sum_{d\geq 0}(1-e^{\frac{-1}{\gamma}})^d$$

Moreover, as $\lim_{d\to+\infty}(1-e^{\frac{-1}{\gamma}})^d = 0$ since for $\gamma > 0, 0 < 1-e^{\frac{-1}{\gamma}} < 1$, on has:

$$\sum_{d\geq 0}|A_d| = \gamma N\frac{1}{1-(1-e^{\frac{-1}{\gamma}})} = \gamma e^{\frac{1}{\gamma}}N$$

◀

Note that this proof stands for any γ value > 0, but that with $\gamma < 1$ the theoretical and practical MPHF sizes increase exponentially as γ get close to zero.

Lemma 1. Let $m(d)$ be memory required during level d and let R be the ratio between the maximal memory needed during the MPHF construction and the MPHF total size denoted by S. Formally,

$$R = \frac{\max_{d\geq 0}(m(d))}{S} = \frac{\max_{d\geq 0}(m(d))}{\gamma e^{\frac{1}{\gamma}}N}$$

First we prove that $\lim_{d\to\infty}\frac{m(d)}{S} = 1$.

$$m(d) = \sum_{i<d}|A_i| + 2|A_d| = \gamma N\left(\frac{1-(1-e^{\frac{-1}{\gamma}})^d}{e^{\frac{-1}{\gamma}}} + 2(1-e^{\frac{-1}{\gamma}})^d\right)$$

Since for $\gamma > 0$, $0 < 1-e^{\frac{-1}{\gamma}} < 1$, then $\lim_{d\to\infty} m(d) = \gamma e^{\frac{1}{\gamma}}N$. Thus $\lim_{d\to\infty}\frac{m(d)}{S} = 1$. Before going further, we need to compute $m(d+1) - m(d)$:

$$\begin{aligned}m(d+1) - m(d) &= \sum_{i<d+1}|A_i| + 2|A_{d+1}| - \sum_{i<d}|A_i| + 2|A_d| \\ &= |A_d| + 2|A_{d+1}| - 2|A_d| = 2|A_{d+1}| - |A_d| \\ &= 2\gamma N(1-e^{\frac{-1}{\gamma}})^{d+1} - \gamma N(1-e^{\frac{-1}{\gamma}})^d \\ &= \gamma N(1-e^{\frac{-1}{\gamma}})^d(2(1-e^{\frac{-1}{\gamma}}) - 1) \\ &= \gamma N(1-e^{\frac{-1}{\gamma}})^d(1-2e^{\frac{-1}{\gamma}})\end{aligned}$$

We now prove $R \leq 1$ when $\gamma \leq \frac{1}{\log(2)}$ and also, $R < 2$ when $\gamma > \frac{1}{\log(2)}$.

- Case 1: $\gamma \leq \frac{1}{\log(2)}$
  We have $\frac{m(0)}{S} = 2e^{-\frac{1}{\gamma}} \leq 2e^{-\log(2)} = 1$.
  Moreover, as $m(d+1) - m(d) = \gamma N(1 - e^{\frac{-1}{\gamma}})^d(1 - 2e^{\frac{-1}{\gamma}})$ and as, with $\gamma \leq \frac{1}{\log(2)}$: $1 - e^{\frac{-1}{\gamma}} \geq 0.5$, and $1 - 2e^{\frac{-1}{\gamma}} \geq 0$ then $m(d+1) - m(d) \geq 0$, thus, m is an increasing function.
  To sum up, with $\gamma \leq \frac{1}{\log(2)}$, we have **1/** that $\frac{m(0)}{S} \leq 1$, **2/** that $\lim_{d\to\infty} \frac{m(d)}{S} = 1$, and **3/** that m is increasing, then $R \leq 1$.
- Case 2: $\gamma > \frac{1}{\log(2)}$ We have $\frac{m(0)}{S} = 2e^{-\frac{1}{\gamma}}$. With $\gamma > \frac{1}{\log(2)}$, $1 < \frac{m(0)}{S} < 2$. Moreover, $m(d+1) - m(d) = \gamma N(1 - e^{\frac{-1}{\gamma}})^d(1 - 2e^{\frac{-1}{\gamma}})$ is negative as: $1 - e^{\frac{-1}{\gamma}} > 0$ and $1 - 2e^{\frac{-1}{\gamma}} < 0$ for $\gamma > \frac{1}{\log(2)}$. Thus m is a decreasing function with d.
  With $\gamma > \frac{1}{\log(2)}$, we have **1/** that $\frac{m(0)}{S} < 2$, **/2** that $\lim_{d\to\infty} \frac{m(d)}{S} = 1$ and **/3** that m is decreasing. Thus $R < 2$.

◀

Algorithms pseudo-codes

Algorithm 1: `MPHF` construction.

Data: F_0 a set of N keys, integers γ and $last$
Result: array of bit arrays $\{A_0, A_1, \ldots, A_{last}\}$, hash table H
i=0;
while F_i *not empty and* $i \leq last$ **do**
  $A_i = ArrayFill(F_i, \gamma)$;
  foreach *key* x *of* F_i **do**
    $h = hash(x) \mod (\gamma * N)$;
    if $A_i[h] == 0$ **then**
      $F_{i+1}.add(x)$
  i=i+1;
Construct H using remaining elements from F_{last+1};
Return $\{A_0, A_1, \ldots, A_{last}, H\}$

In practice F_i with $i > 1$ are stored on disk (see Section 3.4). The hash table H ensures that elements in F_{last+1} are mapped without collisions to integers in $[|F_0| - |F_{last+1}| + 1, |F_0|]$

Algorithm 2: *ArrayFill*

Data: F array of N keys, integer γ
Result: bit array A
Zero-initialize A and C two bit arrays with $\gamma * N$ elements;
foreach *key* x *of* F **do**
  $h = hash(x) \mod (\gamma * N)$;
  if $A[h] == 0$ *and* $C[h] == 0$ **then**
    $A[h] = 1$;
  if $A[h] == 1$ *and* $C[h] == 0$ **then**
    $A[h] = 0$;
    $C[h] = 1$;
  if $A[h] == 0$ *and* $C[h] == 1$ **then**
    Skip;
Delete C;
Return A;

Note that the case $A[h] == 1$ and $C[h] == 1$ never happens.

Algorithm 3: `MPHF` query

Data: bit arrays $\{A_0, A_1, \ldots, A_{\text{last}}\}$, hash table H, key x
Result: integer index of x
i=0;
while $i \leq last$ **do**
  $h = hash_i(x) \mod A_i.size()$;
  if $A_i[h] == 1$ **then**
    return $\sum_{j<i} |A_j| + rank(A_i[h])$;
  $i = i + 1$;
return *H[x]* ;

Note, when x is not an element from the key set of the `MPHF`, the algorithm may return a wrong integer index.

Commands

In this section we describe used commands for each presented result. Time and memory usages where computed using "*/usr/bin/time –verbatim*" unix command. The disk usage was computed thanks to a home made script measuring each 1/10 second the size of the directory using the "*du -sk*" unix command, and recording the highest value. The *BBhash* library and its *Bootest* tool are available from https://github.com/rizkg/BBHash.

Commands used for Section 4.1:

```
for ((gamma=1;gamma<11;gamma++)); do
./Bootest 1000000000 1 ${gamma} -bench
done
```

Note that 1000000000 is the number of keys tested and 1 is the number of used cores.

Additional tests, with larger key set and 8 threads:

```
for ((gamma=1;gamma<11;gamma++)); do
./Bootest 1000000000 1 ${gamma} -bench
done
```

Commands used for Section 4.2:

```
for keys in 10000000000 100000000000; do
./Bootest ${keys} 8 2 -bench
done
```

Commands used for Section 4.3:

We remind that our benchmark code, testing EMPHF, EMPHF MEM, CHD, and Sux4J is available at https://github.com/rchikhi/benchmphf.

- *BBhash* commands:
  ```
  for keys in 1000000 10000000 100000000 10000000000\
   10000000000 100000000000; do
  ./Bootest ${keys} 1 2 -bench
  done
  ```
- *BBhash* command with nodisk (Table 1) was
  `./Bootest 1000000000 1 2 -bench -nodisk`
  and
  `./Bootest 1000000000 8 2 -bench -nodisk`
  respectively for one and height threads. Other commands from Table 1 were deduced from previously presented *BBhash* computations.
- Commands EMPHF & EMPHF HEM:
  ```
  for keys in 1000000 10000000 100000000 10000000000\
   10000000000 100000000000; do
  ./benchmphf ${keys} -emphf
  done
  ```
  EMPHF (resp. EMPHF HEM) is tested by using the #define EMPHF_SCAN macro (resp. #define EMPHF_HEM). In order to assess the disk size footprint, the line "*unlink(tmpl);*" from file "*emphf/mmap_memory_model.hpp*" was commented.
- Commands CHD:

```
for keys in 1000000 10000000 100000000 10000000000\
 10000000000 100000000000; do
./benchmphf ${keys} -chd
done
```

- Commands Sux4J:
  for each size, the *"Sux4J/slow/it/unimi/dsi/sux4j/mph/LargeLongCollection.java"* was modified indicating the used size.
  `./run-sux4j-mphf.sh`

Commands used for Section 4.4:

As explained Section 4.4, the keyString.txt file is composed of n-grams extracted from the Google Books Ngram dataset[2], version 20120701.
`./BootestFile keyStrings.txt 10 2`

Commands used for Section 4.5:

BBhash command for indexing a trillion keys, with keys generated on the fly.
`./Bootest 1000000000000 24 2 -onthefly`

[2] http://storage.googleapis.com/books/ngrams/books/datasetsv2.html

Chapter 3

The de Bruijn graph as a reference

In this chapter we are interested in considering the de Bruijn graph as a reference. We first describe the different structures used to represent a genome and argue why a de Bruijn graph could be an interesting object to do so (Section 1). Then we present our theoretical and practical contribution about mapping reads on a de Bruijn graph (Section 2). In a last part, we show a direct application of such a method for read correction (Section 3).

3.1 Genome representations

3.1.1 Reference sequences

The current main representation of a genome is a set of sequences. Fasta files (Figure 3.1) seemed at first adapted to represent a genome. Due to the hardness of producing a genome of reasonable quality via the sequencing data, the loss of allelic information was considered secondary and non-impacting Slightly chimeric sequences were seen as a small cost to pay to access the global structure of the genome. The hardness of the haplotype separation, still present today, tends to show that it was a reasonable choice to postpone such a challenge. But there are several reasons why the set of sequences representation is not ideal and we detail them in the following paragraphs.

Heterozygosity Actual reference sequences are generated by assemblers that ignore the heterozygosity information. Sequences produced from several haplotypes contain nucleotides from different alleles (Figure 3.2). When the dissimilarity between alleles is high, or when they present structural variants [89] (such as a large insertion), the output assembly may present chimeric sequences.

We could argue that we could still use different sequences to represent the alleles. The challenge is to be able to link the differences together in order to produce the different allele sequences. This process is called phasing (Figure 3.2). For low heterozygosity rate (of the order of one per thousand for a human), variation may be quite distant. In order to phase the successive variants it would require information with very long range

```
>seq1
CATGCATCGATGCCATCGATCGATCTAGCTGACTGATCCATGCCTGACCGATCGTAGCTAGCTA
CTACTGTACGTGACTGCTAGTCATCGATCGCATGACTG
>seq2
CATGACCATCGTAGCTAGCGACGTAGCTACTGATCTAGCTGATCGTACGTAGCTGATCTAGCTG
ACTGCT
...
```

Figure 3.1: Example of fasta file. The first line in a FASTA file starts either with a ">" symbol. Following the initial line (used for a unique description of the sequence) is the actual sequence itself in standard one-letter code

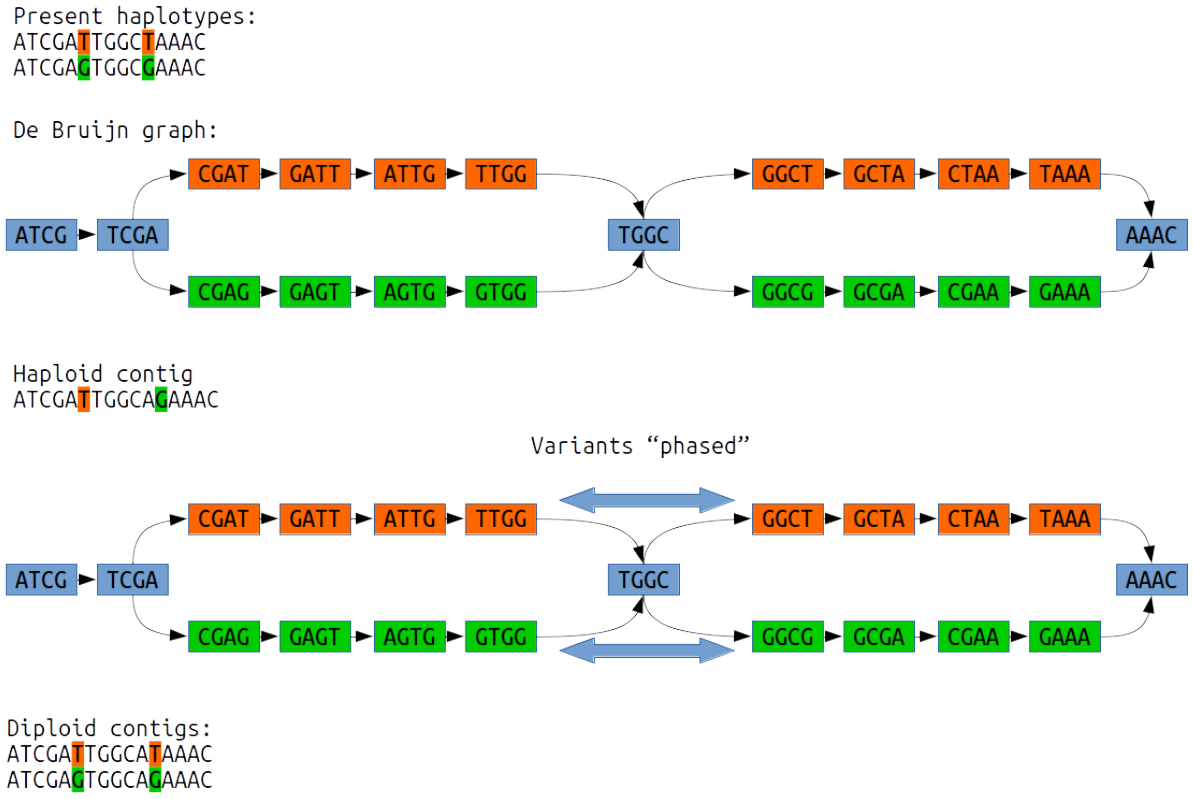

Figure 3.2: Polyploid assembly opposed to haploid assembly. Most assembler practice "haploid" assembly (top) and crush bubbles and output contigs that may contain sequences from different haplotypes. A polyploid assembly would consist to produce haplotypes consistent contigs (bottom).

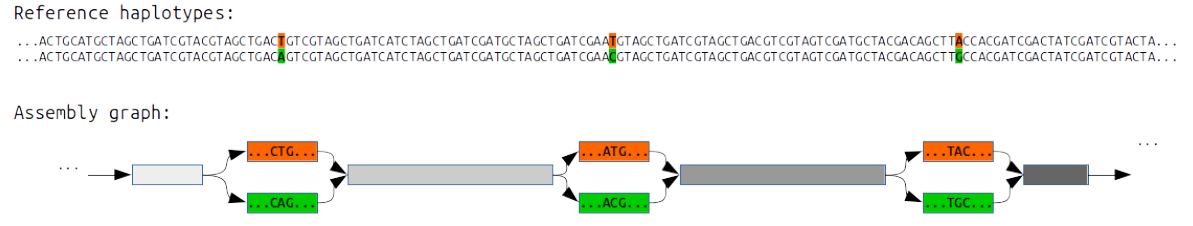

Figure 3.3: Diploid assembly can result in a suit of bubbles. In this example a diploid genome is assembled, homozygous regions in gray are shared by the two haplotypes. Each difference between them create a bubble in the graph.

(Figure 3.3). This fact explains why it appears almost impossible to phase genomes with reads of hundreds of bases.

Nevertheless we are able to study variants even if we are not able to reconstruct the haplotypes. Some variants can be detected from the assembly graph [90] and reported by the assembler. The question may be "how to represent the variant information across the genome?". To place each variant at a position on a reference may be extremely useful. However it presents a concerning allele bias [91] [92]. If some small differences may not be a problem and can be easily be positioned on the reference, larger structural variations are not straightforward to encode and may be completely absent in the reference [93]. Those points show that the linear representation is not a good support for the heterozygosity information yet.

Fragmented genomes Nowadays, most genomes are unfinished and are left at the state of contigs or scaffolds. While this is sufficient for some applications like reads mapping, one can argue that the linear representation loses some information in the case of unfinished assembly. During the assembly process, contigs and scaffolds are extended until an ambiguity on the way to extend is encountered. Even if there are several possibilities that justify the contigs not to be extended, reads mapping could exploit the fact that we know possible extensions for its alignment. Such information could lead to a more efficient mapping, especially around repeats usually responsible for broken contigs . Another problem is that usually, small contigs are not output by assemblers and neither used for scaffolds creation or as references to map on. Again, there are also very good reasons to do so, such as contigs shorter than reads occasioning multiple mapping problems. However, assembly may present "holes" when a complex and hard to assemble region appears. Such a potential source of bias tends to show that a set of sequences representing scaffolds or contigs is not satisfying.

Redundancy Even if we were able to perfectly assemble genomes, we would be interested for many biological applications to store several genomes of different individuals. Very wide projects such as the 1000 genomes [94] plan the sequencing of massive amounts of individuals to study genomic variations among them. The raw storing of the individual sequences is highly inefficient in both terms of storage and searching because of the high redundancy due to the similarity among individuals from a same species. Since the differences between individuals may be very low and the amount of data very large, highly efficient structures may be used in order to index or store the amount of data in an entropic efficient way [95, 96].

3.1.2 Genome graphs

Graph structure is a natural candidate to represent genome for several reasons. Several current representations of a genome as a graph vastly depend on the planned use, as no data structure prevails for common use yet [97] [98]. To index multiple genomes, first representation used annotations to locate variations with respect to a reference [99]. This approach presents several flaws as scalability issues, reference bias and impossibility to represent large structural variations. For pan-genomes [100] propose an ordering of the set of linear sequences, providing an efficient way to create pan-genome references for

reads mapping. In order to represent the variations among haplotypes, the variation graph has been introduced, where haplotypes are encoded as a set of walks through a sequence graph. In order to perform queries on such heavy data structures, the Positional BWT (PBWT) have been proposed [101] and improved in gPBWT [102] to better represent the haplotypes added in the graph. Those structures are very similar to linear sequences since they are supposed to represent finished genomes. Even if these sequences represent a real "map" of a genome with a coordinate system, this is still a hard task to obtain such sequences. Several uses can benefit from a non fully positioned reference: quantification [103] [104], assembly, scaffolding and indexation of repeated sequences or fragmented references [105]. Assembly graphs are natural candidates for this task as they are literally conceived to represent fragmented data. In the following, we present methods and applications relying on assembly graphs as references. Even if such approaches could be based on any assembly graph, the following will focus on the use of a de Bruijn graph.

3.1.3 De Bruijn graphs as references

This part concentrates on the interest of the de Bruijn graph to represent genomes. We will present the following arguments:

- Easiness to construct from sequencing data
- Efficient representation of repeated data
- Represents variants and complex regions

Construct a reference de Bruijn graph To use a de Bruijn graph as a reference, it has to contain the whole genomic information with almost no sequencing error. Building a de Bruijn graph from a reference genome is easy but has a limited interest since the transformation in a de Bruijn graph is lossy as sequences respective positions may be lost. To represent fragmented references such as contigs, we can argue that a graph may contain more information than the contigs set itself. The real challenge is to produce a good reference from raw sequencing data. We present here different strategies to produce a good reference de Bruijn graph and evaluate them. In order to do this, we simulated reads from a known reference and compare the set of kmers from the reference and from the constructed graph. Two kinds of errors may appear:

- Some kmers are in the graph but not in the reference. They are erroneous kmers, also called "false positives"
- Some kmers are in the reference but not in the graph. They are missed kmers also called "false negatives"

As described in the introduction, the main technique to get rid of erroneous kmers is to remove low abundance ones. Three zones can be observed in a kmer spectrum:

- A very high peak of low abundance kmers
- A flat zone
- A bell containing high abundance kmers

The essential idea is that genomic kmers will mostly be present in the third zone while most erroneous kmer will be present in the first zone. By applying a coverage cutoff we only keep the right part of the spectrum curves. The question is how to decide the cutoff parameter. Techniques exist to find the end of the first peak by trying to fit models on the kmer spectrum curve such as kmerGenie [34]. A deeper sequencing makes easier

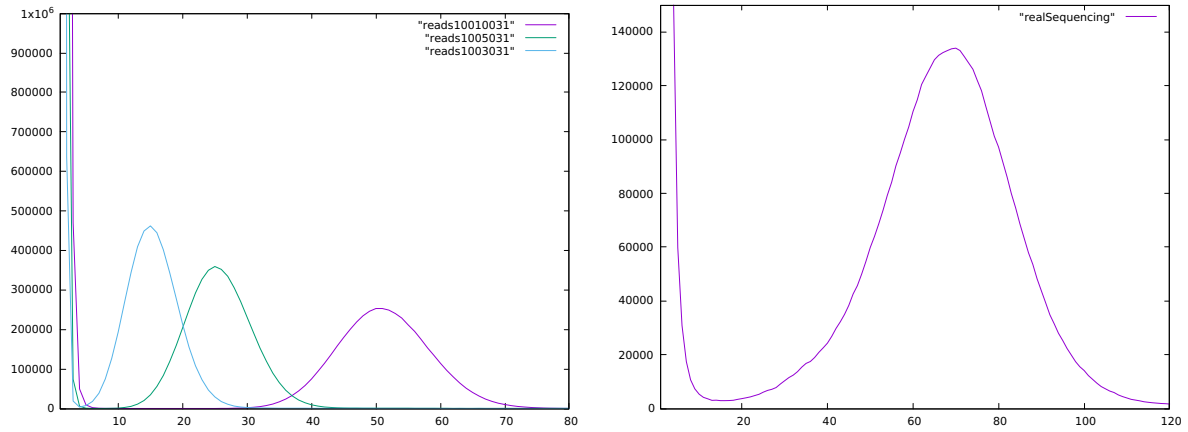

Figure 3.4: Spectrum of 31mer from simulated reads from de Bruijn graph reference (Left). Reads were of length 100 and with coverages of 30 (blue), 50 (green) and 100 (purple) respectively. We observe that higher coverage results in a larger bell zone more distant from the error peak. It is thus easier to separate the erroneous kmers from others with a high coverage. On the right a kmer spectrum of a real sequencing with $k = 31$ and a coverage around 100X from *E. coli*. We can see that unlike spectrum from simulated dataset, the "hole" between the peak and the bell do not reach a value of zero since simulated dataset may not represent all aspect of NGS sequences distribution.

the determination of such a cutoff, as can be seen in Figure 3.4 at left. We note that on simulated datasets, the determination of the cutoff can seem easy because the flat zone has no element. In practice on real dataset (in Figure 3.4 at right), the situation is not this simple since a notable number of element are present with the unexpected medium abundances. The value of k also has a high influence on the kmer spectrum. In Figure 3.5 we observe that the larger the k, the lesser the genomic kmers are covered and the closer to zone 1 and 3. If k is too high compared to the read length it may be difficult to separate the genomic and erroneous kmers since some genomic kmers will be present with a very low abundance. This is mainly due to the fact that when k comes close to the read length, very few kmers are produced by each read. To reduce this problem with large kmers, a step of read correction highly reduces the amount of errors present in the reads and allow genomic kmers to be more covered and therefore to "move" the curve to the right as shown in Figure 3.5.

Another way to remove sequencing errors it to remove "tips" or "short dead ends" from the graph. If $k \geq ReadSize/2$, an isolated sequencing error will not form a bubble but a tip in the graph. As we may be afraid of losing genomic kmers with the solidity cutoff, we advocate the use of low thresholds and therefore apply a tip removal step on the graph. We now evaluate in Tables 3.1 and 3.2 the number of false positives and false negatives in the produced graph based on different strategies using the following methods:

- Kmer filtering with a given threshold
- Using raw reads or reads corrected with the Bloocoo corrector [106]
- Tipping of the graph, removing tips shorter than the read length

We can see that the combination of kmer filtering, read correction and tipping can result in almost perfect reference graphs. Such tasks are the basic steps of assembly. Most

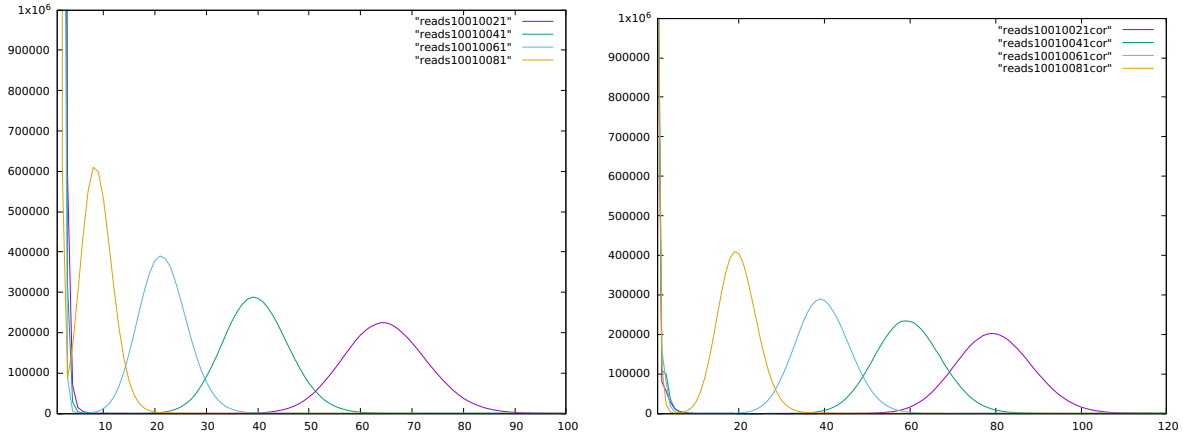

Figure 3.5: Raw reads versus corrected reads. Spectrum with various k from 21 to 81. Reads are simulated from *E. coli* reference. Reads were of length 100 and with a coverage of 100. The k values are 81 (yellow), 61 (blue), 41 (green), 21 (purple). The left plot is the kmer spectrum of reads without correction and the right plot is the kmer spectrum of the reads after correction with Bloocoo. We can see that kmer spectrum from corrected reads are more distant from the error peaks. Reads correction allows an easier erroneous kmers filtering. We also observe that higher kmer size may be difficult to filter as less coverage discrepancy is present to separate erroneous and genomic kmers.

Strategy	# Unitig	FP	FN
Filtering 2	434,510	3,876,482	8
Filtering 2 + tiping	2,474	29,515	8
Filtering 3	28,310	198,088	11
Filtering 3 + tiping	978	582	11
Filtering 5	1273	1716	19
Filtering 5 + tiping	944	17	19
Correction + filtering 2	6,860	16,202	2
Correction + filtering 2 + tiping	1,034	591	2
Correction + filtering 3	1,061	713	7
Correction + filtering 3 + tiping	956	73	7
Correction + filtering 5	952	59	9
Correction + filtering 5 + tiping	944	17	9
Reference	941	0	0

Table 3.1: Evaluation of the de Bruijn graph created from simulated sequencing from *E. coli*, 100x coverage of 100 base pairs reads. We constructed a de Bruijn graph with $k = 51$ and evaluated the number of erroneous kmers (FP) and missing kmers (FN). The abundance threshold (filtering) applied is indicated in the strategy description.

Strategy	# Unitigs	FP	FN
Filtering 2	10,167,089	89,087,143	6
Filtering 2 + tiping	3,580,486	2,696,841	55
Filtering 3	1,103,166	7,294,417	7
Filtering 3 + tiping	432,648	813,483	30
Filtering 5	303,908	1,049,812	15
Filtering 5 + tiping	204,481	231,083	33
Reference	64,044	0	0

Table 3.2: Evaluation of the de Bruijn graph created from simulated sequencing from *C. elegans*, 100x coverage of 100 base pairs reads. We constructed a de Bruijn graph with $k = 51$ and evaluated the number of erroneous kmers (FP) and missing kmers (FN). The abundance threshold (filtering) applied is indicated in the strategy description.

tools add other simplifications and heuristics to produce larger contigs. While there is a plethora of assemblers, we would find relevant to propose tools in order to create a clean and safe graph. This kind of unitigs generator, "unitigers", could be used as a front end by assemblers that could apply their different post-treatments according to their specificity directly on the cleaned graph. In the following, we will rely on such steps to get reference de Bruijn graphs with almost no sequencing error and no missed kmer.

Efficient representation of redundancy We have previously seen that tools exist to construct the de Bruijn graph in an efficient way. The graph constructions of table 3.1 on *E. coli* took less than 10 minutes with BCALM2. The usage of a (non compacted) de Bruijn graph does not seem interesting since a genome will require $\approx genomeSize$ kmer to be stored. On the other hand, the compacted de Bruijn graph can efficiently represent a genome since each nucleotide will require 2 bits plus the global overhead of the unitigs number.

As shown in the table 3.3 that compacted de Bruijn graph is a space efficient genome representation. Based on such observations, some tools were developed in order to efficiently construct the de Bruijn graph as the representation of a genome or multiple genomes [107] [108] [109]. We already presented the efficiency of BCALM2 in order to construct a compacted de Bruijn graph but it was designed to handle very large reads sets and is not necessarily efficient for de Bruijn graph construction from reference genomes. Twopaco [109] proposed an efficient approach to build such graphs from many reference genomes and was able to construct the de Bruijn graph of one hundred human genomes in less than a day. This method to construct a de Bruijn graph from a reference genome without splitting the reference into its set of kmers is way more efficient than BCALM2.

To access the efficiency of the de Bruijn graph to represent multiple genome, we downloaded 100 *E. coli* completed genome from NCBI and created the de Bruijn graph of those merged reference genomes (Table 3.4). We can see that the de Bruijn graph is a compact data structure to store a large number of genomes even if a larger k present a less interesting compression ratio. For comparison the reference file zipped represented 153 mega bytes. Indeed the de Bruijn graph representation is lossy and can not be compared to the lossless zip compression but [110] argue that this efficient redundancy factoring

Genome	Structure	Number of nucleotides used for representation
E. coli	Reference sequence	4,639,675
E. coli	de Bruijn graph k=51	232,770,375
E. coli	de Bruijn graph k=101	462,106,108
E. coli	Compacted de Bruijn graph k=51	4,611,175
E. coli	Compacted de Bruijn graph k=101	4,619,908
C. elegans	Reference sequence	100,286,401
C. elegans	de Bruijn graph k=51	4,889,975,931
C. elegans	de Bruijn graph k=101	9,864,383,665
C. elegans	Compacted de Bruijn graph k=51	104,537,731
C. elegans	Compacted de Bruijn graph k=101	103,927,665
Human	de Bruijn graph k=51	138,318,741,180
Human	de Bruijn graph k=101	286,185,334,362
Human	Compacted de Bruijn graph k=51	2,809,115,473
Human	Compacted de Bruijn graph k=101	3,083,875,662

Table 3.3: Nucleotides number in de Bruijn graph representations of a genome.

Dataset	# Nucleotides
Reference file	485,354,804
k=25	55,864,796
k=51	89,669,228
k=101	137,742,996
k=201	188,631,137
k=301	215,417,352

Table 3.4: Size of the de Bruijn graph of 100 *E. coli* genomes according to k value used for its construction.

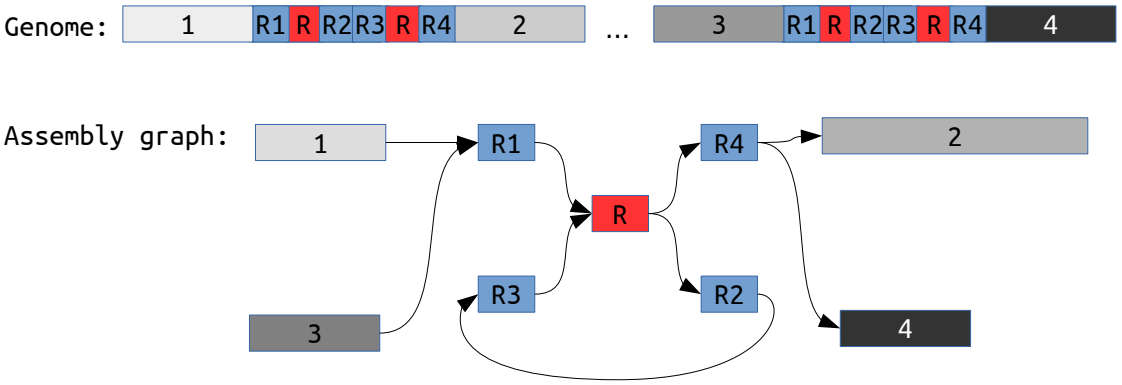

Figure 3.6: Example of nested repeats. In this example, a large repeat in blue contains a small repeat in red. This scenario creates complex cases where the repeats patterns can be nested in one another's.

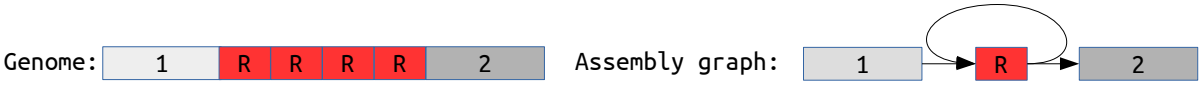

Figure 3.7: Example of tandem repeats. In this example, multiple occurrences of a repeat are consecutive in the genome. This scenario creates cases where it is hard to estimate the number of times the repeat sequence should be present in the assembly.

propriety could be useful for pan-genome analysis.

Complete reads information Assembly, based on the de Bruijn graph or not, may be difficult, especially around repeats or more complex patterns as quasi-repeats (Figure 1.20), nested (Figure 3.6) or tandem repeats (Figure 3.7). Even read mapping is hard around such complex repeats and some regions can show mappability problem [111], since reads can have several possible mappings. The de Bruijn graph representation allow a reduced issue as the repeats of the genome are collapsed. In the cases of an unfinished genome represented by a set of contigs, the problem is different. Contigs are produced from reads, using various heuristics to select path from an assembly graph. This selection may induce biases according to the heuristics used, especially around complex patterns. For example a read not mapped on the contigs set may be due to missing sequences in the contigs outputed. The assemblers biases could be problematic and we argue that mapping directly on a de Bruijn graph can provide a less biased and more complete reference.

In those very complex zones, most assemblers are incapable to output large contigs. Most of the time those regions are absent from the contig set, because they are not outputted by the assembler. Indeed assembler that output all contigs produce small contigs in such cases. One conclusion is that scaffolding may be impracticable on such situations because it rely on read mapping on large contigs. The situation of those region can not be improved by usual scaffolding methods, and are lost in most assembly processes. In such regions, the graph is complex and branching but the genomic sequences are still paths of the graph. The information carried by the graph is then superior to the

one brought by contigs, as the graph contains information either on branching regions or on how the contigs could be ordered.

A second point is that the graph can carry the haplotypes information. In presence of multiple closely related genomes or haplotypes, minor variations will be represented in bubbles in the graph in an efficient way. De Bruijn graph is an efficient structure to capture variant information and several methods use the de Bruijn graph to discover polymorphisms among several individuals [90] [112] [99] [113]. This property has already been used to index several genomes in order to avoid multiple alignment in whole genomes comparisons [114] [107] and metagenomic quantification [103].

"Genome representations" core messages:

- Linear sequences may not be the ideal representation of a genome
- Several graphs based structures have been proposed to fit various applications to represent finished or almost finished genomes
- The de Bruijn graph may be an efficient structure for applications where a reference genome is not available

3.2 Read mapping on the de Bruijn graph

3.2.1 An efficient tool for an NP-Complete problem

As we have seen, the de Bruijn graph can be efficiently used as a reference. Read mapping is a core operation we want to perform on a reference and most tools are designed to work on a reference genome represented by flat sequences, fragmented or not [115] [60] [59]. Those tools are able to index and to map reads on a set of sequences. The problematic of read mapping is to know if a read can be aligned on the reference, where and with an indication on the alignment quality. As we mentioned before, some complex repeated regions sometimes occur in genomes, and suffer from low mappability [111]. Reads from repeated region may have multiple matching sites and then be hard to be mapped with high confidence. When a genome is represented as a de Bruijn graph, repeated regions are merged in the graph. This factorization reduces the problem of multiple mapping. This de Bruijn graph property explains its usage in genomic or meta-genomic, even on finished reference genomes [105]. We argue that being able to map reads directly on such structures would be less biased and more complete than mapping on contigs.

Another interest of mapping reads on a de Bruijn graph is to improve the de Bruijn graph itself to produce a better assembly. This read information added in the graph can be used to avoid false connections between nodes or to solve some repeats. Surprisingly, despise the interest in such techniques, no practical solution has been designed for this task. Assemblers using this kind of information do not present generic procedure or rely on alignment on flat sequences. All those different usages motivates the need of such tools.

This paper proposes a formal definition of the problem of mapping reads on a de Bruijn graph and proves its NP-completeness. We also provides a practical solution based on several heuristics, called BGREAT.

Technical results are described in the paper "Read mapping on de Bruijn graphs" [116] included at the end of the chapter.

BGREAT is based on simple heuristics in order to be fast and scalable. It follows the seed and extend paradigm by indexing the $k-1$ suffix and prefix of the unitigs as anchors. Once an anchor is found on a read, the rest of the read sequence is mapped to the graph unitigs. By indexing the prefix and suffix and associating them the indices of the unitigs sharing them, BGREAT is able to navigate through the graph unitigs as it was presented in the last chapter. Some unitigs may not contain any anchor, for example if they map entirely on large unitigs. But in such cases those reads can be mapped with a regular mapping tool. Another essential heuristic of BGREAT is the greediness of its mapping process. When several paths can be used, the path with the minimal number of mismatches is picked. The use of those greedy heuristics allows very fast mapping of reads even on a large de Bruijn graph. BGREAT is able to map a human dataset of 3 billion reads on a de Bruijn graph created from it, in less than 5 hours using 20 cores and 10GB. Another interesting conclusion is that we are able to map more reads on the de Bruijn graph than on a set of contigs. An experiment on a human dataset showed that only 63% of the reads were mapped on the contig set where 85% were mapped on the de Bruijn graph using the graph alignment method.

3.2.2 Mapping refinements

The amount of potential applications of read alignment on de Bruijn graph leads us to the conception of several improvements of the BGREAT tool. We propose an advanced version called BGREAT2 (unpublished) with following improvements that we will detail:

- Improved anchor system
- Indexing based on a MPHF
- Optimal mapping and multi-mapping management

In order to improve the performance of our anchoring scheme, we no longer use the $k-1$mer at the extremities of the unitigs. We index kmers from the unitigs and associate to each kmer its position in the unitigs. This modification allows two significant changes.

First, the size of the anchor can be chosen. Therefore a smaller k than the one selected to construct the graph can be used. The order of the de Bruijn graph may be quite large, thus a $k-1$mer would be a bad anchor in terms of sensibility. This way, a high order graph can be used without presenting a change in the anchoring scheme. Indexing all kmers of the graph can be costly, for a human graph indexing all kmers can use up to 200 GB. A parameter can be defined in order to reduce the memory usage of BGREAT2 that regulates which fraction of the kmers are indexed. We can show that even by indexing only a fraction of the kmers (one out of 10 by default), the mapping performances of BGREAT2 are not highly impacted. Indexing 1 out of 10 anchors in each unitig, with at least one anchor indexed by unitigs allowed BGREAT2 to reduce its memory usage from 6,898 MB to 928 MB to index a *C. elegans* reference graph. The indexing time also dropped from 61 to 10 seconds while the ratio of mapped reads went from 99.6% to 99.3%. The mapping time was also impacted, but in a less impressive way and improved from 11k to 15k reads by second on a single thread.

The second interest of the new indexing scheme is that BGREAT2 is self contained and do not need a mapping tool like Bowtie. Since kmers are indexed along the whole length of the unitigs, reads mapping inside a unitig can be anchored and mapped. In order to index kmers and the de Bruijn graph overlaps, BGREAT2 no longer uses a dynamic hash table but the BBhash MPHF. This choice allows a highly reduced memory usage and a faster access to the unitigs compared to previously used dynamic hashing.

We also improved the mapping rules. The algorithm tries to find a perfect mapping (without error), then a mapping with one error and so on until the maximal error number allowed. This way, the obtained mapping is guaranteed to contain a minimal number of mismatch. Several multiple mapping strategies are also proposed: output all equivalent mapping, output one mapping, or none. Albeit some optimizations and features have still to be added, those improvements allow BGREAT2 to be more than a proof of concept. We will present in the following sections several uses of this tool.

"Read mapping on the de Bruijn graph" core messages:

- Mapping reads on a de Bruijn graph is a hard problem
- With appropriate heuristics we can propose a satisfying and efficient mapping
- Mapping on the de Bruijn graph may allow more mapping hits than mapping on contigs

3.3 De novo, reference guided, read correction

We have shown previously that an almost errorless de Bruijn graph could be constructed. We therefore propose a new method of correction for short reads by aligning them on a reference represented by a de Bruijn graph constructed from the reads themselves. Correction methods based on alignment on a reference genome would have the two same essential drawbacks than reference guided assembly:

- Need for a reference
- Correction may be biased by the reference used

Here we propose to construct the reference directly from the reads to be corrected in order to avoid biases. We argue that we do not need to produce a good assembly in order to obtain a reference good enough to correct the reads. The idea of such a technique can be summarized in the following steps:

- Graph construction
- Graph cleaning
- Read mapping on the graph

3.3.1 Limits of kmer spectrum

Most state of the art correctors are based on kmer spectrum techniques. Other techniques based on suffix array or on multiple alignment, are presented in [117]. Those techniques are less used because they rely on memory expensive data structures and do not scale well on large genomes and datasets. Kmer spectrum correction was proposed when NGS

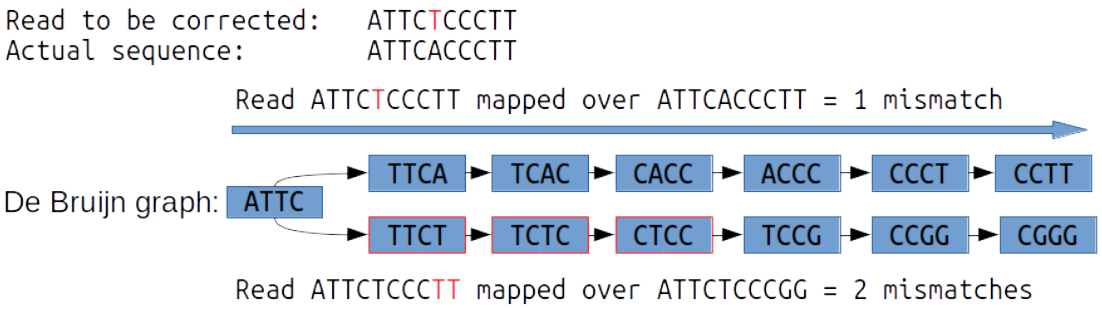

Figure 3.8: Example of read correction by mapping it on a de Bruijn graph. The query read is mapped onto the graph and the bases of the graph are trusted over the read ones. The read is thus corrected according to the reference graph. An alternative mapping for the reads would be ATTCACCCGG but this path includes two mismatches and is therefore not optimal. This example also shows a problem of kmer spectrum technique. In the read to correct, the sequencing error (the red T) is validated by the genomic kmers TTCT, TCTC and CTCC (framed in red). According to heuristics employed by those tools, this base is unlikely to be a sequencing error and will not be corrected.

reads appeared and were assembled with de Bruijn graph [46]. As in de Bruijn graph assembly, a set of "solid" kmers are computed. Those highly abundant kmers are very likely to be errorless and are indexed. When a "non solid" kmer is found in a read, the tool tries to convert it into a solid kmer in the most parsimonious way. Since such correctors only index trusted kmers against low abundance ones, bases of a read covered by trusted kmers will be considered as correct. In the example of Figure 3.8, all bases of the read will be considered correct because they are covered by several solid kmers. When an erroneous base is found, the algorithms try to replace the erroneous kmers by solid kmers. A hard part for those kind of algorithms is when close errors are found. If multiple errors appear on a single kmer, not all the possible alternative kmers can be tested and no real satisfying solution has been found for such cases.

3.3.2 Reads correction by de Bruijn graph mapping

Here we present BCOOL, a still unpublished proof of concept short reads corrector based on read mapping on a de Bruijn graph. The idea is to create an almost errorless de Bruijn graph from the reads to be corrected and use it as a reference to correct the reads mapped on it. After the graph cleaning step, we map each read on the graph and when a mismatch is detected, the nucleotide of the graph is trusted over the nucleotide of the read, and the read is modified to match with the de Bruijn graph sequence.

What are the differences between the kmer spectrum correction technique and this proposition? Both kmer spectrum and our approach use a set of trusted kmers as reference to correct the reads. The difference is that we use the structure of the de Bruijn graph, *i.e.* how the kmers are connected. There are several advantages to use this information. Tip removal can typically remove sequencing errors above the solidity threshold. It allows our set of solid kmers to contain less erroneous kmers. We also have a better comprehension of complex cases. In Figure 3.8 a kmer spectrum method would not correct anything since the sequencing error is covered by multiple solid kmers and would consider it safe. Our

Kmer size used	FP	FN	TP	Correction ratio	% Erroneous read
BCOOL k=21	2,731	11,235	4,627,370	332	0.16
BCOOL k=31	1,092	2,427	4,636,805	1,318	0.04
BCOOL k=41	915	**1,593**	**4,637,775**	**1,850**	**0.03**
BCOOL k=51	1,150	2,183	4,637,245	1,392	0.03
BCOOL k=61	2,146	2,590	4,636,858	980	0.04
BCOOL k=71	2,452	43,202	4,596,267	102	0.58
BFC	2,105	78,679	4,559,123	57	0.67
Bloocoo	4,147	209,071	4,430,443	22	3.49
Lighter	**624**	6,538	4,633,059	648	0.09
Musket	1,031	6,456	4,632,901	620	0.13

Table 3.5: Correction benchmark on simulated 100bp reads (100X coverage 1% error rate) from *E. coli* by varying the k parameter. Solidity threshold to construct the de Bruijn graph was fixed to 5 and tipping length to 100.

method maps the whole read, making use of the sequences next to the error to correct it. Close errors can be problematic for other correctors, when multiple errors occur on a kmer the number of kmers able to replace it is exponential. It would be difficult to check the solidity of all possible kmers to replace the erroneous one. Our alignment method is not impacted by such cases since we do not correct kmers but entire reads. In order to assess the potential of such a technique we tested it against state of the art read correctors. In our benchmarks we will compare those four correctors Musket [118], BFC [119], Lighter [120] and Bloocoo [106] for their performances in our tests and in several published benchmarks.

We generated simulated reads with known errors, corrected them and evaluated the correction results. We use several metrics to evaluate the correction quality. The false positives are correct bases that have been modified or incorrect bases that have been wrongly modified. It can be seen as errors introduced by the correction. The false negative are sequencing errors that were not modified. It can be seen as not corrected errors. The true positives are sequencing errors correctly corrected. To provide a global measure of the correction, we evaluate the number of errors before and after the correction, counting FP and FN. A ratio of 10 mean that the amount of error was divided by ten by the correction. We also present the read percentage that contain at least an error.

The first results on the *E. coli* bacteria are reported in Table 3.5. We can observe that most tools propose a very good correction. All tools achieve to divide the error rate by more than 10, showing that they take care of most sequencing errors. Results also show that almost all reads are errorless after the correction. We see that BCOOL is able to provide excellent correction with adapted parameters. Globally, we argue that correctors provide very good corrections on small genomes.

The results on the larger genome of *C. elegans* are presented in Tables 3.6 and 3.7 on reads of length 100 and 250 respectively. Here we see that correctors still provide good correction by dividing the global error rate and proposing a vast majority of perfect reads but in a less impressive way than on *E. coli*. Here again BCOOL provides very interesting results on both read length of 100 and 250.

The results on the human genome are presented in Table 3.8. We see that BCOOL is

Kmer size used	FP	FN	TP	Ratio error rate	% erroneous read
BCOOL k=21	1,455,866	6,185,857	93,792,770	13	4.05
BCOOL k=31	1,420,047	1,529,334	98,558,023	34	1.57
BCOOL k=41	883,155	753,403	99,419,269	61	0.87
BCOOL k=51	651,040	498,536	99,716,829	87	0.59
BCOOL k=61	625,830	**447,835**	**99,795,401**	**93**	**0.49**
BCOOL k=71	803,946	1,214,915	99,042,681	50	0.97
BFC	**289,081**	4,020,868	96,187,218	23	2.23
Bloocoo	800,396	6,471,767	93,805,016	14	5.23
Lighter	613,314	2,789,652	97,490,978	29	2.16
Musket	1,120,401	4,932,597	95,250,573	17	4.47

Table 3.6: Correction benchmark on simulated 100bp reads (100X coverage 1% error rate) from *C. elegans* by varying the k parameter. Solidity was fixed to 5 and tipping length to 100.

Kmer size used	FP	FN	TP	Ratio error rate	% erroneous read
BCOOL k=21	1,048,454	7,723,461	92,278,080	11	7.72
BCOOL k=31	1,039,760	3,501,749	96,582,586	22	4.20
BCOOL k=41	787,674	2,099,774	98,052,234	35	2.67
BCOOL k=51	608,383	1,424,096	98,771,993	49	1.87
BCOOL k=61	500,090	1,132,677	99,092,022	61	1.49
BCOOL k=71	411,359	996,580	99,247,373	71	1.28
BCOOL k=81	343,784	950,693	99,304,838	77	1.27
BCOOL k=91	301,430	**944,491**	**99,318,476**	**80**	**1.12**
BCOOL k=101	271,094	978,287	99,291,125	80	1.12
Musket	812,832	3,691,147	96,498,970	22	7.22
Lighter	520,365	3,022,271	97,254,509	28	4.76
Bloocoo	873,717	5,746,372	94,519,928	15	10.38
BFC	**176,279**	6,289,736	93,933,429	16	5.26

Table 3.7: Correction benchmark on simulated 250bp reads (100X coverage 1% error rate) from *C. elegans* by varying the k parameter. Solidity was fixed to 5 and tipping length to 100.

Kmer size used	FP	FN	TP	Ratio error rate	% erroneous read
BCOOL k=41	186,222,378	165,218,857	2,346,232,116	7	5.16
BCOOL k=51	137,556,747	115,336,678	2,400,321,919	10	3.52
BCOOL k=61	99,095,091	88,713,912	2,429,964,113	13	2.47
BCOOL k=71	**85,492,564**	**79,637,965**	**2,440,258,246**	**15**	**2.10**
Lighter	55,280,808	883,251,227	1,632,909,855	3	27.83
Bloocoo	80,025,724	374,402,778	2,148,076,789	6	11.16
BFC	88,500,362	962,587,329	1,497,440,057	2	27.59

Table 3.8: Correction benchmark on simulated 100bp reads (100X coverage 1% error rate) from Human by varying the k parameter. Solidity was fixed to 5 and tipping length to 100.

able to scale up to large genome and dataset as the human one. Musket needs more than 250 GB and was not able to correct this dataset on our testing server. Once again we see that correctors provide less impressive results on larger genomes. Those results show that BCOOL provides good read correction even compared to state of the art correctors.

Our proposed workflow can be summarized into:

- Best kmer size and abundance threshold selection (Not done yet)
- Graph construction
- Graph cleaning
- Reads mapping on the graph

We see that BCOOL can lead to significant improvements to reads correction and can scale up to very large genomes. But the approach is still parameter dependent. A high k value allows the construction of a high quality graph but can create holes. Furthermore we only evaluated BCOOL on very high coverage dataset and the solidity threshold is also an impactful parameter that can be non trivial to set for users not aware of de Bruijn graph properties. An automated evaluation of the kmer spectrum could infer the good values to set, in order to get rid of most sequencing error without losing genomic kmers. In order to infer those parameters we could, as in Kmergenie, use a fast kmer spectrum estimator such as ntCard [121]. Many improvements and tuning have yet to be performed to propose an efficient and robust corrector but preliminary results are encouraging.

"De novo, reference guided, read correction" core messages:

- Mapping reads on a reference de Bruijn graph enables to correct them
- All operations of such a method are scalable

Limasset *et al. BMC Bioinformatics* (2016) 17:237
DOI 10.1186/s12859-016-1103-9

BMC Bioinformatics

RESEARCH ARTICLE Open Access

Read mapping on de Bruijn graphs

Antoine Limasset[1*], Bastien Cazaux[2,3], Eric Rivals[2,3] and Pierre Peterlongo[1]

Abstract

Background: Next Generation Sequencing (NGS) has dramatically enhanced our ability to sequence genomes, but not to assemble them. In practice, many published genome sequences remain in the state of a large set of contigs. Each contig describes the sequence found along some path of the assembly graph, however, the set of contigs does not record all the sequence information contained in that graph. Although many subsequent analyses can be performed with the set of contigs, one may ask whether mapping reads on the contigs is as informative as mapping them on the paths of the assembly graph. Currently, one lacks practical tools to perform mapping on such graphs.

Results: Here, we propose a formal definition of mapping on a de Bruijn graph, analyse the problem complexity which turns out to be NP-complete, and provide a practical solution. We propose a pipeline called *GGMAP* (Greedy Graph MAPping). Its novelty is a procedure to map reads on branching paths of the graph, for which we designed a heuristic algorithm called *BGREAT* (de Bruijn Graph REAd mapping Tool). For the sake of efficiency, *BGREAT* rewrites a read sequence as a succession of unitigs sequences. *GGMAP* can map millions of reads per CPU hour on a de Bruijn graph built from a large set of human genomic reads. Surprisingly, results show that up to 22 % more reads can be mapped on the graph but not on the contig set.

Conclusions: Although mapping reads on a de Bruijn graph is complex task, our proposal offers a practical solution combining efficiency with an improved mapping capacity compared to assembly-based mapping even for complex eukaryotic data.

Keywords: Read mapping, De Bruijn graph, NGS, Sequence graph, path, Hamiltonian path, Genomics, Assembly, NP-complete

Background

Next Generation Sequencing technologies (NGS) have drastically accelerated the generation of sequenced genomes. However, these technologies remain unable to provide a single sequence per chromosome. Instead, they produce a large and redundant set of reads, with each read being a piece of the whole genome. Because of this redundancy, it is possible to detect overlaps between reads and to assemble them together in order to reconstruct the target genome sequence.

Even today, assembling reads remains a complex task for which no single piece of software performs consistently well [1]. The assembly problem itself has been shown to be computationally difficult, more precisely NP-hard [2]. Practical limitations arise both from the structure of genomes (repeats longer than reads cannot be correctly resolved) and from the sequencing biases (non-uniform coverage and sequencing errors). Applied solutions represent the sequence of the reads in an assembly graph: the labels along a path of the graph encode a sequence. Currently, most assemblers rely on two types of graphs: either the de Bruijn graph (DBG) for the short reads produced by the second generation of sequencing technologies [3], or for long reads the overlap graph (which was introduced in the Celera Assembler [4]) and variants thereof, like the string graph [5]. Then, the assembly algorithm explores the graph using heuristics, selects some paths and outputs their sequences. Due to these heuristics, the set of sequences obtained, called contigs, is biased and fragmented because of complex patterns in the graph that are generated by sequencing errors, and genomic variants and repeats. The set of contigs is rarely satisfactory and is

*Correspondence: antoine.limasset@irisa.fr
[1] IRISA Inria Rennes Bretagne Atlantique, GenScale team, Campus de Beaulieu, 35042 Rennes, France
Full list of author information is available at the end of the article

usually post-processed, for instance, by discarding short contigs.

The most frequent computational task for analyzing a set of reads is mapping them on a reference genome. Numerous tools are available to map reads when the reference genome has the form of a set of sequences (e.g. BWA [6] and Bowtie [7]). The goal of mapping on a finished genome sequence is to say whether a sequence can be aligned to this genome, and in this case, at which location(s). This is mostly done with a heuristic (semi-global) alignment procedure that authorizes a small edit or Hamming distance between the read and genome sequences. Read mapping process suffers from regions of low mappability [8]. Repeated genomic regions may not be mapped precisely since the reads mapping on these regions have multiple matches. When a genome is represented as a graph, the mappability issue is reduced, as occurrences of each repeated region are factorized, limiting the problem of multiple matches of reads.

When the reference is not a finished genome sequence, but a redundant set of contigs, the situation differs. The mapping may correctly determine whether the read is found in the genome, but multiple locations may for instance not be sufficient to conclude whether several true locations exist. Conversely, an unfruitful mapping of a read may be due to an incomplete assembly or to the removal of some contigs during post-processing. In such cases, we argue it may be interesting to consider the assembly graph as a (less biased and/or more complete) reference instead of the set of contigs. Then mapping on the paths of this graph is needed to complement mapping on set of contigs. This motivates the design and implementation of BGREAT.

In this context, we explore the problem of mapping reads on a graph. Aligning or mapping sequences on sequence graphs (a generic term meaning a graph representing sequences along its paths) has already been explored in the literature in different application contexts: assembly, read correction, or metagenomics.

In the context of assembly, once a DBG has been built, mapping the reads back to the graph can help in eliminating unsupported paths or in computing the coverage of edges. To our knowledge, no practical solution has been designed for this task. Cerulean assembler [9] mentions this possibility, but only uses regular alignment on assembled sequences. Allpaths-LG [10] also performs a similar task to resolve repeats using long noisy reads from third generation sequencing techniques. Its procedure is not generic enough to suit the mapping of any read set on a DBG. From the theoretical view point, the question is related to the NP-hard *read-threading* problem (also termed *Eulerian superpath problem* [2, 11]), which consists in finding a read coherent path in the DBG (a path that can be represented as a sequence of reads as defined in [5]). The assembler called SPADES [12] threads the reads against the DBG by keeping track of the paths used during construction, which requires a substantial amount of memory. Here, we propose a more general problem, termed *De Bruijn Graph Read Mapping Problem* (DBGRMP), as we aim at mapping to a graph any source of NGS reads, either those reads used for building the graph or other reads.

Recently, the hybrid error correction of long reads using short reads has become a critical step to leverage the third generation of sequencing technologies. The error corrector LoRDEC [13] builds the DBG of the short reads, and then aligns each long read against the paths of the DBG by computing their edit distance using a dynamic programming algorithm (which is slow for our purposes). For shorts reads correction, several tools that evaluate the k-mer spectrum of reads to correct the sequencing errors use a probabilistic or an exact representation of a DBG as a reference [14, 15].

In the context of metagenomics, Wang et al. [16] have estimated the taxonomic composition of a metagenomics sample by mapping reads on a DBG representing several genomes of closely-related bacterial species. In fact, the graph collapses similar regions of these genomes and avoids redundant mapping. Their tool maps the read using BWA on the sequence resulting from the random concatenation of unitigs of the DBG. Hence, a read cannot align over several successive nodes of the graph (ER: il y a un pb ce n'est pas vrai). Similarly, several authors have proposed to store related genomes into a single, less repetitive, DBG [17–19]. However, most of these tools are efficient only when applied to very closely related sequences that result in flat graphs. The *BlastGraph* tool [19], is specifically dedicated to the mapping of reads on graphs, but is unusable on real world graphs (see Results section).

Here, we formalize the mapping of reads on a De Bruijn graph and show that it is NP-complete. Then we present the pipeline *GGMAP* and dwell on *BGREAT*, a new tool which enables to map reads on branching paths of the DBG (Section *GGMAP*: a method to map reads on de Bruijn Graph). For the sake of efficiency, *BGREAT* adopts a heuristic algorithm that scales up to huge sequencing data sets. In Section Results, we evaluate *GGMAP* in terms of mapping capacity and of efficiency, and compare it to mapping on assembled contigs. Finally, we discuss the limitations and advantages the of *GGMAP* and give some directions of future work (Section Discussion).

Methods

We formally define the problem of mapping reads on a DBG and investigate its complexity (Section Complexity of mapping reads on the paths of a DBG). Besides, we propose a pipeline called *GGMAP* to map short

reads on a representation of a DBG (Section *GGMAP*: a method to map reads on de Bruijn Graph). This pipeline includes *BGREAT*, a new algorithm mapping sequences on branching paths of the graph (Section *BGREAT*: mapping reads on branching paths of the CDBG).

Complexity of mapping reads on the paths of a DBG

In this section, we present the formal problem we aim to solve and prove its intractability. First, we introduce preliminary definitions, then formalize the problem of mapping reads on paths of a DBG, called the De Bruijn Graph Read Mapping Problem (DBGRMP), and finally prove it is NP-complete. Our starting point is the well-known Hamiltonian Path Problem (HPP); we apply several reductions to prove the hardness of DBGRMP.

Definition 1 (de Bruijn graph)**.** *Given a set of strings $S = \{r_1, r_2, \ldots, r_n\}$ on an alphabet Σ and an integer $k \geq 2$, the de Bruijn graph of order k of S ($dBG_k(S)$) is a directed graph (V, A) where:*

$V = \{d \in \Sigma^k | \exists i \in \{1, \ldots, n\}$ *such that d is a substring of* $r_i \in S\}$, *and*
$A = \{(d, d') |$ *if the suffix of length* $k-1$ *of d is a prefix of* $d'\}$.

Definition 2 (Walk and Path of a directed graph)**.** *Let G be a directed graph.*

- *A walk of G is an alternating sequence of nodes and connecting edges of G.*
- *A path of G is a walk of G without repeated node.*
- *A Hamiltonian path is a path that that visits each node of G exactly once.*

Definition 3 (Sequence generated by a walk in a dBG_k)**.** *Let G be a de Bruijn graph of order k. A walk of G composed of l nodes $(v_1, \ldots, v_l)$ generates a sequence of length $k+l-1$ obtained by the concatenation of v_1 with the last character of v_2, of v_3 ,..., of v_l.*

We define the *de Bruijn Graph Read Mapping Problem* (*DBGRMP*) as follows:

Definition 4 (De Bruijn Graph Read Mapping Problem)**.** *Given*

- *S, a set of strings over Σ,*
- *k, an integer such that $k \geq 2$,*
- *$q := q_1 \ldots q_{|q|}$ a word of Σ^* such that $|q| \geq k$,*
- *a cost function $F : \Sigma \times \Sigma \rightarrow \mathbb{N}$, and*
- *a threshold $t \in \mathbb{N}$,*

decide whether there exists a path of the $dBG_k(S)$ composed of $|q| - k + 1$ nodes (generating a word $m := m_1 \ldots m_{|q|} \in \Sigma^{|q|}$) such that the cost $C(m, q) := \sum_{i=1}^{|q|} F(m_i, q_i) \leq t$.

We recall the definition of the *Hamiltonian Path Problem* (*HPP*), which is NP-complete [20].

Definition 5 (*Hamiltonian Path Problem* (*HPP*))**.** *Given a directed graph G, the HPP consists in deciding whether there exists a Hamiltonian path of G.*

To prove the NP-completeness of DBGRMP we introduce two intermediate problems. The first problem is a variant of the Asymmetrical Travelling Salesman Problem.

Definition 6 (*Fixed Length Asymmetric Travelling Salesman Problem* (*FLATSP*))**.** *Let*

- *l be an integer,*
- *$G := (V, A, c)$ be a directed graph whose edges are labeled with a non-negative integer cost (given by the function $c : A \rightarrow \mathbb{N}$),*
- *$t \in \mathbb{N}$ be a threshold.*

FLATSP consists in deciding whether there exists a path $p := (v_1, \ldots, v_l)$ of G composed of l nodes whose cost $c(p) := \sum_{j=1}^{l-1} c((v_j, v_{j+1}))$ satisfies $c(p) \leq t$.

We consider the restriction of FLATSP to instances having a unit cost function (i.e., where $c(a) = 1$ for any $a \in A$) and where l equals both the threshold and the number of nodes in V. This restriction makes FLATSP very similar to HPP, and the hardness result quite natural.

Proposition 1. *FLATSP is NP-complete even when restricted to instances with a unit cost function and satisfying $l = |V| = t$.*

Proof. We reduce HPP to an instance of FLATSP where the cost function c simply counts the edges in the path, and where the path length l equals the threshold t and the number of nodes in V.

Let $G = (V, A)$ be a directed graph, *which is an instance of HPP*. Let $H = (V, A, c : A \rightarrow \{1\})$, and $l := |V|$ and $t := l$. Thus (H, l, t) is an instance of FLATSP.

Let us now show that there is an equivalence between the existence of a Hamiltonian path in G and the existence of a path $p = (v_1, \ldots, v_l)$ of H such that $c(p) \leq t$. Assume that G has a Hamiltonian path p. In this case, p is also a path in H of length $|V|$, and then the cost of p equals its length, i.e. $c(p) = \sum_{i=1}^{|V|} 1 = |V|$. Hence, there exists a path p of H such that $c(p) \leq t = |V|$.

Assume that there exists a path $p = (v_1, \ldots, v_{|V|})$ of H such that $c(p) \leq t$. As p is a path it has no repeated

nodes, and as by assumption $l = |V|$, one gets that p is a Hamiltonian path of H, and thus also a Hamiltonian path of G, since G and H share the same set of nodes and edges. □

The second intermediate problem is called the *Read Graph Mapping Problem* (*GRMP*) and is defined below. It formalizes the mapping on a general sequence graph. Hence, DBGRMP is a specialization of GRMP, since it considers the case of the de Bruijn graph.

Definition 7 (Graph Read Mapping Problem)**.** *Given*

- *a directed graph* $G = (V, A, x)$*, whose edges are labeled by symbols of the alphabet* $(x : A \to \Sigma)$*,*
- $q := q_1 \dots q_{|q|}$ *a word of* Σ^**,*
- *a cost function* $F : \Sigma \times \Sigma \to \mathbb{N}$*,*
- *a threshold* $t \in \mathbb{N}$*,*

GRMP consists in deciding whether there exists a path $p := (v_1, \dots, v_{|q|+1})$ *of G composed of* $|q| + 1$ *nodes, which generates a word* $m := m_1 \dots m_{|q|} \in \Sigma^{|q|}$ *such that* $m_i := x((v_i, v_{i+1}))$*, and which satisfies* $\sum_{i=1}^{|q|} F(m_i, q_i) \leq t$*. Here, m is called the word generated by p.*

Proposition 2. *GRMP is NP-complete.*

Proof. We reduce FLATSP to GRMP.

Let $(G = (V, A, c : A \to \mathbb{N}), l \in \mathbb{N}, t \in \mathbb{N})$ be an instance of FLATSP. Let $\Sigma = \{y_1, \dots, y_{|\Sigma|}\}$ an alphabet larger than the largest value of $c(A)$, and let s be the application such that $s : \{0, \dots, |\Sigma|\} \to \Sigma$ and such that for each i in $\{0, \dots, |\Sigma|\}$, $s(i) = y_i$. Let $H = (V, A, x := s \circ c)$ and let α be a letter that does not belong to Σ, let $q = \alpha^{l-1}$ and F such that for each i in $\{0, \dots, |\Sigma|\}$, $F(\alpha, y_i) = i$. Thus, we obtain $|q| = l - 1$.

Now, let us show that there is an equivalence between the existence of a path $p = (v_1, \dots, v_l)$ of G such that $c(p) \leq t$ and the existence of a path $p' = (u_1, \dots, u_{|q|+1})$ of H composed of $|q| + 1$ nodes, which generates a word $m = m_1 \dots m_{|q|}$ of $\Sigma^{|q|}$, where each $m_j = x((u_j, u_{j+1}))$, and such that $\sum_{j=1}^{|q|} F(m_j, q_j) \leq t$. Assume that there exists a path $p = (v_1, \dots, v_l)$ of G such that $c(p) \leq t$. By definition, p is a path in H. Let m be the word generated by p. Thus we have $\sum_{j=1}^{|q|} F(m_j, q_j) = \sum_{j=1}^{l-1} F(m_j, \alpha) = \sum_{j=1}^{l-1} c((v_j, v_{j+1})) \leq t$.

Now, suppose that there exists a path $p' = (u_1, \dots, u_{|q|+1})$ of H composed of $|q| + 1$ nodes, which generates a word $m = m_1 \dots m_{|q|}$ of $\Sigma^{|q|}$, where each $m_j = x((u_j, u_{j+1}))$, and such that $\sum_{j=1}^{|q|} F(m_j, q_j) \leq t$. By the construction of H, p' is a path in G of length $|q| + 1 = l$. Hence, we obtain $\sum_{j=1}^{l-1} c((u_j, u_{j+1})) = \sum_{j=1}^{|q|} F(m_j, \alpha) = \sum_{j=1}^{l-1} F(m_j, q_j) \leq t$. □

Theorem 1. *DBGRMP is NP-complete.*

Figure 1 illustrates the gadget used in the proof of Theorem 1. Basically, the gadget creates a DBG node (a word) formed by concatening the labels of the two preceding edges in the original graph.

Proof. Let us now reduce GRMP to DBGRMP.

Let $(G := (V, A, x : A \to \Sigma), q \in \Sigma^*, F : \Sigma \times \Sigma \to \mathbb{N}, t \in \mathbb{N})$ be an instance of GRMP. Let \$ and Δ be two distinct letters that do not belong to Σ, and let $\Sigma' := \Sigma \cup \{\$, \Delta\}$. Let V' be a set of words of length 2 defined by

$$
\begin{aligned}
V' := & \left\{\alpha_i\beta_j \mid x(i,j) \right. \\
& \left. = \alpha \text{ and } \exists\, l \in V \text{ such that } x(j,l) = \beta \right\} && \textbf{set } 1 \\
\bigcup & \left\{\Delta_i\$_i \mid \exists\, j \in V, \text{ such that } x(i,j) \right. \\
& \left. = \alpha \text{ and } \nexists\, l \in V \text{ such that } (l,i) \in A \right\} && \textbf{set } 2 \\
\bigcup & \left\{\$_i\alpha_i \mid \exists\, j \in V, \text{ such that } x(i,j) \right. \\
& \left. = \alpha \text{ and } \nexists\, l \in V \text{ such that } (l,i) \in A \right\}. && \textbf{set } 3
\end{aligned}
\tag{1}
$$

Any letter of a word in V' is a symbol of Σ' numbered by a node of V. Moreover, if that symbol is taken from V then it labels an edge of A that goes out a node, say i, of V, and the number associated to that symbol is i. In fact, V' is the union of three sets (see Eq. 1):
set 1 considers the cases of an edge of A labeled α followed by an edge labeled β, sets 2 and 3 contain the cases of an edge of A labeled α that is not preceded by another edge of A; for each such edge one creates two words: $\Delta_i\$_i$ in set 2 and $\$_i\alpha_i$ in set 3.

Let H be the 2-dBG of V'; note that Σ' is the alphabet of the words of V'. Now let z be the application from V' to Σ that for any α_i of V' satisfies $z(\alpha_i) = \alpha$. (Note that in this equation, the right term is a shortcut meaning the symbol of α_i without its numbering i; this shortcut is used only for the sake of legibility, but can be properly written with a heavier notation). Let $F' : \Sigma' \times \Sigma \to \mathbb{N}$ be the application such that $\forall(\alpha_i, \beta) \in \Sigma' \times \Sigma$, $F'(\alpha_i, \beta) = F(z(\alpha_i), \beta) = F(\alpha, \beta)$.

Let us show that this reduction is a bijection that transforms a positive instance of GRMP into a positive instance of DBGRMP. Assume there exists a path $p := (v_1, \dots, v_{|q|+1})$ of G which generates a word $m = m_1 \dots m_{|q|} \in \Sigma^{|q|}$ satisfying $m_i = x((v_i, v_{i+1}))$ and such that $\sum_{i=1}^{|q|} F(m_i, q_i) \leq t$. We show that there exists a path p' of G' which generates a word $m' = m'_1 \dots m'_{|q|} \in \Sigma'^{|q|}$ such that $\sum_{i=1}^{|q|} F'(m'_i, q_i) \leq t$.

We build the path p' as the "concatenation" of two paths, denoted p'_{start} and p'_{end}, that we define below. Let $\gamma_j := x((v_j, v_{j+1}))_{v_j} = (m_j)_{v_j}$ for all j between 1 and $|q|$. One has that $\gamma_j \in \Sigma'$. Now, let

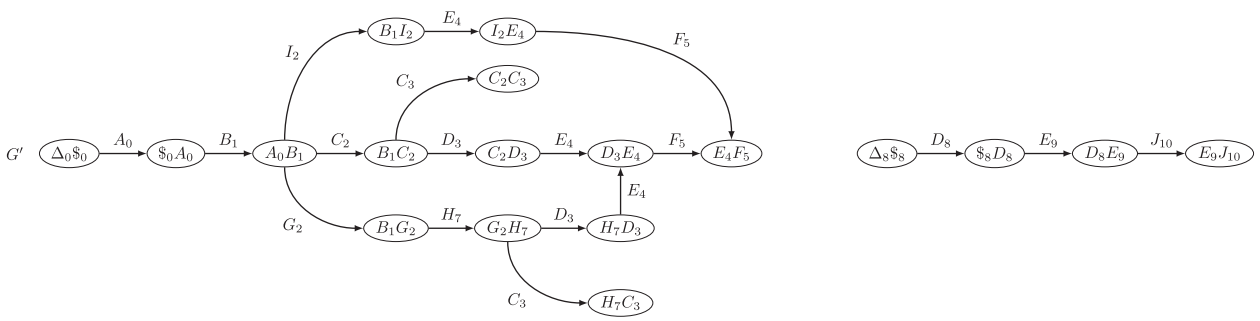

Fig. 1 Illustration of the gadget used in the proof of Theorem 1. Encoding a directed graph into a DBG of order 2. The directed graph G (*top*) admits the same words than the 2-DBG G' (*bottom*), if we ignore the numbers

$$p'_{start} := \begin{cases} \left(x((v_{l'},v_l))_{v_{l'}}x((v_l,v_1))_{v_l},\ x((v_l,v_1))_{v_l}x((v_1,v_2))_{v_1}\right) \\ \quad \text{if } \exists\ l,l' \in V \text{ such that } (l,1) \in A \text{ and } (l',l) \in A \\ \left(\$_{v_l}x((v_l,v_1))_{v_l},\ x((v_l,v_1))_{v_l}x((v_1,v_2))_{v_1}\right) \\ \quad \text{if } \exists\, l \in V \text{ such that } (l,1) \in A \text{ and } \nexists\ l' \in V \text{ such that } (l',l) \in A \\ \left(\Delta_{v_1}\$_{v_1},\ \$_{v_1}x((v_1,v_2))_{v_1}\right) \\ \quad \text{otherwise.} \end{cases}$$

and let

$$p'_{end} := \left(\gamma_1\gamma_2,\ \ldots,\ \gamma_{|q|-1}\gamma_{|q|}\right).$$

Let m' denote the word generated by p'. Clearly, one sees that $m' = (m_1)_{v_1}\ldots(m_{|q|})_{v_{|q|}}$, and since $m_i = z((m'_i)_{v_i})$, one gets that $z(m') = m$ and $\sum_{i=1}^{|q|} F'\left(m'_i, q_i\right) = \sum_{i=1}^{|q|} F\left(m_i, q_i\right) \leq t$.

In the other direction, the proof is similar since our construction is a bijection. □

GGMAP: a method to map reads on de Bruijn Graph

We propose a practical solution for solving DBGRMP. We consider the case of short (hundred of base pairs) reads with a low error rate (1 % of substitution), which is a good approximation of widely used NGS reads. Since errors are mostly substitutions, mapping is computed using the Hamming distance.

Our solution is designed for mapping on a compacted de Bruijn graph (CDBG) any set of short reads, either those used to build the graph or reads from another individual or species. We recall that a CDBG is representation of a DBG in which each non branching path is merged into a single node. The sequence of each node is called a *unitig*. Figure 2 shows a DBG and the associated CDBG.

In a CDBG, the nodes are not necessarily k-mers, words of length k, but *unitigs*, with some unitigs being longer than reads. Thus, while mapping on a CDBG, one distinguishes between two mapping situations: **i**/ the reads mapping completely on a unitig of the graph, and **ii**/ the reads whose mapping spans two or more unitigs. For the latter, we say that the read *maps on a branching path of the graph*.

Taking advantage of the extensive research carried out for mapping reads on flat strings, *GGMAP* uses Bowtie2 [7] to map the reads on the unitigs. In addition, *GGMAP* integrates our proposed new tool, called *BGREAT*, for mapping reads on branching paths of the CDBG. Figure 3 provides an overview of the pipeline.

GGMAP takes as inputs a query set of reads and a reference DBG. To avoid including sequencing errors in the DBG, we construct the reference DBG after filtering out all k-mers whose coverage lies below a user-defined threshold c. This error removal step is a classical preprocessing step that is performed in k-mer based assemblers. The unitigs of the CDBG are computed using *BCALM*2 (the parallel version of *BCALM* [21]), using the k-mers having a coverage $\geq c$. *GGMAP* uses such a set of unitigs as DBG.

We now propose a detailed description of *BGREAT*.

BGREAT: mapping reads on branching paths of the CDBG

As previously mentioned, *BGREAT* is designed for mapping reads on branching paths of a CDBG, using reasonable resources both in terms of time and memory. Our approach follows the usual "seed and extend" paradigm. More generally, the proposed implementation applies heuristic schemes, both regarding the indexing and the alignment phases.

Indexing heuristic

We remind that our algorithm maps reads that span at least two distinct unitigs. Such mapped reads inevitably traverse one or more DBG edge(s). In a CDBG, edges are represented by the prefix and suffix of size $k-1$ of each unitig. We call such sequences the *overlaps*. In order to limit the index size and the computation time, our algorithm indexes only overlaps that are later used as seeds. Those overlaps are good anchors for several reasons: they are long enough ($k-1$) to be selective, they cannot be shared by more than eight unitigs (four starting and four ending with the overlap), and a CDBG usually has a reasonable number of unitigs and then of overlaps. For

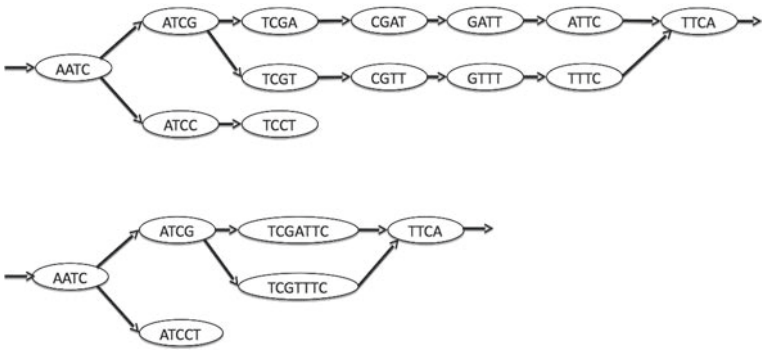

Fig. 2 A toy example of a DBG of order k with $k = 4$ (*top*) and its compacted version (*bottom*)

instance, the CDBG in our experiment with human data has 70 million unitigs and 87 million overlaps for 3 billion k-mers). In our implementation, the index is a minimal perfect hash table indicating for each overlap the unitig(s) starting or ending with this $(k-1)$-mer. Using a minimal perfect hash function limits the memory footprint, while keeping efficient query times (see Table 3).

Read alignment

Given a read, each of its $k-1$-mers is used to query the index. The index detects which $k-1$-mers represent an overlap of the CDBG. An example of a read together with the matched unitigs are displayed on Fig. 4. Once the overlaps and their corresponding unitigs have been computed, the alignment of the read is performed from left to right as presented in Algorithm 1. Given an overlap position i on the read, the unitigs starting with this overlap are aligned to the sequence of the read starting from position i. The best alignment is recorded. In addition, to improve speed, if one of the at most four unitigs ending with the same overlap is the next overlap detected on the read, then this unitig is tested first, and if the alignment contains less mismatch than the user defined threshold, the other unitigs are not considered. Note that this optimization does not apply for the first and last overlaps of a read.

Algorithm 1: Greedy algorithm for mapping a read on multiple unitigs once the potential overlaps present in the read have been detected.

```
Data: Read r, Integer n
for the n first overlaps of r do
    Find a path begin that map the begin of r
    if begin found then
        for the n last overlaps of r do
            Find a path end that maps the end of r
            if end found then
                Find (in a greedy way) a path cover that
                map the read from begin to end
                if cover found then
                    write path;
                    return
```

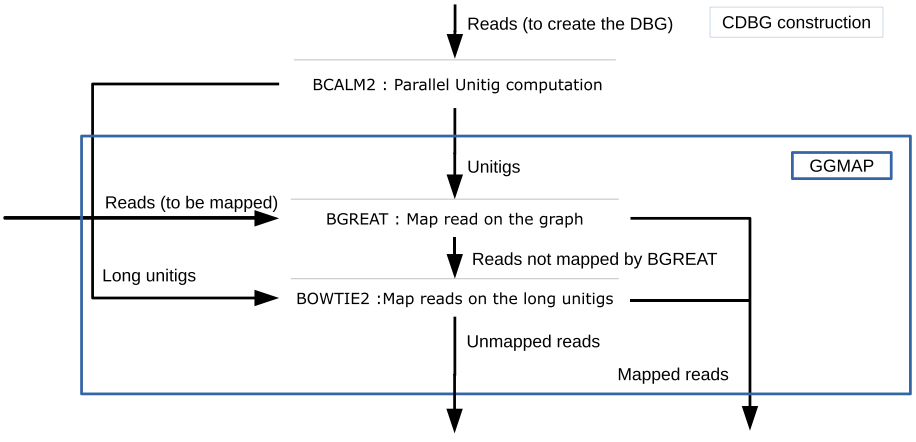

Fig. 3 Unitig construction, as used in the proposed experiments (*upper part* of the figure) and *GGMAP* pipeline. Reads to be mapped can be distinct from reads used for building the graph. Long unitigs are unitigs longer than the reads. We remind that tools BCALM and BOWTIE2 are respectively published in [7, 21]

Fig. 4 Representation of the mapping of a read (top sequence) on a CDBG, whose nodes are represented on lines 2, 3, and 4. (step 1) the overlaps of the graph that are also present in the read are found (here *TACAC*, *GCTGC*, and *AGCTA*, represented on line 1). (step 2) unitigs that map the beginning and the end of the read are found (those represented on line 2). (step 3) cover the rest of the read, guided by the overlaps (here with unitigs represented on lines 3 and 4)

This mapping procedure is performed only if the two extremities of the read are mapped by two unitigs. The extreme overlaps of the read enables BGREAT to quickly filter out unmappable reads. For doing this, the first (resp. last) overlap of the read is used to align the read to the first (resp. last) unitig. Note that, as polymorphism exists between the read and the graph, some of the overlaps present on the read may be spurious. In this case the alignment fails, and the algorithm continues with the next (resp. previous) overlap. At most n alignment failures are authorized in each direction. If a read cannot be anchored neither on the left, nor on the right, it is considered as not aligned to the graph.

Note that the whole approach is greedy: given two or more possible choices, the best one is chosen and backtracking is excluded. This results in a linear time mapping process, since each position in the read can lead to a maximum of four comparisons, and the algorithm continues as long as the cumulated number of mismatches remains below the user defined threshold. Because of heuristics, a read may be unmapped or wrongly mapped for any of the following reasons.

- All overlaps on which the read should map contain errors, in this case the read is not anchored or only badly anchored and thus not mapped.
- The n first or n last overlaps of the read are spurious, in this case the *begin* or *end* is not found and the read is not mapped. By default and in all experiments $n = 2$.
- The greedy choices made during the path selection are wrong.

We implemented *BGREAT* as a dependence-free tool in C++ available at github.com/Malfoy/BGREAT.

Results

Beforehand we give details about the data sets (Subsection Data sets and CDBG construction), then we perform several evaluations of *GGMAP* and of *BGREAT*. First, we compare graph mapping to mapping on the contigs resulting from an assembly (Subsection Graph mapping vs assembly mapping). Second, we assess how many reads are mapped on branching paths vs on unitigs (Subsection Mapping on branching paths usefulness). Third, we evaluate the efficiency of BGREAT in both terms of throughput and scalability (Subsection *GGMAP* perfor mances), then assess the quality of the mapping itself (Subsection *GGMAP* accuracy). All *BGREAT* alignments were performed authorizing up to two mismatches.

There are very few published tools to compare *GGMAP* with. Indeed, we found only one published tool, called *BlastGraph* [19], which was designed for mapping reads on a DBG. However, on our simplest data set coming from the *E.coli* genome (see Table 1), *BlastGraph* crashed after $\approx$ 124 h of computation. Thus, *BlastGraph* was not further investigated here.

Data sets and CDBG construction

For our experiments we used publicly available Illumina read data sets from species of increasing complexity: from the bacterium *E.coli*, the worm *C.elegans*, and from Human. Detailed information about the data sets are given in Additional file 1: Table S1 (identifiers, read length, read numbers, and coverages – from 70x to 112x–).

For each of these three data sets, we generated a CDBG using BCALM. From the *C.elegans* read set, we additionally generated an artificially complex graph, by using small k and c values (respectively 21 and 2). This particular graph, called *C.elegans_cpx*, contains lot of small unitigs. We used it to assess situations of highly complex and/or low quality sequencing data. The characteristics of the CDBG obtained on each of these data sets are given in Table 1.

Graph mapping vs assembly mapping

We compared *GGMAP* to the popular approach consisting in mapping the reads to the reference contigs computed by an assembler. For testing this approach, for each of the three sets used, we first assembled them and then we mapped back the reads on the obtained set of contigs. We used two different assemblers, the widely used Velvet [22], and Minia [23], a memory efficient assembler based on Bloom filters. Finally, we used Bowtie2 for mapping the reads on the obtained contigs.

The results reported in Table 2 show that the number of reads mapped on assembled contigs is smaller than the one obtained with *GGMAP*. We obtained similar results in terms of number of reads mapped on the

Table 1 CDBG used in this study

CDBG Id	Reads Id	*k*	*c*	Number of unitigs	Mean length of unitigs
E.coli	SRR959239	31	3	42,843	134
*C.elegans*_norm	SRR065390	31	3	1,627,335	93
*C.elegans*_cpx	SRR065390	21	2	8,273,338	34
Human	SRR345593 SRR345594	31	10	69,932,343	70

*C.elegans*_cpx and *C.elegans*_norm are two distinct graphs, constructed using the same read set from *C.elegans* genome. The suffixes *norm* and *cpx* respectively stand for "normal" (using $c = 3$ and $k = 31$) and for "complex" (using a low threshold $c = 2$ and small value $k = 21$)

assemblies yielded by Velvet and Minia (see Additional file 1: Table S2). Let us emphasize that on the Human dataset, *GGMAP* maps 22 additional percents of reads on the graph than Bowtie2 does on the assembly.

We notice that the more complex the graph, the higher the advantage of mapping on the CDBG. This is due to the inherent difficulty of assembling with huge and highly branching graphs. This is particularly prominent in the results obtained on the artificially complex *C.elegans_cpx* CDBG.

We also highlight that our approach is resource efficient compared to most assembly processes. For instance, Velvet used more than 80 gigabytes of memory to compute the contigs for the *C. elegans* data set with $k = 31$. On this data set, our workflow used at most 4 GB memory (during k-mer counting). In terms of throughput, using *BGREAT* and then Bowtie2 on long unitigs is comparable to using Bowtie2 on contigs alone. See Section *GGMAP* perfor mances for more details about *GGMAP* performances.

Mapping on branching paths usefulness

Mapping the reads on branching paths of the graph is not equivalent to simply mapping the reads on unitigs. Indeed, at least 13 % of reads (mapping reads SRR959239 on the *E.coli* DBG) and up to 66 % of reads (mapping reads SRR065390 on *C.elegans_cpx* DBG) map on the branching paths of the graph (see Fig. 5). These reads cannot be mapped when using only the set of unitigs as a reference. As expected, the more complex the graph, the larger the benefit of *BGREAT*'s approach. On the complex *C.elegans_cpx* graph, only 23 % of reads can be fully mapped on unitigs, while 89 % of them are mapped by additionally using *BGREAT*. On a simpler graph as *C.elegans_norm* the gap is smaller, but remains significant (72 vs 93 %). Complete mapping results are shown in Additional file 1: Table S3.

Table 2 Percentage of mapped reads, either mapping on contigs (here obtained thank to the Minia assembler) or mapping on CDBG with *GGMAP*

Set	% mapped on contigs	% mapped on CDBG
E.coli	95.57	97.16
C.elegans_norm	80,60	93,24
C.elegans_cpx	56,33	89,15
Human	63,16	85,70

Non reflexive mapping on a CDBG

The *GGMAP* approach is also suitable for mapping a distinct read set from the one used for constructing the DBG. We mapped another read set from *C.elegans* (SRR1522085) on the *C.elegans_norm* CDBG. Results in this situation are similar to those observed when performing reflexive mapping (i.e., when mapping the reads used to construct this graph): among 89 % of mapped reads, 15 % were mapped on branching paths of the graph (See Fig. 5).

GGMAP performances

Table 3 presents *GGMAP* time and memory footprints. It shows that *BGREAT* is very efficient in terms of throughput while using moderate resources. Presented heuristics and implementation details allow *BGREAT* to scale up to real-world instances of the problem, being able to map millions of reads per CPU hour on a Human CDBG with a low memory footprint. *BGREAT* mapping is parallelized and can efficiently use dozens of cores.

GGMAP accuracy

To measure the impact of the read alignment heuristics, we forced the tool to explore exhaustively all potential alignment paths once a read is anchored on the graph. Results on the *E.coli* dataset show that the greedy approach is much faster than the exhaustive one (38× faster), while the mapping capacity is little impacted: the overall number of mapped reads increases by only 0.03 % with the exhaustive approach. We thus claim that the choice of the greedy strategy is a satisfying trade-off.

To further evaluate the *GGMAP* accuracy, we assess the recall and mapping quality in the following experiment. We created a CDBG from Human chromosome 1 (hg19 version). Thus, each k-mer of the chromosome appears in the graph. Furthermore, from the same sequence, we

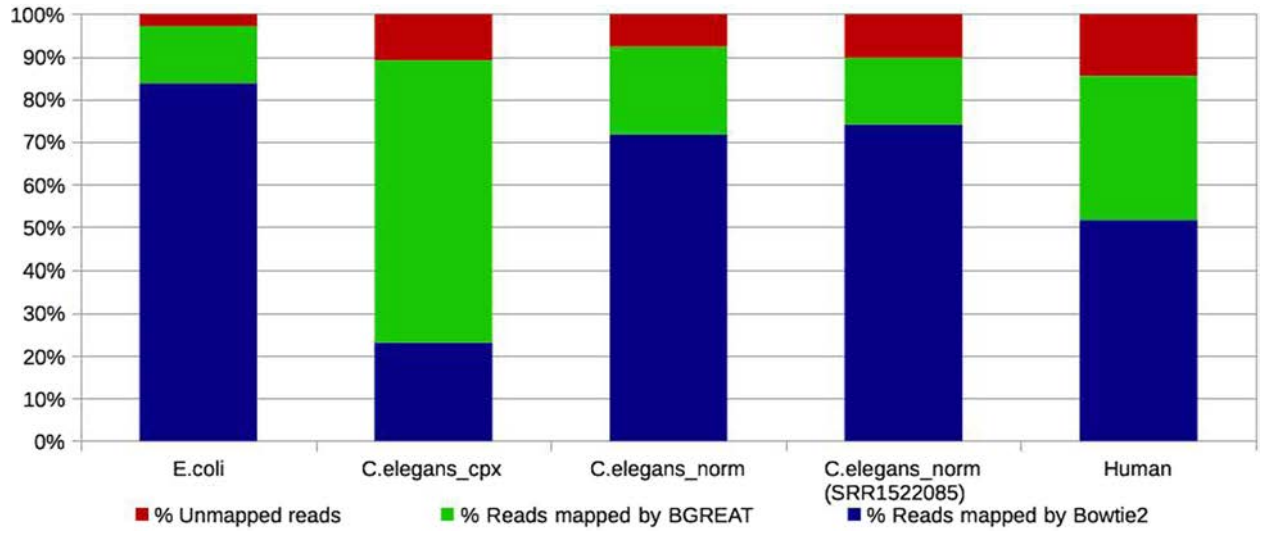

Fig. 5 *GGMAP* mapping results for the different read sets. In the "*C.Elegans_norm* (SRR1522085)" case, reads from SRR1522085 are mapped on the CDBG obtained using reads from read set SRR065390. For all other results, the same read set was used both for constructing the CDBG and during the mapping

simulated reads with distinct error rates (0, 0.1, 0.2, 0.5, 1 and 2 %). For each error rate value, we generated one million reads. We evaluated the *GGMAP* results by mapping the simulated reads on the graph. As the graph is error free, except in some rare cases due to repetitions, the differences between a correctly mapped read and the path it maps to in the graph occur at erroneous positions of the read. If this is not the case, we say that the read is not mapped at its optimal position. Among the error free positions of a simulated read, the number of mismatches observed between this read and the mapped path is called the "*distance to optimal*". Results are reported in Table 4 together with the obtained recall (number of mapped reads over the number of simulated reads). Those results show the limits of *BGREAT* while mapping reads from divergent individuals. With 2 % of substitutions in reads, only 90.85 % of the reads are perfectly mapped. Nevertheless, with this divergence rate, 97.28 % of reads are mapped at distance at most one from optimum. With over 99 % of perfectly mapped reads, these results show that with the current sequencing characteristics, i.e. a 0.1 % error rate, the mapping accuracy of *BGREAT* is suitable for most applications.

Discussion

We proposed a formal definition of the de Bruijn graph Read Mapping Problem (DBGRMP) and proved its NP-completeness. We proposed a heuristic algorithm offering

Table 3 Time and memory footprints of *BGREAT* and BOWTIE2

		BGREAT			BOWTIE2		
CDBG Id	Mapped set (nb reads)	Wall clock time	CPU time	Memory	Wall clock time	CPU time	Memory
E.coli	SRR959239 (5,128,790)	28 s	1m40	19 MB	1m17	3m53	29 MB
C.elegans_cpx	SRR065390 (67,155,743)	19m21	72m31	975 MB	8m12	33m	1.66 GB
C.elegans_norm	"	13m03	51m28	336 MB	17m49	72m31	493 MB
C.elegans_norm	SRR1522085 (22,509,110)	1m54	7m13	336 MB	3m29	14m12	493 MB
Human	SRR345593 SRR345594 (2,967,536,821)	4h30	87 h	9.7 GB	4h38	90h15	21 GB

Indicated wall clock times use four cores, except for the human samples for which 20 cores were used

Table 4 *GGMAP* mapping results on simulated reads from the reference of the human chromosome 1 with default parameters

% Errors in simulated reads	Distance to optimum of *BGREAT* mapped reads (percentage)				
	0	1	2	3	≥ 4
0	100	0	0	0	0
0.1	99.31	0.52	0.09	0.04	0.04
0.2	98.79	0.91	0.21	0.07	0.02
0.5	97.2	2.17	0.41	0.17	0.05
1	94.88	3.72	0.92	0.41	0.07
2	90.85	6.43	1.79	0.83	0.1

Results show the recall of *GGMAP* and the quality of *BGREAT* mapping, as represented by the "distance to optimum" value. For instance 94.88 % of the reads were mapped without error, 3.72 % were mapped with a distance to the optimum of one etc. Due to approximate repeats in human chromosome 1, the reported distance to optimum is an upper bound

a practical solution. We developed a tool called *BGREAT* implementing this algorithm using a compacted de Bruijn graph (CDBG) as a reference.

From the theoretical viewpoint, the problem DBGRMP considers paths rather than walks in the graph. The current proof of its hardness does not seem to be adaptable to the cases of walks. A perspective is to extend the hardness result to that more general case.

We emphasize that our proposal does not enable genome annotation. It has been designed for applications aiming at a precise quantification of sequenced data, or a set of potential variations between the reads and the reference genome. In this context, it is essential to map as much reads as possible. Experiments show that a significant proportion of the reads (between $\approx$ 13and $\approx$ 66 % depending on the experiment) can be only mapped on branching paths of the graph. Hence, mapping only on the nodes of the graph or on assembled contigs is thus insufficient. This statement holds true when mapping the reads used for building the graph, but also with reads from a different experiment. Moreover, our results show that a potentially large number of reads (up to $\approx$ 32%) that are mapped on a CDBG cannot be mapped on a classical assembly.

With *GGMAP*, the mapping quality is very high: using Human chromosome 1 as a reference and reads with a realistic error rate (similar to that of Illumina technology), over 99 % of the reads are correctly mapped. The same experiment also pointed out the limits of mapping reads on a divergent graph reference ($\geq$ 2 % substitutions): approximately 10 % of the reads are mapped at a suboptimal position.

A weak point of *BGREAT* lies in its anchoring technique. Reads mapped with *BGREAT* must contain at least one exact $k-1$-mer that is an arc of the CDBG, *i.e.*, an overlap between two connected nodes. This may be a serious limitation when the original read set diverges greatly from the reads to be mapped. Improving the mapping technique may be done by using not only unitig overlaps as anchors at the cost of higher computational resources. Another solution may consist in using a smarter anchoring approach, like spaced seeds, which can accommodate errors in the anchor [24].

A natural extension consists in adapting *BGREAT* for mapping, on the CDBG obtained from short reads, the long (a few kilobases in average) and noisy reads produced by the third generation of sequencers, whose error rate reaches up to 15 % (with mostly insertion and deletion errors for e.g. Pacific Biosciences technology). Such adaptation is not straightforward because of our seeding strategy, which requires long exact matches. The anchoring process must be very sensitive and very specific, while the mapping itself must implement a Blast-like heuristic or an alignment-free method. However, mapping such long reads on a DBG could be of interest for correcting these reads as in [13], or for solving repeats, if long reads are mapped on the walks (which main include cycles) of the DBG. Our NP-completeness proof only considers mapping on (acyclic) paths. Proving the hardness of the problem of mapping reads on walks of a DBG remains open.

Incidentally, using the same read set for constructing the CDBG and for mapping opens the way to major applications. Indeed, the graph and the exact location of each read on it may be used for **i/** read correction as in [15], by detecting differences between reads and the mapped area of the graph in which low support k-mers likely due to sequencing errors are absent, or for **ii/** read compression by recording additionally the mapping errors, or for **iii/** both correction and compression by conserving only for each read its mapping location on the graph.

Having for each read (used for constructing the graph or not) its location on the CDBG also provides the opportunity to design algorithms for enriching the graph, for instance enabling a quantification that is sensitive to local variations. This would be valuable for applications such as variant calling, analysis of RNA-seq variants [25], or of metagenomic reads [26].

Additionally, *BGREAT* results provide pieces of information for distant k-mers in the CDBG, about their co-occurrences in the mapped read data sets. This offers a way for the resolution, in the de Bruijn graph, of repeats larger than k. It could also allow to phase the polymorphisms and to reconstruct haplotypes.

Conclusion

A take home message is that read mapping can be significantly improved by mapping on the structure of an assembly graph rather than on a set of assembled contigs (respectively $\approx$22 % and $\approx$ 32 % of additional reads mapped for the Human and a complex *C.elegans* data

sets). This is mainly due to the fact that assembly graphs retains more genomic information than assembled contigs, which also suffer from errors induced by the complexity of assembly. Moreover, mapping on a compacted De Bruijn Graph can be fast. The availability of *BGREAT* opens the door to its application to fundamental tasks such as read error correction, read compression, variant quantification, or haplotype reconstruction.

Additional file

Additional file 1: Read mapping on De Bruijn graphs additional file. Three complementary tables are presented. Main characteristics of data sets used in this study. Assembly and mapping approach comparison. Results of BGREAT on real read sets. (PDF 40 kb)

Abbreviations

CDBG, Compacted De Bruijn graph; DBG, De Bruijn graph; DBGRMP, De Bruijn graph read mapping problem; FLATSP, fixed length assymetric travelling salesman problem; GRMP, graph read mapping problem; HPP, Hamiltonian path problem

Acknowledgements

We would like to thank Yannick Zakowski, Claire Lemaitre and Camille Marchet for proofreading the manuscript and discussions.

Funding

This work was funded by French ANR-12-BS02-0008 Colib'read project, by ANR-11-BINF-0002, and by a MASTODONS project.

Availability of data and materials

Our implementations are available at github.com/Malfoy/BGREAT. In addition to the following pieces of information, Additional file 1: Table S1 presents the main characteristics of these datasets.
SRR959239 http://www.ncbi.nlm.nih.gov/sra/?term=SRR959239
SRR065390 http://www.ncbi.nlm.nih.gov/sra/?term=SRR065390
SRR1522085 http://www.ncbi.nlm.nih.gov/sra/?term=SRR1522085
SRR345593 and SRR345594 http://www.ncbi.nlm.nih.gov/sra/?term=SRR345593.

Authors' contributions

PP initiated the work and designed the study. AL, BC and ER designed the formalism and the proofs of NP-hardness. AL designed the algorithmic framework, implemented the *BGREAT* and performed the tests. All authors wrote and accepted the final version of the manuscript.

Competing interests

The authors declare that they have no competing interests.

Ethics approval and consent to participate

Not applicable.

Author details

[1]IRISA Inria Rennes Bretagne Atlantique, GenScale team, Campus de Beaulieu, 35042 Rennes, France. [2]L.I.R.M.M., UMR 5506, Université de Montpellier et CNRS, 860 rue de St Priest, F-34392 Montpellier Cedex 5, France. [3]Institut Biologie Computationnelle, Université de Montpellier, F-34392 Montpellier, France.

Received: 9 December 2015 Accepted: 26 May 2016
Published online: 16 June 2016

References

1. Bradnam KR, Fass JN, et al. Assemblathon 2: evaluating de novo methods of genome assembly in three vertebrate species. GigaScience. 2013;2:10. doi:10.1186/2047-217X-2-10.
2. Nagarajan N, Pop M. Parametric complexity of sequence assembly: theory and applications to next generation sequencing. J Comput Biol. 2009;16(7):897–908. doi:10.1089/cmb.2009.0005.
3. Chaisson MJ, Pevzner PA. Short read fragment assembly of bacterial genomes. Genome Res. 2008;18(2):324–30. doi:10.1101/gr.7088808.
4. Myers EW, Sutton GG, et al. A whole-genome assembly of Drosophila. Science (New York, N.Y.) 2000;287(5461):2196–204. doi:10.1126/science.287.5461.2196.
5. Myers EW. The fragment assembly string graph. Bioinformatics. 2005;21(Suppl 2):79–85. doi:10.1093/bioinformatics/bti1114.
6. Li H, Durbin R. Fast and accurate short read alignment with Burrows-Wheeler transform. Bioinformatics. 2009;25(14):1754–60. doi:10.1093/bioinformatics/btp324.
7. Langmead B, Salzberg SL. Fast gapped-read alignment with Bowtie 2. Nat Methods. 2012;9(4):357–9. doi:10.1038/nmeth.1923.
8. Lee H, Schatz MC. Genomic dark matter: the reliability of short read mapping illustrated by the genome mappability score. Bioinformatics. 2012;28(16):2097–105. doi:10.1093/bioinformatics/bts330.
9. Deshpande V, Fung EDK, Pham S, Bafna V. Cerulean: A Hybrid Assembly Using High Throughput Short and Long Reads. In: Lecture Notes in Computer Science vol. 8126 LNBI. Springer; 2013. p. 349–63. doi:10.1007/978-3-642-40453-5_27.
10. Ribeiro FJ, Przybylski D, Yin S, Sharpe T, Gnerre S, Abouelleil A, Berlin AM, Montmayeur A, Shea TP, Walker BJ, Young SK, Russ C, Nusbaum C, MacCallum I, Jaffe DB. Finished bacterial genomes from shotgun sequence data. Genome Res. 2012;22(11):2270–7. doi:10.1101/gr.141515.112.
11. Pevzner PA, Tang H, Waterman MS. An Eulerian path approach to DNA fragment assembly. Proc Natl Acad Sci. 2001;98(17):9748–53. doi:10.1073/pnas.171285098.
12. Bankevich A, Nurk S, Antipov D, Gurevich AA, Dvorkin M, Kulikov AS, Lesin VM, Nikolenko SI, Pham S, Prjibelski AD, Pyshkin AV, Sirotkin AV, Vyahhi N, Tesler G, Alekseyev MA, Pevzner PA. SPAdes: a new genome assembly algorithm and its applications to single-cell sequencing. J Comput Biol. 2012;19(5):455–77. doi:10.1089/cmb.2012.0021.
13. Salmela L, Rivals E. LoRDEC: accurate and efficient long read error correction. Bioinformatics. 2014;30(24):3506–14. doi:10.1093/bioinformatics/btu538.
14. Yang X, Chockalingam SP, Aluru S. A survey of error-correction methods for next-generation sequencing. Brief Bioinform. 2013;14(1):56–66. doi:10.1093/bib/bbs015.
15. Benoit G, Lavenier D, Lemaitre C, Rizk G. Bloocoo, a memory efficient read corrector. In: European Conference on Computational Biology (ECCB); 2014. https://gatb.inria.fr/software/bloocoo/.
16. Wang M, Ye Y, Tang H. A de Bruijn graph approach to the quantification of closely-related genomes in a microbial community. J Comput Biol. 2012;19(6):814–25. doi:10.1089/cmb.2012.0058.
17. Huang L, Popic V, Batzoglou S. Short read alignment with populations of genomes. Bioinformatics. 2013;29(13):361–70. doi:10.1093/bioinformatics/btt215.
18. Dilthey A, Cox C, Iqbal Z, Nelson MR, McVean G. Improved genome inference in the MHC using a population reference graph. Nat Genet. 2015;47(6):682–8. doi:10.1038/ng.3257.
19. Holley G, Peterlongo P. Blastgraph: Intensive approximate pattern matching in sequence graphs and de-bruijn graphs. In: Stringology; 2012. p. 53–63. http://alcovna.genouest.org/blastree/.
20. Karp RM. Reducibility Among Combinatorial Problems. In: 50 Years of Integer Programming 1958-2008. Berlin, Heidelberg: Springer; 2010. p. 219–41. doi:10.1007/978-3-540-68279-0_8. http://link.springer.com/10.1007/978-3-540-68279-0_8.
21. Chikhi R, Limasset A, Jackman S, Simpson JT, Medvedev P. On the representation of de bruijn graphs. In: Research in Computational Molecular Biology. Springer; 2014. p. 35–55. doi:10.1007/978-3-319-05269-4-4.
22. Zerbino DR, Birney E. Velvet: algorithms for de novo short read assembly using de Bruijn graphs. Genome Res. 2008;18(5):821–9. doi:10.1101/gr.074492.107.
23. Chikhi R, Rizk G. Space-efficient and exact de Bruijn graph representation based on a Bloom filter. Algorithm Mol Biol. 2013;8(1):22. doi:10.1186/1748-7188-8-22.
24. Vroland C, Salson M, Touzet H. Lossless seeds for searching short patterns with high error rates. In: Combinatorial Algorithms. Springer; 2014. p. 364–75.

25. Sacomoto GA, Kielbassa J, Chikhi R, Uricaru R, Antoniou P, Sagot MF, Peterlongo P, Lacroix V. Kissplice: de-novo calling alternative splicing events from rna-seq data. BMC Bioinformatics. 2012;13(Suppl 6):5.
26. Ye Y, Tang H. Utilizing de Bruijn graph of metagenome assembly for metatranscriptome analysis. Bioinformatics. 2015btv510. Oxford Univ Press. arXiv preprint arXiv:1504.01304.

Chapter 4

Improving an assembly paradigm

In this chapter we are interested in improving the de Bruijn graph assembly. We first describe the limitations of actual assembly methods and propose a high order de Bruijn graph as a scalable and efficient solution. Then we propose a proof of concept method to construct such graph based on successive mapping. In the last part, we present different results of our proof of concept assembler and show its potential to tackle several existing limitations.

4.1 Assembly challenges

4.1.1 Two assembly paradigms

As we previously saw, OLC and string graph approaches are able to use the full read information. They allow better repeats handling than the de Bruijn graph and produce more contiguous assemblies. From a qualitative point of view, the best assemblers should be those based on string graphs, as assessed by the Assemblathon assembly competition [122] [123]. However, in the introduction, we presented those approaches as intractable because of the need to index the very large reads sets and to compute pairwise alignments. Large genomes are costly to assemble, and alternative approaches have been preferred for scalability reasons. Contrary to string graphs, despite being tractable, the de Bruijn graph do not use full read information and produce fragmented assembly. Moreover, high order de Bruijn graphs are hard to obtain because they require a high sequencing depth.

4.1.2 Why read length matters: polyploid assembly

The major practical difference between the de Bruijn graph and the string graph is thus the sequence length information they use. Having full length helps solve repeats, but since read lengths are bounded and particularly short for NGS, the theoretical edge of the string graph over the de Bruijn graph is not major. De Bruijn graph assemblers are not really behind string graph in practice, they can use an order above 60 from a high coverage, when the reads length goes up to a few hundred. Then, the advantages of the string graphs will come in the range of repeats between k and the read length. We mentioned in the introduction that *C. elegans* genome presents 1,892,518 repeated 61-mers and 824,254 repeated 301-mers. Even a string graph using 300 bp reads could not solve those repeats. But in some cases the number of short repeats can become extremely high, for instance in diploid genomes. As mentioned before, the variations between two alleles create bubble patterns in assembly graphs. Yet, according to the heterozygosity rate of a given genome, the three following situations may be distinguished.

- Low heterozygosity rate ($< 0.1\%$): most variant are isolated from the others and each appears as a bubble in the graph (Figure 3.3). Most nodes of the graph represent homozygous regions of the underlying genome.
- High heterozygosity rate($> 5\%$): the divergence between the sequences is so high that the two alleles are mostly separated in the assembly graph (for example with a high k as seen in the Figure 4.1). Nodes of the graph mostly represent heterozygous

regions, they yield large bubbles that relatively well represent and differentiate the diverging alleles.

- Medium heterozygosity rate ($\approx 1\%$): present a mix of heterozygous and homozygous regions that may be difficult to distinguish. It results into complex graphs, hard to produce long contigs in practice.

The heterozygosity rates given are rather orders of magnitude than accurate values (Figures 4.5 and 4.6). Furthermore, these rates are based on the use of hundred of bases NGS reads. If we were able to produce longer reads, we could use a higher k and a heterozygosity rate of 1% would be easier to manage.

Heterozygosity rates vary a lot across species as seen in Figure 4.2 from [124]. From our current knowledge, one of the lowest rates is present in Lynx individuals that have a rate of the order of 0.01%, where the heterozygous events are likely to be far apart. On the opposite, in species like *C. remanei*, with a rate of 5%, the variations should be quite close to each other on average and therefore a large order de Bruijn graph would be able to differentiate two alleles in the heterozygous regions. Most de Bruijn graph assemblers present a "comfort zone" below the rate of 0.5%, where they are able to produce large contigs by crushing bubbles. Medium rates (around 1%) result in extremely poor assemblies. As mentioned, very high rates ($> 5\%$) genomes are manageable, although we currently count few of them among known assembled species.

4.1.3 Taking the best of both worlds

In any case, haploid or polyploid, the faculty to solve repeats is the key to more contiguous assemblies. Even if this problem is dramatically more frequent in polyploid assembly, enhancing this capacity would positively impact any assembly problem. We explained that each presented graph structure shows advantages as well as drawbacks. The question we should ask is "What kind of graph structure would we ideally want ?"

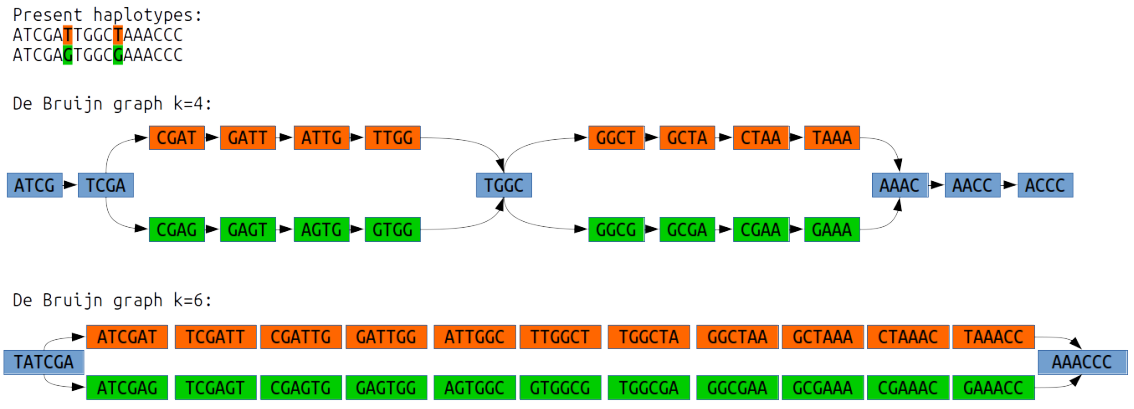

Figure 4.1: Example of phased and non-phased variants depending on k. With $k = 4$ the graph presents two bubbles, and the assembler has no mean to output phased contigs. With $k = 6$ the variants are phased and the graph is reduced into a single heterozygous region of two contigs.

Figure 4.2: Heterozygosity rates across species [124].

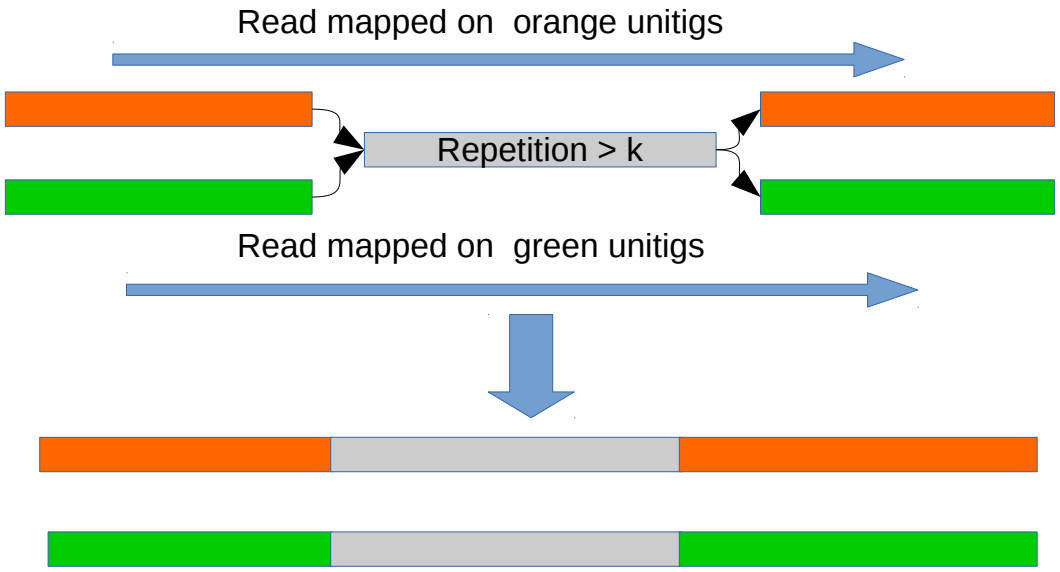

Figure 4.3: The general idea to improve the de Bruijn graph using read length. The reads information is used here in order to "solve" the gray repeat, *ie* to determine the possible contexts of this repeats into contigs. Here two contexts exist according to reads, the orange-gray-orange contig an the green-gray-green.

Preferably we look for a graph:

- Using the whole read information, in order to get events phased in the reads also phased in the graph (not a de Bruijn graph)
- Easy to compute (not a string graph)

While several contributions improved the string graphs scalability, the de Bruijn graph assemblers main drawbacks was also subject to improvements. The fact that most people used de Bruijn graph assemblers led to new methods to improve the repeat resolution of such graphs, in order to produce less fragmented assemblies [39, 125]. The global idea is represented in the Figure 4.3. We have a graph made of trusted sequences, built from reads kmers. We are interested in adding back the read information in order to simplify some situations. The questions about how to make the best use of such information (how to produce longer contigs according the read information) and how to obtain it (how to map reads on the graph) has not reached a consensus yet.

A simple solution would be to construct a very high order de Bruijn graph with a k value near the size of the reads. But such a graph would require a tremendous coverage as shown in Table 4.1.

This raises the question: "How to construct a high order de Bruijn graph without a massive coverage ?" Several assemblers have tried to improve their repeat resolution power by constructing a high order De Bruin graph. Tools like IDBA [39] or SPAdes [36] tackle this question by using multiple kmer sizes approaches. The idea is to use several de Bruijn graph with different orders, to be able to detect all overlaps with the low k values, and solve some repeats with a high k value. Each of those techniques have shown how to improve assemblies in a consistent way, even if they present drawbacks that may be addressed. In the following we propose a new way to use the multiple kmer idea to

Coverage	k=240	k=230	k=220
100X	67	44	27
100X filter	94	81	63
200X	45	18	6
200X filter	81	50	24
500X	14	1	0
500X filter	42	8	1
1000X	2	0	0
1000X filter	9	0	0

Table 4.1: Percentage of missing kmers in a graph created with 250bp reads simulated from a small virus genome of 48,502 base pairs called Lambda phage with 1% error rate. The filtering consisted in removing kmers with an abundance of one. Even a tremendous coverage does not allow to construct directly a high order de Bruijn graph.

build a very high order de Bruijn graph based on the techniques previously proposed in this document.

"Assembly challenges" core messages:

- Using the whole length of the reads can reduce assembly fragmentation
- This is especially important in polyploid assembly
- A high order de Bruijn graph would be a scalable structure very similar to the string graph

4.2 Building very high order de Bruijn graph

In this section we propose a new method using the previously described tools able to construct very high order de Bruijn graphs without the need of a costly coverage.

4.2.1 Successive mapping strategy

We have shown previously that the mapping of the reads on the de Bruijn graph produced very accurate corrected reads. Such reads with their lower error rates could be used in order to build a higher order de Bruijn graph. However we can see unitigs as trusted genome sequences that we want to order. Our hypothesis is that a read mapping on a path of unitigs validates their ordering. It means that a mapped read validates the whole path and not only the nucleotides where the read actually maps. As the Masurca assembler [126], we call super-read (SR) the paths of unitigs that correspond to a read mapping. An interesting property of a SR is that its length is superior (or equal) to its associated read because it may be extended by the first and last unitig sequences (Figure 4.4). Unlike unitigs, that can be smaller than reads in complex regions, the SR are guaranteed to be at least the size of the reads, and most of the time larger.

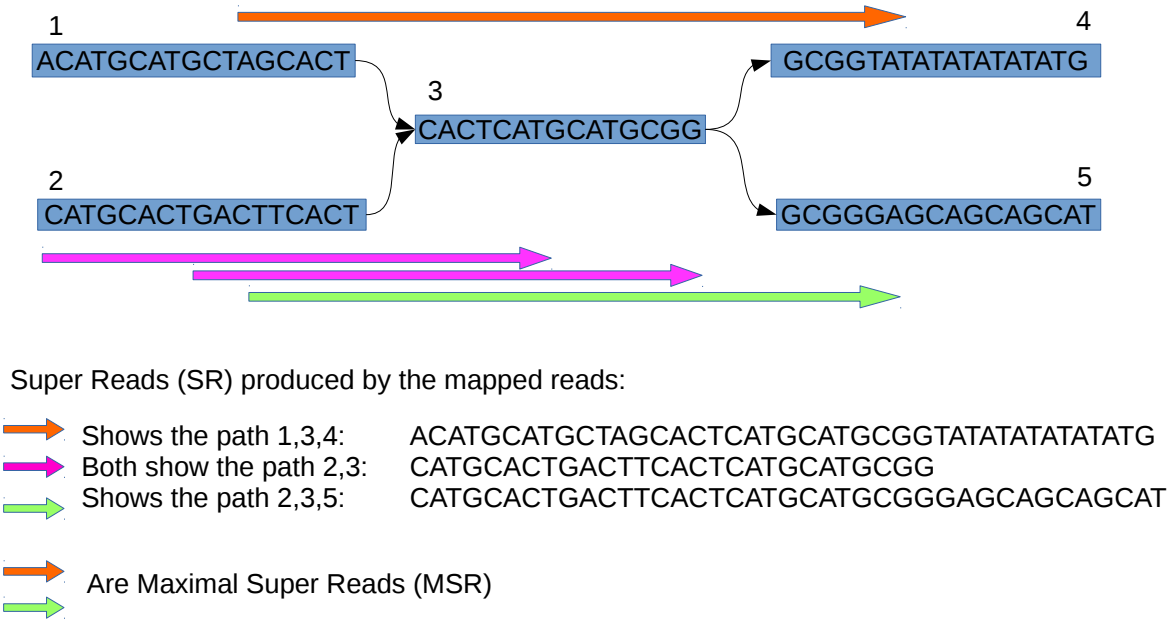

Figure 4.4: Super-reads generated by mapping the reads on the de Bruijn graph. For example the orange read maps on the unitigs 1, 3 and 4, its associated super-read is therefore the concatenation of those unitigs. We can see that super-reads may be longer than their associated reads, for example the orange read "AGCACTCATGCATGCG-GTAT" is associated to the super-read "ACATGCATGCTAGCACTCATGCATGCG-GTATATATATATATG". Several reads mapping on the same path generate the same super-read as the two pink reads, and some super-reads may be included in another super-reads. Super-reads that are not included in any other super-read are called maximal super-reads.

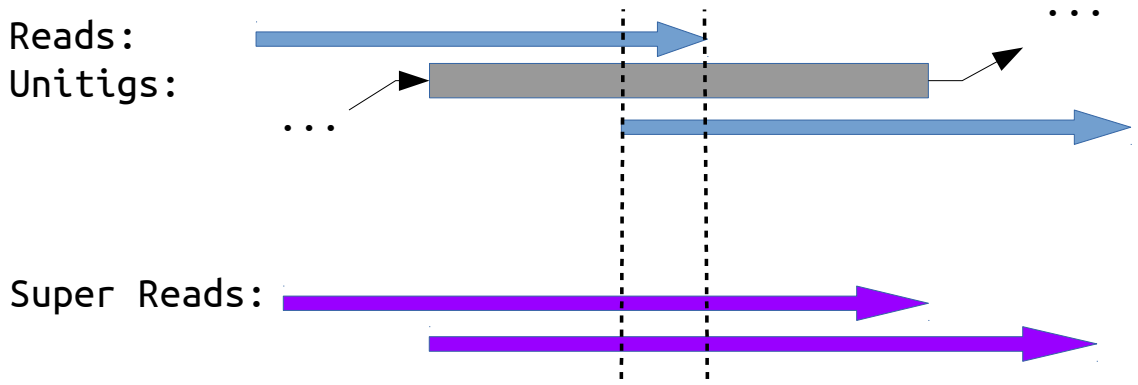

Figure 4.5: How the de Bruijn graph unitigs can enlarge the overlaps. In this example the two reads in blue share a small overlap. They are mapped on the de Bruijn graph and they both map on the gray unitig. Their SR in purple, enlarged with the unitig sequence, share a larger overlap than their associated reads.

To assess the quality of the SR obtained by mapping on a clean de Bruijn graph we conducted the following experiment.

- Simulation of 100X reads from a reference genome
- Read correction with Bloocoo
- de Bruijn graph construction with k=51
- Tip removal
- Read mapping on the graph to generate SR
- Duplicates removal from SR
- SR mapping on the reference genome

On the *E. coli* genome 1,642 different SR were obtained, all mapped on the reference genome with a similarity of 99%. All SR but 5 mapped with a similarity of 100% on the reference. On the *C. elegans* genome 357,183 different SR were obtained. All SR but 38 were mapped on the reference with a similarity of 95%. With a higher similarity of 99%, the number of unmapped SR was only of 260 and all SR but 502 mapped perfectly. Those experiments show that SR are very accurate sequence that can be almost considered as genome sequences.

Our idea is to use the sequences from such SR in order to produce higher order de Bruijn graph (Figure 4.9). By constructing a de Bruijn graph from the SR with a augmented k, we obtain a simpler de Bruijn graph than the first one constructed, with longer unitigs, less repeats and less spurious edges. As in the classical multiple k approaches like IDBA, the interest of starting with a low k is to be able to detect small overlaps between reads. Small overlaps are detected with the first value of k and kept in the latter graph with higher k because the overlaps between the SR are larger than the overlaps between reads. (Figure 4.5). However, if most of the time the overlaps are enlarged by the unitigs graph, it is possible that the augmented overlap is not large enough to be kept in the later steps.

We can raise k more gradually in order for the unitigs of the overlaps to be larger, along with the simplification of the graph, to avoid loosing overlaps. Furthermore some

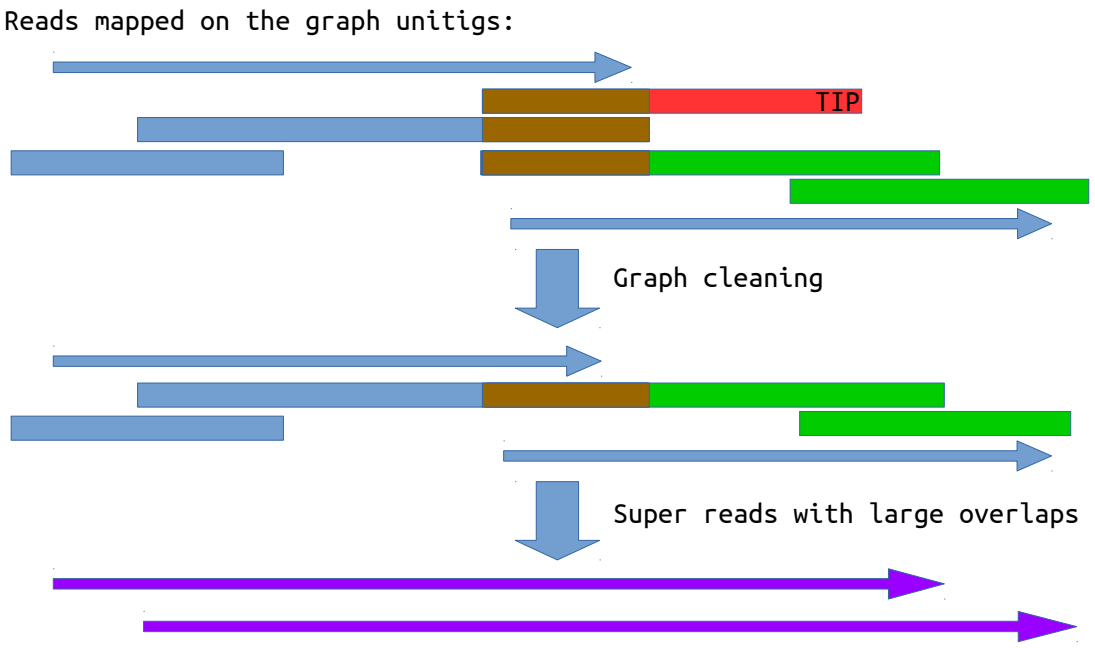

Figure 4.6: How the graph cleaning at each step helps the mapping of reads. In this example, two reads in blue are mapped on the de Bruijn graph unitigs. They share a small overlap that can not be augmented because the graph is branching due to a sequencing error. The tip removal allow to get a large unitig and two SR in purple that share a large overlap. If a too high value of k was used directly to build the de Bruijn graph, the overlaps between the two reads would have been lost.

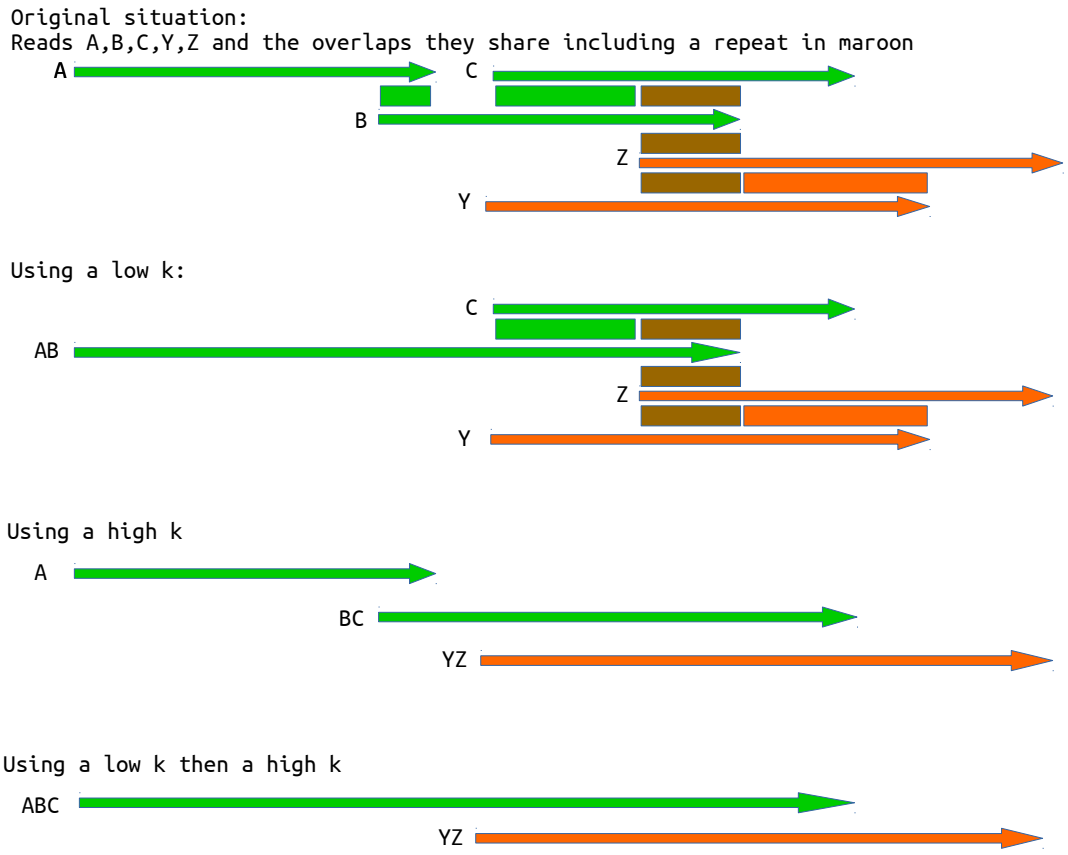

Figure 4.7: This example shows the interest of using several value of k. In this situation, the reads A and B share a small green overlap and the reads B and C share a large one with the reads Y and Z (in green and orange respectively). However, these overlaps share a medium repeat in brown. The low k assembly is fragmented because of the repeat but is able to detect the small overlap. The high k graph is able to avoid the repeat fragmentation but misses the small overlap. The successive construction allows to take advantages of both k values.

errors would become tips since the higher the k, the more sequencing errors become tips, so the graph can be cleaned in a better way at each step. Then the removal of such errors would also allow the generation of more contiguous unitigs, and would help in mapping reads with a higher k as it is shown in Figure 4.6. The Figure 4.7 shows the interest of such an iterative approach. We propose a workflow based on the presented ideas as a proof of concept assembler called BWISE. We give a high level view of the BWISE workflow and a more detailed version is presented in Figure 4.8.

- Read correction
- de Bruijn graph construction
- Graph tipping
- Read mapping on the de Bruijn graph
- Super-Reads selection
- Higher de Bruijn graph construction

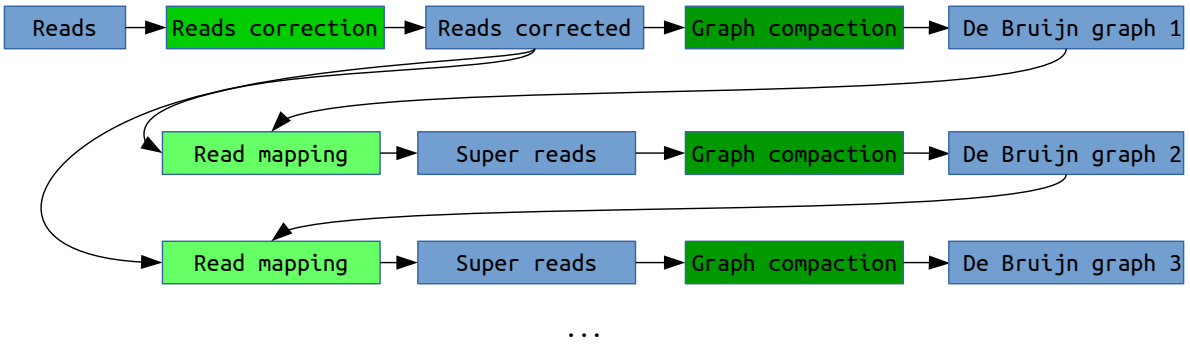

Figure 4.8: Bwise workflow. Data are represented in blue and software in green. The initial reads are corrected and a first de Bruijn graph is constructed. Then the corrected reads are mapped on the first de Bruijn graph to generate SR that are used to construct a second de Bruijn graph. The corrected reads are once again mapped on the second de Bruijn graph to produce new SR and so on.

First, the reads are corrected to contain as low sequencing error as possible to enable fast mapping of the reads on the graph. Secondly, the unitigs are computed and a tip removal step is performed to further clean the graph. Thirdly, reads are mapped on the de Bruijn graph, generating SR. In a fourth step, the redundant SR are removed from the SR set. Two phenomenons are responsible for redundancy. First, a very high number of reads may map on large unitigs. Many super-reads will therefore just consist in one large unitig. Secondly, some super-reads may be included in other super-reads and are therefore not informative. Such super-reads are discarded and only Maximal Super Reads (SR that are not included in any other) are conserved. Then MSR sequences are used to build a higher order de Bruijn graph.

We show the interest of our iterative approach in order to construct a very high k de Bruijn graph. Let us fix the goal of constructing a high order de Bruijn graph from 250 base pairs reads. We start by building a de Bruijn graph with $k = 51$ from corrected reads, then we use the MSR to obtain kmers of superior order. We create the successive graphs by adding 50 to the order at each step, results about unitig numbers, false positives and false negatives are presented in Table 4.2. We can see that we are able to construct a very high order de Bruijn graph with very few holes and errors, showing the interest of this approach.

4.2.2 Beyond read length

In order to further improve our assembler, we integrated the use of paired end reads. PE reads come from very close regions of the genome. Classical PE libraries have a fragment size between 300 and 800bp. In some cases the paired reads can even overlap if the size of the two reads is larger than the fragment size (for example pair of 250 bp reads with an fragment size of 450). The idea to use this longer distance information is to map a pair of reads separately on the graph and to create a single SR if the two SR generated by the read pair overlap. The toy example in Figure 4.9 shows that we can create a long

Order	Unitig number	FP	FN
k=51	949	408	32
k=101	447	72	32
k=151	329	25	32
k=201	270	54	32
k=251	222	198	92

Table 4.2: High order de Bruijn graph construction for 100X of 250 bp reads with 1% error rate from E.coli containing more than 4.6 million 251-mer.

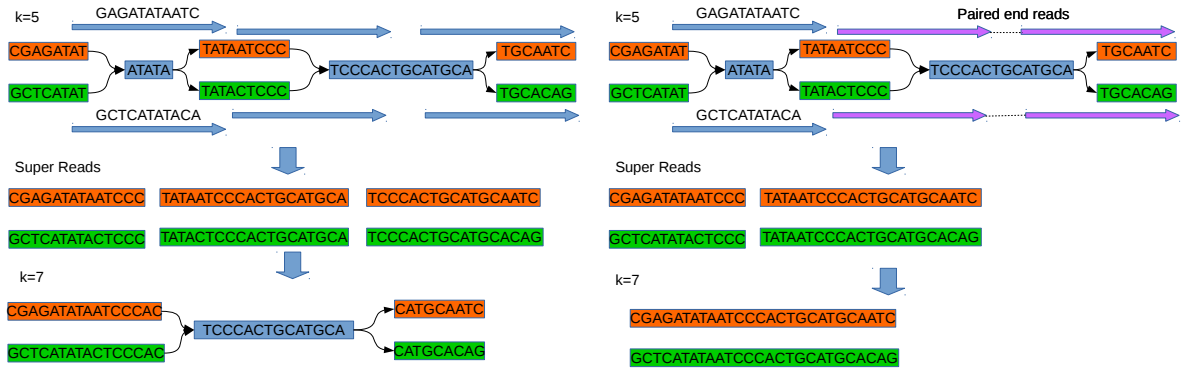

Figure 4.9: How to construct a high order de Bruijn graph with read mapping. A first de Bruijn graph is constructed with $k = 5$. Reads are mapped on it, producing super-reads. A new de Bruijn graph is constructed from the super-reads with a higher order, here $k = 7$. With paired reads we are able to phase more distant events or to solve larger repeats. In this example the purple reads in the right figure are paired. They produce longer SR than the unpaired reads from the left figure and are able to phase the three variants.

SR from a pair of non overlapping reads. The simpler the graph is , the longer the unitigs are and the more we will be able to connect distant paired reads. A SR generated from a pair of reads can be larger than the fragment size of the pair. Using this fact, we can generate SR longer than the reads and construct a de Bruijn graph with an order above the read length.

We already shown in the previous table a de Bruijn graph with $k = 251$ constructed from reads of length 250. However paired end reads allow both an easier process and the use of orders that can be superior to the size of the reads (Table 4.3). Following this principle, there is actually no upper bound on the distance we may authorize between reads as soon as we have unitigs large enough to connect them. Further work will consist in including longer range information as mate pair reads that present a way larger fragment size. We can also consider adding different types of data such as long reads but such data would require specific steps in the workflow. The use of long range information should solve large repeats and phase distant variants, providing a more continuous assembly, haploid or polyploid.

Order	Unitig number	FP	FN
k=51	949	408	32
k=101	576	1,208	32
k=151	356	341	32
k=201	276	125	32
k=251	224	210	108
k=301	122	67	5,992

Table 4.3: High order de Bruijn graph construction for 100X of 250 bp reads from E.Coli containing more than 4.6 million 301-mer.

"Building very high order de Bruijn graph" core messages:

- Super Reads created from read mapping can be seen as trusted sequences
- Iterative mapping allows the construction of high order de Bruijn graph
- Paired end reads allow the construction of a de Bruijn graph with k above the reads length

4.3 Assembly results

4.3.1 Haploid genome

In Table 4.4 we present some assembly results on various haploid genomes to assess the efficiency of our proposed solution against other assemblers.

We observe that BWISE is able to produce contigs as long as state of the art assemblers. We also note the scalability of BWISE that successfully performed a human genome assembly with less than 60 GB of RAM in two days. SPAdes and Platanus were

Assembly	#contigs	N50	N80
E. coli BWISE	56	210,994	126,805
*E. coli*SPAdes	71	178,400	83,005
E. coli Platanus	272	133,252	28,361
E. coli Minia	309	107,878	40,937
C. elegans BWISE	2,527	122,287	56,547
*C. elegans*SPAdes	6,550	103,128	43,904
C. elegans Platanus	21,413	52,661	18,330
C. elegans Minia	35,317	23,641	6,906
Human BWISE	132,272	74,148	29,705
Human Minia	884,788	20,202	7,381

Table 4.4: Simulated of 100x datasets on various genomes. Simulated reads are paired with 250 base pairs and a fragment size of 800.

Assembly	#contigs	N50	N80
E. coli 1% BWISE	307	64,217	29,717
E. coli 1% SPAdes	5,027	1,827	728
E. coli 1% Platanus	2,887	6,526	1,999
E. coli 2% BWISE	60	325,047	181,988
E. coli 2% SPAdes	2,411	12,632	3,450
E. coli 2% Platanus	2,488	6,896	2,028
E. coli 3% BWISE	8	2,610,736	605,526
E. coli 3% SPAdes	121	246,674	116,251
E. coli 3% Platanus	1,614	10,019	3,157
E. coli 4% BWISE	5	3,424,913	2,655,158
E. coli 4% SPAdes	4	3,424,851	3,424,851
E. coli 4% Platanus	1,071	13,346	5,279
E. coli 5% BWISE	2	4,639,590	4,639,574
E. coli 5% SPAdes	3	4,639,654	4,639,654
E. coli 5% Platanus	711	22,275	9,117

Table 4.5: Assembly comparison of simulated 100x datasets on *E. coli* genome with simulated heterozygosity. Simulated reads are paired with 250 base pairs and a fragment size of 800.

not able to perform the human assembly on the used machine with 250 GB of RAM.

4.3.2 Diploid genome

As seen before, assembly of highly heterozygous genomes is a case where the number of small repeats can be tremendous and where the use of the whole read length can lead to very significant improvements. In Table 4.5 we present assembly results of several tools, including BWISE, when used on regions with a known uniform heterozygosity. We first simulated uniform heterozygosity on haploid genomes and thereafter simulated reads from the artificial diploid references. We may argue that this does not represent a realistic scenario as the simulated heterozygosity "destroys" the repeats along the genomes. If a large repeat appear in the genome, we may add different variations on the different occurrences of the repeats. Thus, each occurrence of the repeat may be different from the other in the artificial diploid references. Therefore, presented results should only be seen as comparative evaluation. We can make several observations from these results. First we see that high heterozygosity is easy to solve and that tools achieve to produce long phased contigs since most variants were close to each other. Still, it is interesting to see that BWISE performs better than SPAdes on all rates. Especially on medium rates where SPAdes is not able to produce long contigs, results show that BWISE is able to do so.

On wider genomes, Table 4.6 shows that BWISE present similar performances and is still able to produce large contigs. On such dataset SPAdes and Platanus were not able to complete on the used server (2 days of timeout and 250 GB of RAM).

Assembly	#contigs	N50	N80
C. elegans 1% BWISE	9,505	50,870	22,850
C. elegans 2% BWISE	1,674	310,249	136,205
C. elegans 3% BWISE	429	1,089,405	488,453
C. elegans 4% BWISE	196	2,626,274	1,074,422

Table 4.6: Assembly comparison of simulated 100x datasets on *C. elegans* genome with simulated heterozygosity. Simulated reads are paired with 250 base pairs and a fragment size of 800. SPAdes exceeded the timeout.

Assembly	#contigs	N50	N80
E. coli 0.5% BWISE	1,370	10,390	5,633
E. coli 0.5% SPAdes	973	17,487	6,778
E. coli 0.5% Platanus	2,224	11,128	3,671
E. coli 0.1% BWISE	3,835	4,520	3,233
E. coli 0.1% SPAdes	55	210,905	88,598
E. coli 0.1% Platanus	926	40,837	14,688

Table 4.7: Assembly comparison of simulated 100x datasets on *E. coli* genome with low simulated heterozygosity. Simulated reads are paired with 250 base pairs and a fragment size of 800.

4.3.3 Future works

The essential limitation of BWISE workflow is noticeable when the heterozygosity rate is low (Table 4.7). Each time variants are pairwise too far away from each other, the assembly is broken since the contigs cannot be extended in a sure way. The results show that with 0.5% heterozygosity, neither BWISE or SPAdes are able to produce long contigs. On 0.1% SPAdes is able to produce long contigs where BWISE produces very short ones. This can be explained as SPAdes crushes bubbles and therefore produces a haploid assembly of 4.6 mega-bases while BWISE tries to produce a diploid assembly of 9.2 mega-bases.

The fact that we are not able to phase all heterozygous regions raise the question of the assembly representation. As it can be seen on Figure 4.10, the first and last variants were not phased by the reads, and BWISE produces the four contigs of solution 1. While being correct, this format may not be optimal for each usage. In some cases, we may be interested to get longer sequences by crushing bubbles. Another way to solve this problem would be to add longer range information in order to phase more distant variants. With more distant linked reads (as mate pair reads) phasing more distant variant should be possible and therefore produce more contiguous assembly.

The proposed solution based on successive mapping provides very interesting result and scale up to large genome. However several aspects have to be studied as the assembly quality, the behavior of BWISE with different coverage and the integration of longer range information. Several operations of the proposed workflow are quite redundant, as the repeated mapping of the whole read set. Thus, optimization of the workflow could

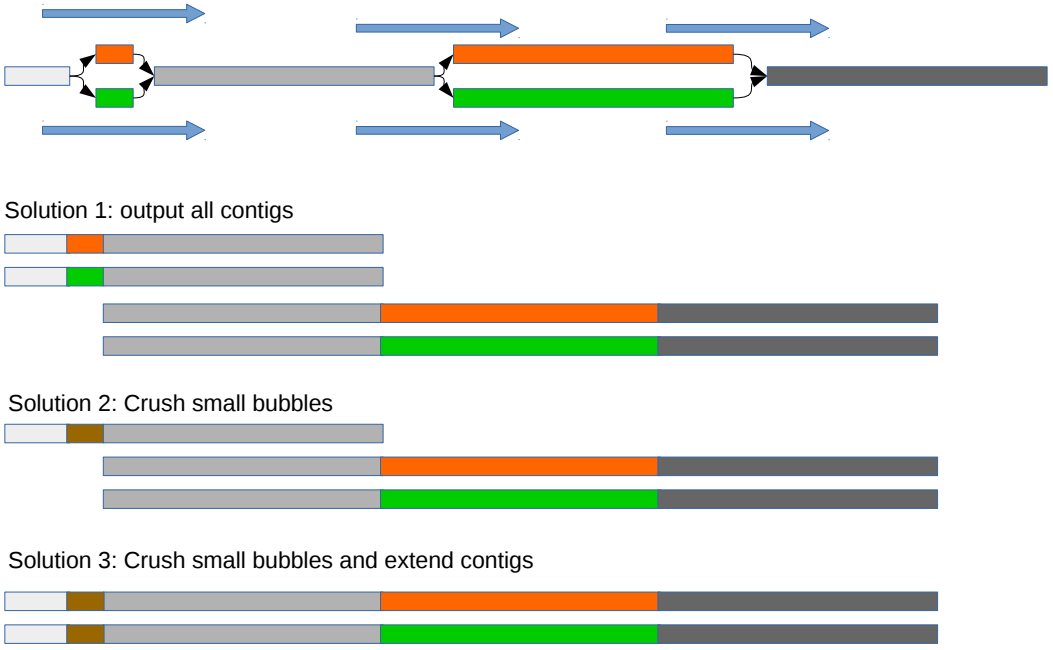

Figure 4.10: Potential representations of non phases contigs. The first solution proposed is a bit redundant. To limit the redundancy we can crush the remaining small bubbles as showed in solution 2. To further improve contigs length we could also extend the contigs with the merged bubbles sequences, as it is done in solution 3.

lead to sensible performances improvement. This method is not completely explored and in depth investigations are required to provide a robust and usable assembler for haploid and diploid genomes and eventually more complex assembly cases as pool-seq or meta-genomes.

"Assembly results" core messages:

- Iterative mapping leads to state of the art haploid assembly
- Iterative mapping outperforms state of the art tools in highly heterozygous zones
- Low heterozygosity polyploid assembly is difficult with short reads alone

Chapter 5

Conclusions and perspectives

5.1 Conclusion

Two main directions were explored during this thesis. The first aim was to propose new structures to handle the scaling of de Bruijn graph usages. The second aim was to propose new methods and techniques to no longer consider the de Bruijn graph only as an assembler internal representation but as a proper reference structure that can be used for various applications. This part reminds and summarizes our contributions and main results toward those objectives.

Several reasons justify the need for the development of efficient data structures in genome assembly. The assembly of human sized genomes still presents important scalability issues, with the need of very important resources. Very large genomes are common across the living organisms, especially plant genomes, and efficient assembly methods for such genomes are necessary to remove the present locks. The second interest of efficients techniques, for any genome, is to democratize the treatments of such datasets and make their analysis less costly and more accessible. In order to address the large genome assembly scalability using de Bruijn graphs, we proposed several practical and theoretical results. We described a new category of data structure called Navigational Data Structure that enable de Bruijn graph assembly with lower space requirements than the previously known bounds. Implementations of such structures could lead to extremely memory efficient assemblers. We proposed a proof of concept low memory usage assembler called DBGFM. The core idea of this method is to index the simple paths (unitigs) of the de Bruijn graph instead of its kmers into a memory efficient data structure, here a FM-index. DBGFM present of very low memory usage and is able to assemble a human genome with 1.5 GB of RAM. A take-home message is that considering a de Bruijn graph as a set of unitigs is an efficient practical choice. However the construction of such sets in low memory is not trivial. We therefore proposed a new method based on external memory to construct this set called BCALM. This method was slow and sequential, and the construction of the compacted de Bruijn graph was the bottleneck of DBGFM as in many assemblers. We therefore conceived a efficient and parallel method called BCALM2 to compute de Bruijn graph unitigs. BCALM2, relying on external memory present order of magnitude less memory that known implementation and was also significantly faster. Such a method can lead to a low memory assembly even for genome larger than the human one. In BCALM2, and in several other applications, we encountered the challenge of indexing extremely large numbers of object. The need for memory efficient and fast structure lead us to use minimal perfect hash functions (MPHF). Existing MPHF library presented very high memory usage and running time during construction. Thus we proposed a new scalable MPHF library, designed to construct MPHF from very large key set with moderate resources called BBHASH. The main property of BBHASH is the ability to construct a MPHF without memory overhead, and the construction is also parallel and faster than other available implementations. This implementation could be used virtually in any HPC field. We believe that such an efficient data structure could especially benefit in numerous applications in bioinformatics, where the need to index large number of objects is fundamental.

The de Bruijn graph is mainly seen as an assembly graph, a simple temporary structure used to generate contigs. If the de Bruijn graph is mainly used for assembly, many properties of this structure have been used outside of its first domain of application. For

numerous reasons, the classical representation of a genome into linear sequences is not fitted for all application. To cope with these problems, several new genome representations based on graph are proposed. If the de Bruijn graph is not an efficient representation for finished genomes, its properties can be used in advantageous ways in several applications. With efficient techniques to construct a (compacted) de Bruijn graph, we argued that it could be a good reference to represent one or multiple genomes. We presented different examples of the interest of such a representation. The unfinished genomes, where the de Bruijn graph may contain more information than the contigs sequences. Heterozygous genomes, as a de Bruijn graph contain and can represent efficiently the different variations that may be present in such genomes. Very redundant dataset that can be factored by the efficient representation of the repeated sequences in a de Bruijn graph.

Multiple applications could use such a reference graph directly, as read set comparison in meta-genomics or mapping read on the de Bruijn graph sequences. Thus, we argued that we need a read mapping method working directly on the de Bruijn graph structure to permit its versatile use as a reference. In order to enable the use of such representation, we provided a tool able to map reads efficiently on a de Bruijn graph called BGREAT. Despite the fact that we proved that mapping reads on a de Bruijn graph is a NP-Complete problem, the use of efficient heuristics allows BGREAT to be very efficient on current datasets. We presented several direct advantages of such a method. Results confirm that mapping reads directly on the de Bruijn graph instead of mapping on contigs allows a higher ratio of read mapped, showing the interest of mapping on the de Bruijn graph over mapping on its sequences. We also proved that the de Bruijn graph is a very accurate reference with very few sequencing errors and keeping the heterozygosity information. This facts show the de Bruijn graph as a very sound and easy to construct structure. Using this property, we presented an out of the box usage with the read corrector BCOOL. Results of the proof of concept read correction by mapping them on the de Bruijn graph are very interesting. BCOOL corrected several time more errors than state of the art correctors on high coverage simulated data. Outside of read correction, we believe that such a method could be used in very diverse usage of NGS data as assembly, scaffolding, read compression or variant calling.

Applied to genome assembly, the de Bruijn graph is essentially used for its scalability, but the splitting of the reads into kmers fragments the assembly. One of the motivation behind the conception of a tool like BGREAT was to improve the de Bruijn graph assembly continuity. The idea was to map reads on a de Bruijn graph in order to add back the reads information into de Bruijn graph assembly. To use this information, we proposed a new method to build a high order de Bruijn graph using the read length at a reasonable cost. The idea is to map reads on the de Bruijn graph and to use the mapped path sequences to build a de Bruijn graph with a higher order. We implemented this idea into a proof of concept assembler called BWISE. Similarly to the string graphs, the produced high order de Bruijn graph are able to use almost the whole length of the reads in order to solve repeats and produce more contiguous assembly than regular de Bruijn graph. We also showed that such an approach could benefit of paired end reads in order to produce de Bruijn graph with kmer size above the read length. Using those paired reads information improve the assembly continuity and the size of the contigs without a scaffolding step as we work directly on the de Bruijn graph. Preliminary results on simulated data on haploid genomes exhibit assemblies equivalent to state of the art as-

semblers. However, the most interesting results of this approach was a good assembly continuity in heterozygous regions were state of the art assemblers produce assembly of very poor quality. On our simulated data, BWISE is able to produce phased contigs and to assemble separately the different allele where other de Bruijn graph assembler are not able to make the distinction. We believe that the next assemblers will have to propose such heterozygosity awareness and that BWISE is a promising approach to do so.

Those results convey the idea that the de Bruijn graph as a data structure has not been used yet to the maximum of its potential.

5.2 Proposed methods

This thesis resulted in the following open-source softwares:

BCALM Proof of concept of de Bruijn graph compaction in low memory. BCALM was used in the DBGFM assembler to compute the unitigs to index and is described in [74]. Available `https://github.com/Malfoy/BCALM`

BCALM2 Efficient and parallel de Bruijn graph compaction in low memory. BCALM2 was used for the de Bruijn graph results presented in this document. It is used for the de Bruijn graph compaction before the use of BGREAT2 in BCOOL and BWISE. It is described in [82]. Available `https://github.com/GATB/bcalm`

BBHASH Scalable minimal perfect hash function construction. BBHASH MPHF is used in BCALM2, GATB and SRC softwares. It is described in [86]. Available `https://github.com/rizkg/BBHash`

BGREAT Proof of concept of read mapping on de Bruijn graph. It is described in [116]. Available `https://github.com/Malfoy/BGREAT`

BGREAT2 Read mapping on de Bruijn graph. An improved and self contained version of BGREAT. As said before it is used with BCALM2 in BCOOL and BWISE workflows. The improvements applied are described in chapter 3. Available `https://github.com/Malfoy/BGREAT2`

BCOOL de Bruijn graph based read correction. This read corrector and its preliminary results are presented in chapter 3. Available `https://github.com/Malfoy/BCOOL`

BWISE High order de Bruijn graph Assembly. This assembler and its preliminary results on haploid and diploid genomes are presented in chapter 4. Available `https://github.com/Malfoy/BWISE`

5.3 Future of sequencing

In 2008, the next generation sequencing revolutionized the genomic field. While it is hard to forecast the future of such fast evolving and diverse technologies, we can observe emerging usages of new kinds of data and imagine the potential benefits and challenges they may bring.

5.3.1 Third generation sequencing

New sequencing technologies are arising. Called third generation sequencing, those technologies propose order of magnitude longer sequences, but currently with a very high error rate (up to 15% [127]). Both Pacific Bioscience [6] and Oxford Nanopore [7] technologies are improving and may provide even longer reads with a low error rate in the near future. Those technologies have the potential to be a major shift in genome assembly. With such long sequences, most repeats could be solved and the "one contig one chromosome" dream could be achieved. But those sequences still present huge challenges. Tools have to deal with the important error rate. To make it worse, the type of error is mainly insertions and deletions errors which are harder to deal with than the substitutions usually encountered in NGS reads. The costs of such sequencing are currently high compared to NGS, while a high coverage is necessary in order to cope with the error rate. To assemble such sequences, the de Bruijn graph is not usable in a straightforward way. The string graph is therefore recovering its central place in assembly. But the computational cost of such an assembly is going through the roof and many tools have been proposed to make the overlap detection phase extremely efficient. Two approaches are advocated. The hybrid assembly approaches are based on the assumption that long reads could be used coupled with short reads to produce a high quality assembly in a cost efficient way. Either by correcting long reads with short reads [128] [129], or to scaffold short reads contigs with long reads [130] or assemble long and short reads together directly [131]). The pure assembly approaches [132] argues that long reads alone have the potential to produce the best assemblies because of their less biased sequencing protocols and that we should work on the computational cost of the assemblers.

5.3.2 Long range short reads sequencing

New short reads based technologies are also making an entrance into bioinformatics. The 3C protocol [133] could provide read pairs from the same chromosome with no upper bound on the insert size and therefore help phasing and solving repeats with no upper bound of size [51]. The 10X [134] technology provides short reads localized on short genome region of size around 10,000 bases. This kind of sequencing has shown great potential for phasing since reads from a pack come from the same allele [135] while it seems *a priori* harder to do so with long reads given their error rates. But for haploid assembly those technologies also great potential for solving repeats while being accurate sequences. The long reads present the advantages of representing a continuous sequences while 10X, like paired reads give sequences separated by holes.

5.4 Future of this work and perspectives

The different methods proposed in this documents could be continued in many ways, with refinements, new applications or integrations of other kinds of information. Some perspective considered are presented here.

While assembly of shorts reads is often considered as a well know subject, large genomes still require tremendous resources. But even for smaller genomes, faster structures and assembly methods will be required for the throughput of assemblers to follow the sequencer one. An efficient assembler, implementing a NDS based on unitigs indexed on a MPHF, could provide a extremely efficient method to work on very large genomes without current scalability problems or provide assembly with a very high throughput.

We showed that, beyond assembly, the de Bruijn graph is an extremely efficient data structure to represent a genome, or a set of genomes, with interesting properties that could be used in other fields than assembly. A scalable read corrector providing a very high ratio of perfect read would be a very useful tool for applications as assembly or variant calling. Our proposed corrector BCOOL could be tuned to adapt to real data and to different coverage in order to provide almost perfect short reads.

Several other applications of mapping read on a de Bruijn graph can be found as quantification, variant calling or read compression could be proposed in the near future. As the read correction, reads could be mapped on the graph and we could encode their path in the de Bruijn graph instead of their sequences in order to compress them. More direct usages would be quantification in genomic, meta-genomic or RNA-seq by mapping the read on a de Bruijn graph and extracting information from the amount of reads mapped on given components of the de Bruijn graph. Simple variant calling have been shown to be done directly from a de Bruijn graph structure, an interesting perspective would be to use the reads mapping on this graph and use this information to capture more complex cases.

Polyploid assembly is also a widely encountered problem while representing a very small part of the literature. We pointed those problems and presented new solutions and perspectives to address those challenges. BWISE polyploid assembly methods have still to be tuned to fit to the challenges of the real data but the door to the assembly of polyploid genomes is open. The use of paired reads directly on a de Bruijn graph could also inspire new scaffolding techniques that are mainly based on read mapping on contigs. The use of other kind of information in order to produce larger contigs is also a natural prolongation of this work. 10X reads seems a really promising technique to improve the polyploid assembly of BWISE, as the mapping method does not need important changes. The length of 10X DNA fragments could lead to extremely large and phased contigs.

We essentially talked about polyploid assembly but techniques used by BWISE could also be relevant for meta-genomic assembly. The read mapping could help distinguish the different individuals present in a de Bruijn graph as it is used now to separate the different alleles. To do so we have to address the challenges of heterogeneous coverages that can exist in meta-genomic. Almost the same argumentation could be used for transcriptome assembly of RNA sequences, an efficient assembler able to use Paired End data could tackle the combinatorial aspect of RNA assembly.

As BGREAT is able to align short reads on a de Bruijn graph, we are naturally interested to adapt our methods to be able to align long reads on a De Bruijn graph.

This kind of tool could have, as BGREAT, several applications. The alignment of the long reads on the de Bruijn graph could be used to correct their errors, or the alignment information could be used to order contigs or unitigs of the de Bruijn graph. Such a method could also be used in the context of pure long reads assembly as a efficient mean to detect overlap between them. The computational cost of long reads assembly has also to be addressed and efficient data structures will be required as well in order to efficiently assemble the future deluge of such datasets. We believe that with decreasing error rate the de Bruijn graph could be useful even for pure long reads assembly [136] as the de Bruijn graph paths represent easy to compute efficient consensus sequences.

Bibliography

[1] James D Watson and Francis HC Crick. A structure for deoxyribose nucleic acid. *Nature*, 171, 2004.

[2] Eric S Lander, Lauren M Linton, Bruce Birren, Chad Nusbaum, Michael C Zody, Jennifer Baldwin, Keri Devon, Ken Dewar, Michael Doyle, William FitzHugh, et al. Initial sequencing and analysis of the human genome. *Nature*, 409(6822):860–921, 2001.

[3] J Craig Venter, Mark D Adams, Eugene W Myers, Peter W Li, Richard J Mural, Granger G Sutton, Hamilton O Smith, Mark Yandell, Cheryl A Evans, Robert A Holt, et al. The sequence of the human genome. *science*, 291(5507):1304–1351, 2001.

[4] Frederick Sanger, Steven Nicklen, and Alan R Coulson. Dna sequencing with chain-terminating inhibitors. *Proceedings of the National Academy of Sciences*, 74(12):5463–5467, 1977.

[5] David R Bentley, Shankar Balasubramanian, Harold P Swerdlow, Geoffrey P Smith, John Milton, Clive G Brown, Kevin P Hall, Dirk J Evers, Colin L Barnes, Helen R Bignell, et al. Accurate whole human genome sequencing using reversible terminator chemistry. *nature*, 456(7218):53–59, 2008.

[6] John Eid, Adrian Fehr, Jeremy Gray, Khai Luong, John Lyle, Geoff Otto, Paul Peluso, David Rank, Primo Baybayan, Brad Bettman, et al. Real-time dna sequencing from single polymerase molecules. *Science*, 323(5910):133–138, 2009.

[7] Andrew H Laszlo, Ian M Derrington, Brian C Ross, Henry Brinkerhoff, Andrew Adey, Ian C Nova, Jonathan M Craig, Kyle W Langford, Jenny Mae Samson, Riza Daza, et al. Decoding long nanopore sequencing reads of natural dna. *Nature biotechnology*, 32(8):829–833, 2014.

[8] Daniel Aird, Michael G Ross, Wei-Sheng Chen, Maxwell Danielsson, Timothy Fennell, Carsten Russ, David B Jaffe, Chad Nusbaum, and Andreas Gnirke. Analyzing and minimizing pcr amplification bias in illumina sequencing libraries. *Genome biology*, 12(2):R18, 2011.

[9] Juliane C Dohm, Claudio Lottaz, Tatiana Borodina, and Heinz Himmelbauer. Substantial biases in ultra-short read data sets from high-throughput dna sequencing. *Nucleic acids research*, 36(16):e105–e105, 2008.

[10] Project Encode. The encode (encyclopedia of dna elements) project. *Science*, 306(5696):636–640, 2004.

[11] Sushmita Roy, Jason Ernst, Peter V Kharchenko, Pouya Kheradpour, Nicolas Negre, Matthew L Eaton, Jane M Landolin, Christopher A Bristow, Lijia Ma, Michael F Lin, et al. Identification of functional elements and regulatory circuits by drosophila modencode. *Science*, 330(6012):1787–1797, 2010.

[12] Jennifer Harrow, Adam Frankish, Jose M Gonzalez, Electra Tapanari, Mark Diekhans, Felix Kokocinski, Bronwen L Aken, Daniel Barrell, Amonida Zadissa, Stephen Searle, et al. Gencode: the reference human genome annotation for the encode project. *Genome research*, 22(9):1760–1774, 2012.

[13] Todd J Treangen and Steven L Salzberg. Repetitive dna and next-generation sequencing: computational challenges and solutions. *Nature Reviews Genetics*, 13(1):36–46, 2012.

[14] Jonghwan Kim, Akshay A Bhinge, Xochitl C Morgan, and Vishwanath R Iyer. Mapping dna-protein interactions in large genomes by sequence tag analysis of genomic enrichment. *Nature Methods*, 2(1):47–53, 2005.

[15] David S Johnson, Ali Mortazavi, Richard M Myers, and Barbara Wold. Genome-wide mapping of in vivo protein-dna interactions. *Science*, 316(5830):1497–1502, 2007.

[16] Kari-Jouko Räihä and Esko Ukkonen. The shortest common supersequence problem over binary alphabet is np-complete. *Theoretical Computer Science*, 16(2):187–198, 1981.

[17] Granger G Sutton, Owen White, Mark D Adams, and Anthony R Kerlavage. Tigr assembler: A new tool for assembling large shotgun sequencing projects. *Genome Science and Technology*, 1(1):9–19, 1995.

[18] Xiaoqiu Huang and Anup Madan. Cap3: A dna sequence assembly program. *Genome research*, 9(9):868–877, 1999.

[19] René L Warren, Granger G Sutton, Steven JM Jones, and Robert A Holt. Assembling millions of short dna sequences using ssake. *Bioinformatics*, 23(4):500–501, 2007.

[20] William R Jeck, Josephine A Reinhardt, David A Baltrus, Matthew T Hickenbotham, Vincent Magrini, Elaine R Mardis, Jeffery L Dangl, and Corbin D Jones. Extending assembly of short dna sequences to handle error. *Bioinformatics*, 23(21):2942–2944, 2007.

[21] Eugene W Myers, Granger G Sutton, Art L Delcher, Ian M Dew, Dan P Fasulo, Michael J Flanigan, Saul A Kravitz, Clark M Mobarry, Knut HJ Reinert, Karin A Remington, et al. A whole-genome assembly of drosophila. *Science*, 287(5461):2196–2204, 2000.

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

[23] R Staden. A mew computer method for the storage and manipulation of dna gel reading data. *Nucleic Acids Research*, 8(16):3673–3694, 1980.

[24] Alexey Gurevich, Vladislav Saveliev, Nikolay Vyahhi, and Glenn Tesler. Quast: quality assessment tool for genome assemblies. *Bioinformatics*, 29(8):1072–1075, 2013.

[25] Eugene W Myers. The fragment assembly string graph. *Bioinformatics*, 21(suppl 2):ii79–ii85, 2005.

[26] Jared T Simpson and Richard Durbin. Efficient construction of an assembly string graph using the fm-index. *Bioinformatics*, 26(12):i367–i373, 2010.

[27] Jared T Simpson and Richard Durbin. Efficient de novo assembly of large genomes using compressed data structures. *Genome research*, 22(3):549–556, 2012.

[28] Nicolaas Govert De Bruijn. A combinatorial problem. 1946.

[29] Dmitri Loguinov, Anuj Kumar, Vivek Rai, and Sai Ganesh. Graph-theoretic analysis of structured peer-to-peer systems: routing distances and fault resilience. In *Proceedings of the 2003 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 395–406. ACM, 2003.

[30] M Frans Kaashoek and David R Karger. Koorde: A simple degree-optimal distributed hash table. In *International Workshop on Peer-to-Peer Systems*, pages 98–107. Springer, 2003.

[31] Pavel A Pevzner, Haixu Tang, and Michael S Waterman. An eulerian path approach to dna fragment assembly. *Proceedings of the National Academy of Sciences*, 98(17):9748–9753, 2001.

[32] Paul Medvedev, Konstantinos Georgiou, Gene Myers, and Michael Brudno. Computability of models for sequence assembly. In *International Workshop on Algorithms in Bioinformatics*, pages 289–301. Springer Berlin Heidelberg, 2007.

[33] 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.

[34] Rayan Chikhi and Paul Medvedev. Informed and automated k-mer size selection for genome assembly. *Bioinformatics*, page btt310, 2013.

[35] Ruibang Luo, Binghang Liu, Yinlong Xie, Zhenyu Li, Weihua Huang, Jianying Yuan, Guangzhu He, Yanxiang Chen, Qi Pan, Yunjie Liu, et al. Soapdenovo2: an empirically improved memory-efficient short-read de novo assembler. *Gigascience*, 1(1):18, 2012.

[36] 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.

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

[38] Jonathan Butler, Iain MacCallum, Michael Kleber, Ilya A Shlyakhter, Matthew K Belmonte, Eric S Lander, Chad Nusbaum, and David B Jaffe. Allpaths: de novo assembly of whole-genome shotgun microreads. *Genome research*, 18(5):810–820, 2008.

[39] 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 Berlin Heidelberg, 2010.

[40] 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.

[41] 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.

[42] Sante Gnerre, Iain MacCallum, Dariusz Przybylski, Filipe J Ribeiro, Joshua N Burton, Bruce J Walker, Ted Sharpe, Giles Hall, Terrance P Shea, Sean Sykes, et al. High-quality draft assemblies of mammalian genomes from massively parallel sequence data. *Proceedings of the National Academy of Sciences*, 108(4):1513–1518, 2011.

[43] Thomas Conway, Jeremy Wazny, Andrew Bromage, Justin Zobel, and Bryan Beresford-Smith. Gossamer—a resource-efficient de novo assembler. *Bioinformatics*, 28(14):1937–1938, 2012.

[44] Rayan Chikhi and Guillaume Rizk. Space-efficient and exact de bruijn graph representation based on a bloom filter. In *International Workshop on Algorithms in Bioinformatics*, pages 236–248. Springer, 2012.

[45] Niranjan Nagarajan and Mihai Pop. Parametric complexity of sequence assembly: theory and applications to next generation sequencing. *Journal of computational biology*, 16(7):897–908, 2009.

[46] Mark Chaisson, Pavel Pevzner, and Haixu Tang. Fragment assembly with short reads. *Bioinformatics*, 20(13):2067–2074, 2004.

[47] Daniel H Huson, Knut Reinert, and Eugene W Myers. The greedy path-merging algorithm for contig scaffolding. *Journal of the ACM (JACM)*, 49(5):603–615, 2002.

[48] Marten Boetzer, Christiaan V Henkel, Hans J Jansen, Derek Butler, and Walter Pirovano. Scaffolding pre-assembled contigs using sspace. *Bioinformatics*, 27(4):578–579, 2011.

[49] Marten Boetzer and Walter Pirovano. Toward almost closed genomes with gapfiller. *Genome biology*, 13(6):1, 2012.

[50] Song Gao, Denis Bertrand, Burton KH Chia, and Niranjan Nagarajan. Opera-lg: Efficient and exact scaffolding of large, repeat-rich eukaryotic genomes with performance guarantees. *Genome biology*, 17(1):1, 2016.

[51] Jean-François Flot, Hervé Marie-Nelly, and Romain Koszul. Contact genomics: scaffolding and phasing (meta) genomes using chromosome 3d physical signatures. *FEBS letters*, 589(20PartA):2966–2974, 2015.

[52] Rei Kajitani, Kouta Toshimoto, Hideki Noguchi, Atsushi Toyoda, Yoshitoshi Ogura, Miki Okuno, Mitsuru Yabana, Masayuki Harada, Eiji Nagayasu, Haruhiko Maruyama, et al. Efficient de novo assembly of highly heterozygous genomes from whole-genome shotgun short reads. *Genome research*, 24(8):1384–1395, 2014.

[53] Yu Peng, Henry CM Leung, Siu-Ming Yiu, and Francis YL Chin. Meta-idba: a de novo assembler for metagenomic data. *Bioinformatics*, 27(13):i94–i101, 2011.

[54] Toshiaki Namiki, Tsuyoshi Hachiya, Hideaki Tanaka, and Yasubumi Sakakibara. Metavelvet: an extension of velvet assembler to de novo metagenome assembly from short sequence reads. *Nucleic acids research*, 40(20):e155–e155, 2012.

[55] Jaume Pellicer, Michael F Fay, and Ilia J Leitch. The largest eukaryotic genome of them all? *Botanical Journal of the Linnean Society*, 164(1):10–15, 2010.

[56] Temple F Smith and Michael S Waterman. Identification of common molecular subsequences. *Journal of molecular biology*, 147(1):195–197, 1981.

[57] Serafim Batzoglou, David B Jaffe, Ken Stanley, Jonathan Butler, Sante Gnerre, Evan Mauceli, Bonnie Berger, Jill P Mesirov, and Eric S Lander. Arachne: a whole-genome shotgun assembler. *Genome research*, 12(1):177–189, 2002.

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

[59] Ben Langmead, Cole Trapnell, Mihai Pop, and Steven L Salzberg. Ultrafast and memory-efficient alignment of short dna sequences to the human genome. *Genome biology*, 10(3):1, 2009.

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

[61] Inanc Birol, Anthony Raymond, Shaun D Jackman, Stephen Pleasance, Robin Coope, Greg A Taylor, Macaire Man Saint Yuen, Christopher I Keeling, Dana Brand, Benjamin P Vandervalk, et al. Assembling the 20 gb white spruce (picea glauca) genome from whole-genome shotgun sequencing data. *Bioinformatics*, page btt178, 2013.

[62] Shaun D Jackman, Benjamin P Vandervalk, Hamid Mohamadi, Justin Chu, Sarah Yeo, S Austin Hammond, Golnaz Jahesh, Hamza Khan, Lauren Coombe, Rene L Warren, et al. Abyss 2.0: Resource-efficient assembly of large genomes using a bloom filter. *Genome Research*, pages gr–214346, 2017.

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

[64] Daisuke Okanohara and Kunihiko Sadakane. Practical entropy-compressed rank/select dictionary. In *Proceedings of the Meeting on Algorithm Engineering & Expermiments*, pages 60–70. Society for Industrial and Applied Mathematics, 2007.

[65] 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.

[66] Chris Purcell and Tim Harris. Non-blocking hashtables with open addressing. In *International Symposium on Distributed Computing*, pages 108–121. Springer, 2005.

[67] Guillaume Rizk, Dominique Lavenier, and Rayan Chikhi. Dsk: k-mer counting with very low memory usage. *Bioinformatics*, page btt020, 2013.

[68] Sebastian Deorowicz, Marek Kokot, Szymon Grabowski, and Agnieszka Debudaj-Grabysz. Kmc 2: Fast and resource-frugal k-mer counting. *Bioinformatics*, 31(10):1569–1576, 2015.

[69] Chengxi Ye, Zhanshan Sam Ma, Charles H Cannon, Mihai Pop, and W Yu Douglas. Exploiting sparseness in de novo genome assembly. *BMC bioinformatics*, 13(6):1, 2012.

[70] Sara El-Metwally, Magdi Zakaria, and Taher Hamza. Lightassembler: fast and memory-efficient assembly algorithm for high-throughput sequencing reads. *Bioinformatics*, page btw470, 2016.

[71] 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.

[72] 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, 2013.

[73] 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, 2014.

[74] 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 International Publishing, 2014.

[75] Evangelos Georganas, Aydın Buluç, Jarrod Chapman, Steven Hofmeyr, Chaitanya Aluru, Rob Egan, Leonid Oliker, Daniel Rokhsar, and Katherine Yelick. Hipmer: an extreme-scale de novo genome assembler. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, page 14. ACM, 2015.

[76] Jintao Meng, Bingqiang Wang, Yanjie Wei, Shengzhong Feng, and Pavan Balaji. Swap-assembler: scalable and efficient genome assembly towards thousands of cores. *BMC bioinformatics*, 15(Suppl 9):S2, 2014.

[77] Jintao Meng, Sangmin Seo, Pavan Balaji, Yanjie Wei, Bingqiang Wang, and Shenzhong Feng. Swap-assembler 2: Optimization of de novo genome assembler at extreme scale. In *Parallel Processing (ICPP), 2016 45th International Conference on*, pages 195–204. IEEE, 2016.

[78] Michael Burrows and David J Wheeler. A block-sorting lossless data compression algorithm. 1994.

[79] 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.

[80] Steven L Salzberg, Adam M Phillippy, Aleksey Zimin, Daniela Puiu, Tanja Magoc, Sergey Koren, Todd J Treangen, Michael C Schatz, Arthur L Delcher, Michael Roberts, et al. Gage: A critical evaluation of genome assemblies and assembly algorithms. *Genome research*, 22(3):557–567, 2012.

[81] T. C Conway and A. J Bromage. Succinct data structures for assembling large genomes. *Bioinformatics*, 27(4):479, 2011.

[82] 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.

[83] 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.

[84] 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.

[85] Aleksey Zimin, Kristian A Stevens, Marc W Crepeau, Ann Holtz-Morris, Maxim Koriabine, Guillaume Marçais, Daniela Puiu, Michael Roberts, Jill L Wegrzyn,

Pieter J de Jong, et al. Sequencing and assembly of the 22-gb loblolly pine genome. *Genetics*, 196(3):875–890, 2014.

[86] 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.

[87] Camille Marchet, Antoine Limasset, Lucie Bittner, and Pierre Peterlongo. A resource-frugal probabilistic dictionary and applications in (meta) genomics. *arXiv preprint arXiv:1605.08319*, 2016.

[88] Camille Marchet, Lolita Lecompte, Antoine Limasset, Lucie Bittner, and Pierre Peterlongo. A resource-frugal probabilistic dictionary and applications in bioinformatics. *arXiv preprint arXiv:1703.00667*, 2017.

[89] Lars Feuk, Andrew R Carson, and Stephen W Scherer. Structural variation in the human genome. *Nature Reviews Genetics*, 7(2):85–97, 2006.

[90] Pierre Peterlongo, Nicolas Schnel, Nadia Pisanti, Marie-France Sagot, and Vincent Lacroix. Identifying snps without a reference genome by comparing raw reads. In *International Symposium on String Processing and Information Retrieval*, pages 147–158. Springer Berlin Heidelberg, 2010.

[91] Jacob F Degner, John C Marioni, Athma A Pai, Joseph K Pickrell, Everlyne Nkadori, Yoav Gilad, and Jonathan K Pritchard. Effect of read-mapping biases on detecting allele-specific expression from rna-sequencing data. *Bioinformatics*, 25(24):3207–3212, 2009.

[92] Débora YC Brandt, Vitor RC Aguiar, Bárbara D Bitarello, Kelly Nunes, Jérôme Goudet, and Diogo Meyer. Mapping bias overestimates reference allele frequencies at the hla genes in the 1000 genomes project phase i data. *G3: Genes| Genomes| Genetics*, 5(5):931–941, 2015.

[93] Peter H Sudmant, Tobias Rausch, Eugene J Gardner, Robert E Handsaker, Alexej Abyzov, John Huddleston, Yan Zhang, Kai Ye, Goo Jun, Markus Hsi-Yang Fritz, et al. An integrated map of structural variation in 2,504 human genomes. *Nature*, 526(7571):75–81, 2015.

[94] 1000 Genomes Project Consortium et al. A global reference for human genetic variation. *Nature*, 526(7571):68–74, 2015.

[95] Agnieszka Danek, Sebastian Deorowicz, and Szymon Grabowski. Indexes of large genome collections on a pc. *PloS one*, 9(10):e109384, 2014.

[96] Guillaume Holley, Roland Wittler, and Jens Stoye. Bloom filter trie–a data structure for pan-genome storage. In *International Workshop on Algorithms in Bioinformatics*, pages 217–230. Springer, 2015.

[97] Benedict Paten, Adam M Novak, Jordan M Eizenga, and Erik Garrison. Genome graphs and the evolution of genome inference. *bioRxiv*, page 101816, 2017.

[98] Adam M Novak, Glenn Hickey, Erik Garrison, Sean Blum, Abram Connelly, Alexander Dilthey, Jordan Eizenga, MA Saleh Elmohamed, Sally Guthrie, André Kahles, et al. Genome graphs. *bioRxiv*, page 101378, 2017.

[99] Alexander Dilthey, Charles Cox, Zamin Iqbal, Matthew R Nelson, and Gil McVean. Improved genome inference in the mhc using a population reference graph. *Nature genetics*, 47(6):682–688, 2015.

[100] Ngan Nguyen, Glenn Hickey, Daniel R Zerbino, Brian Raney, Dent Earl, Joel Armstrong, W James Kent, David Haussler, and Benedict Paten. Building a pan-genome reference for a population. *Journal of Computational Biology*, 22(5):387–401, 2015.

[101] Richard Durbin. Efficient haplotype matching and storage using the positional burrows–wheeler transform (pbwt). *Bioinformatics*, 30(9):1266–1272, 2014.

[102] Adam M Novak, Erik Garrison, and Benedict Paten. A graph extension of the positional burrows-wheeler transform and its applications. In *International Workshop on Algorithms in Bioinformatics*, pages 246–256. Springer, 2016.

[103] Mingjie Wang, Yuzhen Ye, and Haixu Tang. A de bruijn graph approach to the quantification of closely-related genomes in a microbial community. *Journal of Computational Biology*, 19(6):814–825, 2012.

[104] Manuel Garber, Manfred G Grabherr, Mitchell Guttman, and Cole Trapnell. Computational methods for transcriptome annotation and quantification using rna-seq. *Nature methods*, 8(6):469–477, 2011.

[105] Bo Liu, Hongzhe Guo, Michael Brudno, and Yadong Wang. debga: read alignment with de bruijn graph-based seed and extension. *Bioinformatics*, page btw371, 2016.

[106] Gaëtan Benoit, Dominique Lavenier, Claire Lemaitre, and Guillaume Rizk. Bloocoo, a memory efficient read corrector. In *European Conference on Computational Biology (ECCB)*, 2014.

[107] Ilya Minkin, Anand Patel, Mikhail Kolmogorov, Nikolay Vyahhi, and Son Pham. Sibelia: a scalable and comprehensive synteny block generation tool for closely related microbial genomes. In *International Workshop on Algorithms in Bioinformatics*, pages 215–229. Springer, 2013.

[108] Shoshana Marcus, Hayan Lee, and Michael C Schatz. Splitmem: a graphical algorithm for pan-genome analysis with suffix skips. *Bioinformatics*, 30(24):3476–3483, 2014.

[109] Ilia Minkin, Son Pham, and Paul Medvedev. Twopaco: An efficient algorithm to build the compacted de bruijn graph from many complete genomes. *Bioinformatics*, page btw609, 2016.

[110] Timo Beller and Enno Ohlebusch. Efficient construction of a compressed de bruijn graph for pan-genome analysis. In *Annual Symposium on Combinatorial Pattern Matching*, pages 40–51. Springer, 2015.

[111] Hayan Lee and Michael C Schatz. Genomic dark matter: the reliability of short read mapping illustrated by the genome mappability score. *Bioinformatics*, 28(16):2097–2105, 2012.

[112] 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.

[113] Gustavo AT Sacomoto, Janice Kielbassa, Rayan Chikhi, Raluca Uricaru, Pavlos Antoniou, Marie-France Sagot, Pierre Peterlongo, and Vincent Lacroix. K is s plice: de-novo calling alternative splicing events from rna-seq data. *BMC bioinformatics*, 13(6):S5, 2012.

[114] Son K Pham and Pavel A Pevzner. Drimm-synteny: decomposing genomes into evolutionary conserved segments. *Bioinformatics*, 26(20):2509–2516, 2010.

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

[116] Antoine Limasset, Bastien Cazaux, Eric Rivals, and Pierre Peterlongo. Read mapping on de bruijn graphs. *BMC bioinformatics*, 17(1):237, 2016.

[117] Xiao Yang, Sriram P Chockalingam, and Srinivas Aluru. A survey of error-correction methods for next-generation sequencing. *Briefings in bioinformatics*, 14(1):56–66, 2013.

[118] Yongchao Liu, Jan Schröder, and Bertil Schmidt. Musket: a multistage k-mer spectrum-based error corrector for illumina sequence data. *Bioinformatics*, 29(3):308–315, 2013.

[119] Heng Li. Bfc: correcting illumina sequencing errors. *Bioinformatics*, page btv290, 2015.

[120] Li Song, Liliana Florea, and Ben Langmead. Lighter: fast and memory-efficient sequencing error correction without counting. *Genome biology*, 15(11):509, 2014.

[121] Hamid Mohamadi, Hamza Khan, and Inanc Birol. ntcard: A streaming algorithm for cardinality estimation in genomics data. *Bioinformatics*, page btw832, 2017.

[122] Dent Earl, Keith Bradnam, John St John, Aaron Darling, Dawei Lin, Joseph Fass, Hung On Ken Yu, Vince Buffalo, Daniel R Zerbino, Mark Diekhans, et al. Assemblathon 1: a competitive assessment of de novo short read assembly methods. *Genome research*, 21(12):2224–2241, 2011.

[123] Keith R Bradnam, Joseph N Fass, Anton Alexandrov, Paul Baranay, Michael Bechner, Inanç Birol, Sébastien Boisvert, Jarrod A Chapman, Guillaume Chapuis, Rayan Chikhi, et al. Assemblathon 2: evaluating de novo methods of genome assembly in three vertebrate species. *GigaScience*, 2(1):10, 2013.

[124] Ellen M Leffler, Kevin Bullaughey, Daniel R Matute, Wynn K Meyer, Laure Segurel, Aarti Venkat, Peter Andolfatto, and Molly Przeworski. Revisiting an old riddle: what determines genetic diversity levels within species? *PLoS Biol*, 10(9):e1001388, 2012.

[125] Anton Bankevich, Sergey Nurk, Dmitry Antipov, Alexey A. Gurevich, Mikhail Dvorkin, Alexander S. Kulikov, Valery M. Lesin, Sergey I. Nikolenko, Son K. Pham, Andrey D. Prjibelski, Alex Pyshkin, Alexander Sirotkin, Nikolay Vyahhi, Glenn Tesler, Max A. Alekseyev, and Pavel A. Pevzner. SPAdes: A new genome assembly algorithm and its applications to single-cell sequencing. *Journal of Computational Biology*, 19(5):455–477, 2012.

[126] Aleksey V Zimin, Guillaume Marçais, Daniela Puiu, Michael Roberts, Steven L Salzberg, and James A Yorke. The masurca genome assembler. *Bioinformatics*, 29(21):2669–2677, 2013.

[127] Mark J Chaisson and Glenn Tesler. Mapping single molecule sequencing reads using basic local alignment with successive refinement (blasr): application and theory. *BMC bioinformatics*, 13(1):238, 2012.

[128] Kin Fai Au, Jason G Underwood, Lawrence Lee, and Wing Hung Wong. Improving pacbio long read accuracy by short read alignment. *PloS one*, 7(10):e46679, 2012.

[129] Mohammed-Amin Madoui, Stefan Engelen, Corinne Cruaud, Caroline Belser, Laurie Bertrand, Adriana Alberti, Arnaud Lemainque, Patrick Wincker, and Jean-Marc Aury. Genome assembly using nanopore-guided long and error-free dna reads. *BMC genomics*, 16(1):327, 2015.

[130] Marten Boetzer and Walter Pirovano. Sspace-longread: scaffolding bacterial draft genomes using long read sequence information. *BMC bioinformatics*, 15(1):211, 2014.

[131] Viraj Deshpande, Eric DK Fung, Son Pham, and Vineet Bafna. Cerulean: A hybrid assembly using high throughput short and long reads. In *International Workshop on Algorithms in Bioinformatics*, pages 349–363. Springer Berlin Heidelberg, 2013.

[132] Chen-Shan Chin, David H Alexander, Patrick Marks, Aaron A Klammer, James Drake, Cheryl Heiner, Alicia Clum, Alex Copeland, John Huddleston, Evan E Eichler, et al. Nonhybrid, finished microbial genome assemblies from long-read smrt sequencing data. *Nature methods*, 10(6):563–569, 2013.

[133] Job Dekker, Karsten Rippe, Martijn Dekker, and Nancy Kleckner. Capturing chromosome conformation. *science*, 295(5558):1306–1311, 2002.

[134] Jacob O Kitzman. Haplotypes drop by drop. *Nature biotechnology*, 34(3):296–298, 2016.

[135] Neil I Weisenfeld, Vijay Kumar, Preyas Shah, Deanna Church, and David B Jaffe. Direct determination of diploid genome sequences. *bioRxiv*, page 070425, 2016.

[136] Leena Salmela, Riku Walve, Eric Rivals, and Esko Ukkonen. Accurate self-correction of errors in long reads using de bruijn graphs. *Bioinformatics*, page btw321, 2016.