Development of efficient De Bruijn graph-based algorithms for genome assembly

Autor: Borja Freire Castro

Tesis doctoral UDC / 2022

Directores:
José Ramón Parama Gabia
Leena Salmela

Tutora: Susana Ladra González

Development of efficient De Bruijn graph-based algorithms for genome assembly

Autor: Borja Freire Castro

Tesis doctoral UDC / 2022

Directores:
José Ramón Paramá Gabía
Leena Salmela

Tutora: Susana Ladra González

Programa Oficial de Doutoramento en Computación

PhD thesis supervised by
Tesis doctoral dirigida por

José Ramón Paramá Gabía
Departamento de Computación y Tecnologías de la Información
Facultad de Informática
Universidade da Coruña
15071 A Coruña (España)
Tel: +34 881011241
jose.parama@udc.es

Leena Salmela
Department of Computer Science
Faculty of Science
University of Helsinki
FIN-00014 FINLAND
Tel: +358 (0) 2941 911
leena.salmela@cs.helsinki.fi

Tutored by
Tutorizada por

Susana Ladra González
Departamento de Computación y Tecnologías de la Información
Facultad de Informática
Universidade da Coruña
15071 A Coruña (España)
Tel: +34 881011218
susana.ladra@udc.es

Leena Salmela y José Ramón Paramá Gabía, como directores, y Susana Ladra como tutora acreditamos que esta tesis cumple los requisitos para optar al título de doctor internacional y autorizamos su depósito y defensa por parte de Borja Freire Castro cuya firma también se incluye.

Leena Salmela and José Ramón Paramá Gabía, as directors, and Susana Ladra as tutor, certify that this thesis meets the requirements to qualify for the title of international doctor and we authorize its deposit and defense by Borja Freire Castro, whose signature is also included.

A mis padres que hace diez años me encaminaron por la vía de informática en lugar de las matemáticas y eso me ha permitido llegar adonde estoy ahora mismo.

Acknowledgements

Four years ago when I started my doctorate I did it full of doubts and like all Spanish students I dedicated the first month, or year, to searching the internet: what is a doctorate worth? Is the doctorate worth it? a criminal when you have an FPU? Why is it published so much in machine learning?, etc. Despite the fact that the answers were not very encouraging, he decided to follow this path and I honestly do not regret it at all. I have to especially thank my directors for this, the 3 (or two, we'll see), for making my path bearable and pleasant. More than special thanks to José for having acted as a sparring partner for ideas, as a psychologist, as a motivational coach, as a grammar checker and as a punching bag, among other things. Also thank my parents who have supported me for 28 years and when I decided to make a career in begging, also known as PhD candidate, they only had words of support, and a little sorrow.

I would also like to thank the universities of Helsinki, Chile and Saarbrüken for having treated me as one more member of the group. Especially to the University of Helsinki and the group *Genome-Scale Algorithm Design* who hosted me repeatedly.

Finally, thanks to all the entities that have financed me and have allowed me to travel and do my doctorate with full performance. I want to thank the Galician Research Center "CITIC", funded by the Xunta de Galicia and the European Union (European Regional Development Fund - Galicia Program 2014-2020), with the ED431G 2019/01 grant. I would also like to thank the Xunta de Galicia/FEDER-UE that has financed this thesis through the grants [ED431C 2021/53; IG240.2020.1.185; IN852A 2018/14]; to the Ministry of Science and Innovation with scholarships [TIN2016-78011-C4-1-R; FPU17/02742; PID2019-105221RB-C41; PID2020-114635RB-I00]; and to the Academy of Finland [grants 308030 and 323233 (LS)].

Agradecimientos

Hace cuatro años cuando empecé el doctorado lo hice lleno de dudas y como todo estudiante español me dediqué el primer mes, o año, a buscar en internet: ¿para que vale un doctorado?¿merece la pena el doctorado?¿porque te tratan como un delincuente cuando tienes una FPU?¿porque se publica tanto en aprendizaje automático?, etc. Pese a que las respuestas no fueron demasiado alentadoras decidí seguir este camino y sinceramente no me arrepiento para nada. Esto se lo tengo que agradecer especialmente a mis directores, los 3 (o dos ya veremos), por haceme el camino llevadero y agradable. Agradecimiento más que especial para José por haber hecho de sparring de ideas, de psicólogo, de coach motivacional, de corrector gramatical y de saco de boxeo, entre otras cosas. También agradecérselo a mis padres que me han soportado durante 28 años y que cuando decidí hacer carrera en la mendicidad, también conocido como estudiante de doctorado, solo tuvieron palabras de apoyo, y un poquito de pena.

Agradecer también a las universidades de Helsinki, Chile y Saarbrüken por haberme tratado como a un miembro más del grupo. Especialmente a la universidad de Helsinki y al grupo *Genome Scale algorithm design* que me acogió en repetidas ocasiones.

Por último, los agradecimiento a todas las entidades que me han financiado y me han permitido viajar y hacer mi doctorado con total desempeño. Quiero agradecer al Centro de Investigación de Galicia "CITIC", financiado por la Xunta de Galicia y la Unión Europea (European Regional Development Fund- Galicia 2014-2020 Program), con la beca ED431G 2019/01. También agradecer a la Xunta de Galicia/FEDER-UE que ha financiado esta tesis a través de las becas [ED431C 2021/53; IG240.2020.1.185; IN852A 2018/14]; al Ministerio de Ciencia e Innovación con las becas [TIN2016-78011-C4-1-R; FPU17/02742; PID2019-105221RB-C41; PID2020-114635RB-I00]; y a la academia de Finlandia [grants 308030 and 323233 (LS)].

Agradecementos

Hai catro anos cando comecei o doutoramento fíxeno cheo de dúbidas e como todos os estudantes españois dediqueino o primeiro mes, ou ano, a buscar en internet: que vale un doutoramento?, ¿vale o doutoramento?, un delincuente cando tes unha FPU?, por que se publica tanto en machine learning?, etc. A pesar de que as respostas non foron moi alentadoras, decidiu seguir este camiño e sinceramente non me arrepinto nada. Teño que agradecer especialmente aos meus directores este, os 3 (ou dous, xa veremos), por facer o meu camiño levadeiro e agradable. Agradecemento máis que especial a José por ter actuado como compañeira de combate de ideas, como psicólogo, como adestrador motivacional, como corrector gramatical e como saco de boxeo, entre outras cousas. Tamén agradecer aos meus pais que me apoiaron durante 28 anos e cando decidín facer unha carreira de mendicidade, tamén coñecido como estudante de doutoramento, só tiveron palabras de apoio, e un pouco de mágoa.

Tamén me gustaría agradecer ás universidades de Helsinki, Chile e Saarbrüken que me trataran como un membro máis do grupo. Especialmente á Universidade de Helsinki e ao grupo *Genome-Scale Algorithm Design* que me acolleron varias veces.

Para rematar, agradecer a todas as entidades que me financiaron e que me permitiron viaxar e facer o doutoramento con pleno rendemento. Quero agradecer ao Centro Galego de Investigación "CITIC", financiado pola Xunta de Galicia e a Unión Europea (Fondo Europeo de Desenvolvemento Rexional - Programa Galicia 2014-2020), coa subvención ED431G 2019/01. Tamén quero agradecer á Xunta de Galicia/FEDER-UE que financiou esta tese a través das axudas [ED431C 2021/53; IG240.2020.1.185; IN852A 2018/14]; ao Ministerio de Ciencia e Innovación con bolsas [TIN2016-78011-C4-1-R; FPU17/02742; PID2019-105221RB-C41; PID2020-114635RB-I00]; e á Academia de Finlandia [subvencións 308030 e 323233 (LS)].

Abstract

During the last two decades, thanks to the development of new sequencing techniques, the study of the genome has become very popular in order to discover the genetic variation present in both humans and other organisms. The predominant mode of genome analysis is through the assembly of reads in one or multiple chains for as long as possible. The most traditional way of assembly is the one that involves reads from a single genome. In this field, in the last decade, third-generation readings have emerged with new challenges for which there are no efficient solutions. The first contribution that has been made in this thesis is *Compact-Flye*, a tool for the efficient assembly of third-generation reads on the *Flye* algorithm. This tool is based on the ingenious use of compact data structures to improve typical assembly steps such as counting and indexing k-mers. Apart from the assembly of a genome, there are techniques that seek to assemble all the genomes contained in a given sample. This assembly is known as multiple sequence assembly or haplotype reconstruction, a subject also treated in this thesis. Our first approach to solving this has been *viaDBG*, which is the first solution based on *de Bruijn* graphs that offers results comparable to current techniques in viral genome assembly while maintaining the efficiency of these graphs. Our second contribution is *ViQUF*, which is a natural improvement on its predecessor. *ViQUF* completely changes the algorithm of *viaDBG* but continues to be based on the same structures, although with some variations that allow it not only to improve results in terms of time and quality, but also to provide additional information such as an estimate of the relative presence of each species in the sample.

Resumen

Durante las últimas dos décadas, gracias al desarrollo de nuevas técnias secuenciación, el estudio del genoma ha ganado mucha popularidad de cara a conocer la variación genética presente tanto seres humanos como otros organismos. El modo predominante de análisis del genoma es a través del ensamblaje de lecturas en una o múltiples cadenas lo más largas posibles. La manera más tradicional de ensamblaje es el que implica lecturas provenientes de un solo genoma. En este campo, en la última década han surgido las lecturas de tercera generación con nuevos retos para los que no existen soluciones eficientes. La primera aportación que se ha realizado en esta tesis es *Compact-Flye* una herramienta para el ensamblaje eficiente de lecturas de tercera generación sobre el algoritmo *Flye*. Esta herramienta está basada en el uso igenioso de estructuras compactas de datos para mejorar etapas típicas del ensamblaje como el conteo e indexación de k-mers. Al margen del ensamblaje de un genoma existen técnicas que buscan ensamblar todos los genomas contenidos en una muestra determinada. Este ensamblaje es conocido como ensamblaje múltiple de secuencias o reconstrucción de haplotipos, tema también tratado en esta tesis. Nuestra primera aproximación para la resolución de este ha sido *viaDBG*, que es la primera solución basada en grafos de *de Bruijn* que ofrece resultados comparables a las técnicas vigentes en ensamblaje de genomas víricos, mientras que mantiene la eficiencia de estos grafos. Nuestra segunda aportación es *ViQUF*, que es una mejora natural de su predecesor. *ViQUF* cambia totalmente la algoritmia de *viaDBG*, pero sigue cimentándose en las mismas estructuras aunque con alguna variación que le permite no solo mejorar resultados en tiempo y calidad. Sino que además le permite aportar más información como estimaciones relativa de cada especie en la muestra.

Resumo

Durante as dúas últimas décadas, grazas ao desenvolvemento de novas técnicas de secuenciación, o estudo do xenoma fíxose moi popular para descubrir a variación xenética presente tanto nos humanos como noutros organismos. O modo predominante de análise do xenoma é a través da ensamblaxe de lecturas nunha ou varias cadeas o maior tempo posible. A forma máis tradicional de ensamblar é a que implica lecturas dun só xenoma. Neste campo, na última década xurdiron lecturas de terceira xeración con novos retos para os que non existen solucións eficientes. A primeira contribución que se fixo nesta tese é *Compact-Flye*, unha ferramenta para a montaxe eficiente de lecturas de terceira xeración sobre o algoritmo *Flye*. Esta ferramenta baséase no uso intelixente de estruturas de datos compactas para mellorar os pasos típicos de montaxe, como contar e indexar k-mers. Ademais da montaxe dun xenoma, existen técnicas que buscan ensamblar todos os xenomas contidos nunha determinada mostra. Este conxunto coñécese como conxunto de secuencias múltiples ou reconstrución de haplotipos, tema tamén tratado nesta tesis. O noso primeiro enfoque para resolver isto foi *viaDBG*, que é a primeira solución baseada en gráficos *de Bruijn* que ofrece resultados comparables ás técnicas actuais de ensamblaxe de xenoma viral, mantendo a eficiencia destes gráficos. A nosa segunda incorporación é *ViQUF*, que é unha mellora natural con respecto ao seu predecesor. *ViQUF* cambia completamente o algoritmo de *viaDBG* pero segue baseándose nas mesmas estruturas, aínda que con algunha variación que lle permite non só mellorar os resultados en tempo e calidade. Pero tamén permite achegar máis información como estimacións relativas de cada especie da mostra.

Contents

Chapter 1

Introduction

This chapter presents the problem addressed in this thesis, the objetives, and a discussion of the achieved results. First, Section 1.1 introduces the biological background needed to present the addressed problem. Then, Section 1.2 shows the objetives and Section 1.3 shows a discusion of the achieved results and the conclusions.

1.1 Biological background

1.1.1 Genome, DNA, RNA and genetic variation

DNA is the molecule responsible for storing the genetic information of almost every organism. It is located inside each cell and all cells in one organism have almost the same DNA. DNA encodes the genome, which stores generic and particular traits of every single individual; thus, DNA makes us who we are.

DNA is the final product built from the joining of small pieces known as nucleotides. These nucleotides are the basic component of the DNA and they are built from one molecule of sugar (ribose and deoxyribose for RNA and DNA respectively), a phosphate group, and a *nitrogenous* base. Depending on this base, we have adenine (A), cytosine (C), thymine (T), and guanine (G). In the case of RNA, thymine is changed to uracil (U). The four bases have a peer relationship allowing them to form hydrogen bonds between opposite bases, $A-T$ and $C-G$. Each pair is called a *base pair*, *bp* for short. These bonds are responsible for the double-stranded DNA chain, which allows processes such as DNA replication and transcription. These two strands are known as complementary strands and, depending on the direction, they are called *forward* or *reverse* strands. Although they are called complementary, both strands store the same information. This enables a repair mechanism that corrects the DNA when errors appear during the replication process. Finally, not

all organisms are bistranded: several viruses are single-stranded, but they have lost some important properties such as DNA correction.

The genome length varies greatly depending on the organism. For example, humans have around 3,000 million pairs of bases, while a bacteria, such as for example *Escherichia coli*, have 4.5 million, and viruses only have between 10,000–30,000 base pairs.

In both eukaryotic and prokaryotic organisms, DNA can be copied in two different ways through replication and transcription. Transcription is the process of copying particular regions of the DNA into RNA. The RNA is a molecule highly similar to the DNA but single-stranded and focused on protein production: producing ribosomes, transporting aminoacids, and coding proteins. Therefore, transcription produces three different RNAs: rRNA or ribosomal RNA, which is the basic component of the ribosome; tRNA or transfer RNA, which allows amino acids to be transported in the cytoplasm; and mRNA or messenger RNA, which is the one that will be translated into proteins. mRNA is the translation of particular regions of the genome known as exons. The remaining parts of the genes are known as intronic or non-coding regions, typically interspersed with coding regions. Although it is not the main topic of this thesis, exons, and introns have an interesting relationship and most introns are understudied nowadays with no clear function assigned. On the other hand, replication consists in duplicating the genetic information of an organism to duplicate the cell. During the transcription and replication processes, some mistakes can happen, which are sometimes corrected by enzymes known as polymerases (I, II, II). However, when some of these errors are not corrected during the replication process, mutations are produced. These effects go from harmless, silent mutations, to deleterious, removing some functionality, and stopping an entire protein production. In general, mutations can be divided into three different groups, depending on the structural impact they produce: single nucleotide polymorphism (SNPs), insertions or deletions (indels), or structural variations (SVs). During the last years, multiple effects such as cancer, an increase of aggressiveness, or resistance in some viruses or bacterias have been discovered. They have been directly associated with variations, thus creating enormous interest in the research community. In prokaryotic organisms, there is another way to produce mutations, which is the recombination process.

1.1.2 Genome sequencing

In recent years, biology has joined the group of sciences that seek a solution to their problems in computer science. Biological problems are highly complex due not only to the inherent difficulty of the physicochemical processes but also due to the uncertainty and size of the data they handle. This is what prompted the incorporation of computer techniques into the biological field as early as the last decade of the 20th century. The first major project was the Human Genome Project, where the use of computational techniques led to a dizzying increase in the assembly speed. This incorporation allowed the replacement of more traditional sequencing

techniques, such as Sanger or sequencing by chemical decomposition, with new high-performance versions, known as Next Generation Sequencing (NGS) and Third Generation Sequencing (TGS). As a result of this first contact, computer science has gradually been claimed as the solution to many of today's biological problems and is nowadays an essential element in the development of the field of traditional biology.

Both of them, NGS and TGS, rely on a sequencer to produce a set of reads. Each of these reads covers a portion of the genome. However, depending on the technique, the portion and the quality of the reads change:

- NGS produces high-quality short reads of 100–250 bps (base pairs) with less than 1% of error. It allows paired-end reads to be produced, which are pairs of reads with lengths 100–250 bp separated by an inner distance.
- TGS produces low-quality long reads, which are longer than 10,000 bps with a high rate of error, sometimes higher than 15%.

In either case, each read only covers a portion of the genome, and then it is necessary to reconstruct the genome from these reads. This problem is traditionally called *genome assembly*.

Genome assembly is typically conducted by using graphs. Depending on the graph used, the genome assembly can be *de Bruijn* or *overlap graph*-based. The first one is based on building a representation of the reads and their relations by using k-mers, which are substrings of length k contained in the reads. The second one is based on computing overlaps between the reads and joining them when these overlaps have passed some quality controls. The pruned version of these graphs is known as *string* graphs.

1.1.3 Haplotype reconstruction

The DNA contains *genes*, which are sequences of contiguous nucleotides in the DNA molecule (or ribosomal RNA) containing the information necessary for the development or operation of a physiological function. Each gene is located in a fixed position of the DNA sequence, called *locus*. In genetics, it is common to have different versions of the same functional gene, each of those versions is called an *allele*.

In biological terms, a haplotype is a set of DNA variations, or polymorphisms, that tend to be inherited together. A haplotype can refer to a combination of alleles or to a set of single nucleotide polymorphisms[1] (SNPs) found on the same chromosome[2].

From the computational perspective, the *haplotype reconstruction* problem consists in detecting the haplotype(s) present in a set of raw reads obtained from a biological sample, where abundancies and genomes are unknown. More precisely,

[1] A variation in the DNA sequence that affects a single nucleotide.

[2] A chromosome is a set of DNA that represents a part of all genetic information of an organism.

the haplotype reconstruction consists in producing a set of contigs (sections of the genome) as long as possible, which contains both high-frequency and low-frequency alleles or SNPs correctly placed, as well as the different reconstructed haplotypes. Additionally, depending on the problem, it may be necessary to obtain the relative frequency of each allele.

In this thesis, we will treat one well-known *haplotype reconstruction* problem, which is the *viral haplotype reconstruction*. From a purely biological perspective, *viral haplotype reconstruction* aims to ensemble the viral haplotypes contained in a sample, like for example in an infected person. Because the mutation rate of viruses is much higher than that of other species, especially in the case of RNA viruses or retroviruses such as HIV, the diversity in a sample can be extremely high. However, knowing all the different variations and haplotypes that coexist in a patient plays a key role in target medicine since it helps assess the virulence and better define the current infection status. Therefore, with this knowledge, deciding which therapy is most suitable is much easier. Deep sequencing of intra-host viral populations is becoming an important tool for studying viruses with a growing number of applications, including, for example, drug resistance, immune escape, and epidemiology.

At the time this thesis started, the available solutions had several drawbacks, such as high execution time, lack of sensitivity, and the extremely high number of parameters needed to run the process. This made these tools not too attractive for solving real problems; thus, more generic *metagenomics* tools, like *MetaSPAdes*, were used. These techniques were developed to detect different species in a sample, like finding the different microorganisms in wastewater, but not for the reconstruction of viral haplotypes.

1.2 Objectives

This thesis addresses the genome assembly problem, and inside it, it address two lines of work.

The first one is to give a new solution to the *viral haplotype reconstruction* problem, which can be classified as a metagenomic problem. This includes several complex challenges such as: differentiating between genetic variations and errors; managing genetic recombination between the haplotypes; placing the correct SNPs in the correct place avoiding misassemblies or blending different haplotypes.

However, *viral haplotype reconstruction* has some particular features that separate it from the typical *metagenomics* problem. As explained, in *metagenomics*, the target is to find different species, so similarity bounds are laxer than in the case of *viral haplotype reconstruction*, where we are not dealing with different species, but with haplotypes of the same species. Metagenomic assembly tools usually claim to be able to reconstruct genomes within a similarity range from 1%, but in practice, none can capture such similarity and these variations are discarded. However,

viral haplotype reconstruction has to achieve that level of similarity or even more. Although this makes the problem harder, there are two specific features of the *viral haplotype reconstruction* problem that helps. First, depth coverage in *viral haplotype reconstruction* is always very high, from 20000×[3] and beyond, and, as a consequence of this, the full haplotypes are completely contained in the sample. Furthermore, while in *metagenomics* the sample contains a palette of different organisms, in *viral haplotype reconstruction* samples, they only contain haplotypes evolved from the same ancestor. These properties are convenient and have allowed the development of ad hoc tools that achieve much better accuracy than typical metagenomic tools.

As in regular assembly tasks, *viral haplotype reconstruction* has *reference-based* and *de novo* approaches. Reference-based methods use a previously constructed genome (the reference) to guide the assembly of reads from a sample, whereas *de novo* methods only rely on the raw reads. Reference-based methods are not typically effective when the genomic divergence grows. Furthermore, sometimes it is complex to find an adequate reference since the viruses mutate faster than regular organisms. Until 2019, most of the approaches were reference- or *overlap graphs*-based. The reference-based approaches introduced severe biases removing highly divergent strains or being misassembled. The *overlap graph*-based methods had high time and memory requirements or they relied on greedy data pruning that loses, in complex cases, genomic information. Since de Bruijn graph approaches were barely explored and de Bruijn graphs have proved to be useful in the general assembly context, we face the problem from that perspective.

In this thesis, we present two different algorithms to give a competitive solution for the *viral haplotype reconstruction* based on *de Bruijn* graphs. Although de Bruijn approaches are typically known for being less accurate than *string graph* approaches, we were able to keep competitive and sometimes even better results in terms of accuracy, but lowering time and memory consumption by several orders of magnitude. The first of our approaches was viaDBG, which was the first approach that, keeping the de Bruijn performance properties, obtains competitive results in terms of accuracy. However, viaDBG had several weak points when working with extremely tangled data and it did not provide strain frequency estimation. Therefore, we designed and developed our second algorithm ViQUF that, maintaining or even improving viaDBG's assembly quality, can outperform it in terms of memory and speed, and provides accurate haplotype frequency estimation.

The target of the second line of work in this thesis is to reconstruct a unique genome using TGS reads. In this part of the thesis, we focus on the development of data structures to increase the performance of k-mer counting in assemblers that use TGS reads. The number of k-mers is so high in eukaryotic and prokaryotic organisms that huge amounts of memory and time are required, and thus counting k-mers in samples from large genomes such as the human is not feasible except on

[3]This means that given a position of the genome, on average, there are 20000 reads covering that position.

high-capacity computers. Thus, the design of data structures that consume less space and time during that computation is a relevant problem. This produces software much more scalable, i.e. huge genomes can be assembled on smaller computers. To this end, we study the use of state-of-the-art information compression techniques that allow the information to be stored in the computer's main memory, thus saving space and (potentially) reducing disk access times.

1.2.1 Viral haplotype reconstruction

Both viaDBG [FLPS21] and ViQUF [FLPS22] are based on de Bruijn graphs, which we augment with several techniques to improve the accuracy of the reconstructed haplotype sequences. Basically, we adapt the approximate paired de Bruijn graph [MPC$^+$11] to viral quasispecies assembly. The approximate paired de Bruijn graph integrates the information available in the paired-end reads directly into the de Bruijn graph, which allows this information to be utilised directly in contig assembly. Then viaDBG builds clique graphs to detect haplotypes, while ViQUF is based on flow networks and optimisation problems to be able to simplify the de Bruijn graph.

1.2.1.1 viaDBG

viaDBG method has two main steps, error correction and haplotype inference. The error correction step aims to correct the sequencing errors in the reads by first identifying non-false k-mers in the reads and then applying the LoRDEC [SR14] algorithm, a error correction method for correcting sequencing errors in TGS reads, adapted to our case. The haplotype inference step starts by building a de Bruijn graph, where the paired-end information is added and then, some heuristics are used to polish the paired-end information. Finally, the haplotypes are obtained by splitting the de Bruijn graph nodes based on the paired-end information, and then, the contigs are obtained from this new modified de Bruijn graph.

Given a node A of the de Bruijn graph, if all occurrences of the genome fraction that A represents are from the same haplotype, then all paired information of A occur along some path in the de Bruijn graph. Thus all nodes corresponding to A and its paired information are reachable from each other. On the other hand, if the genome fraction that A represents occurs in several haplotypes, then the paired information of A is likely to span some site containing a mutation, and thus, not all the nodes corresponding to the paired information of A are reachable from each other. viaDBG uses this reachability information to split the de Bruijn graph nodes into different haplotypes.

The reachability information must be carefully analysed. For this, an intermediate support graph, called *Cliques Paired de Bruijn Graph* (CPBG) is built for each pair of adjacent nodes of the de Bruijn graph, combining information from the de Bruijn graph and the paired-end reads. In the CPBG, the target is to find the cliques, since each clique represents a different haplotype.

1.2.1.2 ViQUF

This method also starts with the de Bruijn graph built from non-false k-mers. In the de Bruijn graph, the nonbranching paths, unitigs, are compacted into single nodes producing an assembly graph (AG). For each unitig, we associate a set of unitigs linked to it by paired-end reads. Then, AG is processed by taking every pair of adjacent nodes. For each pair, a new directed acyclic graph (DAG) is built including all unitigs linked to those two nodes by paired-end reads. We then determine a minimum path cover of DAG, where each path represents a haplotype. Therefore, the two nodes for which we computed the considered DAG are divided as many times as paths were found.

For this to work properly, each DAG must be carefully processed to achieve a more reliable graph. This includes transforming each DAG into an offset flow network and solving a min-cost flow problem. With the detected flows, the DAG is corrected, yielding a more reliable graph. Finally, the nodes of the AG are divided based on the path covers of its DAGs, and this results in a new AG, called approximate paired AG (APAG). Once this graph is polished, it is used to derive the contigs and their abundances.

With respect to viaDBG, these are the main differences. First, it employs a mathematically rigorous way of determining the non-false k-mers using kernel density estimation, resulting in a better filtering than the method of viaDBG. Second, ViQUF uses path cover to split the adjacent nodes into haplotypes, whereas viaDBG finds cliques in the reachability graph of paired-end nodes. Third, ViQUF uses path cover to find haplotypes and their abundances; thus, considering the coverage in this stage. However, viaDBG only reports nonbranching paths as contigs.

1.2.2 Improving third-generating sequencing assembly

Genome assemblers for the regular genome assembly have been developing over the past decades. The are really mature and thus it is really difficult to improve their performance with respect the quality of the genome assembly. Moreover, in the case of TGS, Flye [LYK+16] was built upon SPAdes [BNA+12], which in turn, is one of the most used and mature assemblers.

Flye that was recently compared with five state-of-the-art assemblers, obtaining better or comparable assemblies, while it is an order of magnitude faster [KYLP19]. Moreover, Flye obtains longer contigs as it doubles the NGA50 metric. Therefore, its results are impressive in all aspects, with only one weak point, the memory consumption.

Flye works in three steps: i) Building a draft genome assembly and generating consensus contig, ii) Treating repetitive regions, and iii) Polishing the final genome. All of these three steps are critical and need to be treated carefully. Even though all the steps are equally relevant, the most memory-demanding phase is the first one

since it has to quickly process all k-mers in reads and build consensus contigs, which requires having all overlaps between reads.

The high error rate of TGS reads produces a large number of false k-mers. In order to overcome this problem, Flye separates the real and false k-mers by counting the appearances of the k-mers in the reads and selecting only those whose frequency is above a threshold t. While this might sound simple, it actually becomes a hard problem when working with long reads, due to the huge amount of k-mers present in the reads, which makes their indexation extremely difficult in a reasonable time and space. In fact, this problem is the origin of a research line by itself [KDD17, RLC13b, MP11].

Flye relies on Cuckoo hash, which consumes large amounts of space in order to obtain fast access times. We replace Cuckoo hash by a data structure based in bitmaps [FLP21] and two well-known operations of the field of *compact data structures* [Nav16], *rank* and *select*. The bitmaps are space efficient and by using rank and select operations on them, we can simulate complex operations. In addition, rank and select are fast operations, indeed, they can be done in constant time regardless the size of the bitmaps. As a result, the new space efficient verion of Flye developed in this thesis is also faster and less energy demanding than the original one.

1.3 Discussion and conclusions

In the last two decades, computational biology has been an active field of research, especially for topics like assembly, Genome-Wide association studies (GWAS), discovery of metabolic path-ways, and so on.

The common thread of this work is genome assembly, always using a de Bruijn graph as the main basic element. The thesis tackles two lines of work inside this subject. First, genome assembly using NGS reads of viruses to find haplotypes and, second, genome assembly using TGS reads.

Currently, both topics are hot topics in bioinformatics, with new publications almost monthly, thus highly competitive contexts. In these topics, we have made the following contributions:

- **Design and implementation of the first competitive *haplotype reconstruction* algorithm based on de Bruijn graphs with high efficiency and competitive results**. **viaDBG** is the first method that provides a solution to the *haplotype reconstruction* problem keeping the properties of de Bruijn graph approaches and providing competitive results compared with the state-of-the-art. At the time we launched viaDBG, there were no reliable de Bruijn graph methods for *viral haplotype reconstruction*. The previous approaches were inaccurate, and thus unreliable. Furthermore, some of them lost their main feature, speed, because they rely on really

complex algorithms. In our case, we were able to adapt a theoretical approach, approximate paired de Bruijn graphs, to keep the speed and to increase accuracy and reliability. The result was viaDBG, a fast and accurate de Bruijn method for *haplotype reconstruction* in viral samples.

- **ViQUF, a new *viral haplotype reconstruction* based on de Bruijn graphs with strain frequency estimation**. After developing viaDBG, the new tool *Virus-VG* was presented, which introduced a new feature, the estimation of the abundance of the haplotypes present in the sample. However, *Virus-VG* needs pre-assembled contigs to produce extended contigs and abundances. Therefore, speed remains a major issue. Then, we started working on ViQUF, which is a completely new approach since viaDBG misses some information required for the abundance estimation step. Although ViQUF is also a de Bruijn graph approach, the key difference is the use of optimization techniques, instead of cliques as its predecessor. *Virus-VG* was based on a variation graph from which an optimization problem was built and then solved. However, the variation graph building and constraints definition were hard and time-consuming. viaDBG uses the same optimization approach, but with some major changes, we can define a similar optimization problem directly on our graph building phase. Therefore, we can skip the preassembly step and the variation graph building, and thus, our approach is much faster with close results, which sometimes are even more accurate. ViQUF is the first *viral haplotype reconstruction* method based entirely on de Bruijn graphs that allows abundance estimation. Furthermore, it is much faster than viaDBG and gives more robust and reliable results.

- **Memory and time-efficient assembly of TGS reads**. Based on *Flye* assembler, we have designed new methods to improve the assembly of TGS reads. The implemented approach does not rely on modern hashing methods like cuckoo hashing, but on simpler data structures like bitmaps. By using multiple leveled bitmaps and regular hashing, our new method saves space, producing a more linear memory allocation and keeping the high speed of cuckoo's hashing. Therefore, it avoids cuckoo hash table duplications, which penalize space because it demands much more memory than it requires, and time because it needs time to move elements from one place to another in memory. As a final result, we allow the execution of *Flye* on personal computers even for medium-large genomes.

These contributions have been materialized in three different software programs: Compact-Flye, viaDBG, and ViQUF. The software run from a command-line interface and, for viaDBG and ViQUF, a Docker is available. About dependencies, all of them rely on different C++ libraries for network flows, graphs, data structures, and bioinformatics such as Lemon, GATB, boost, and SDSL 2.0. Only ViQUF uses Python for the early and final stage of the algorithm and needs the third-party

commercial software Gurobi. For evaluation purposes, we have used Quast 4.3 and the metagenome assembler version metaQuast 4.3, which allows us to produce several different measures to check the quality of an assembly. We realized that different versions of Quast produce serious differences in evaluation. Since we always run the latest version of the evaluator, results might not be always comparable to other articles.

To further improve the process of TGS, there are different lines of work that need to be developed further. First of all, methods to classify correctly k-mers from TGS reads as genomic or not genomic must be designed. From a sequencing perspective, the easy way to do so is to increase further the sequencing depth in TGS experiments or maybe to increase the reliability of these reads. However, from an algorithmic perspective, the requirements fall on a better correction method, or maybe in some statistical approach that allows for prediction or to classify bases as valid or failures. Another improvement would be to produce longer and less fragmented assemblies, but this applies as well to regular assembly procedures, not only to the assembly of *third generation* reads.

On the other hand, *haplotype reconstruction* challenges, such as *metagenomics*, transcriptomic, or metatranscriptomics, which are some of the most studied bioinformatics topics, have only partial or inaccurate solutions. In the *haplotype reconstruction* studied in this thesis, *viral haplotype reconstruction*, several advances have been made in the last years. Therefore, in the close future, improvement margin is much lower, nevertheless, there is still room for improvement. In de Bruijn graph based methods, which is our field of study, improvements fall on the side of reducing assembly fragmentation, testing more on real data benchmarks, and reducing the complexity of the pruning methods.

- Reducing assembly fragmentation: our developed methods are faster and have comparable results. However, they typically produced more contigs than required. The main reason for that is that, when splitting the nodes to build the APAG, sometimes some extra small cliques are produced or small flow remains in fake paths. Although in most cases, we can filter these residues, sometimes we cannot and, as a final result, more contigs are produced; thus, a more fragmented assembly is generated. This situation is common in complex cases like in the real data benchmark. By pruning a little bit more the cliques graphs or the DAGs in viaDBG and ViQUF, respectively, better results would be achieved.

- Testing more on real data: this is a major problem since we started working on the *viral haplotype reconstruction* topic. There is almost no real information available and only in one ground truth is known, thus the evaluation on these datasets is difficult.

- Reducing the complexity of the pruning methods: the two approaches we have developed, viaDBG and ViQUF, have complex pruning methods when

selecting the right cliques and building the DAG. These methods are both hard to explain, debug and change when something fails or better ideas are suggested. Although it is not clear how to make them easier while keeping the actual performance, methods to reduce the steps or make them easier could be explored.

The suggested improvements are all based on NGS reads. However, during the last couple of years, some new approaches for TGS reads have been published. In contrast to NGS, one or a couple of TGS reads are capable of covering the entire virus genome. Therefore, reconstructions, intuitively at least, seem much simpler. Nevertheless, in *viral haplotype reconstruction* the similarity between strains decreases from 1.0% to 0.1%; thus, given the high error rate of these reads, assembling with such levels of similarity are extremely challenging. In fact, during the last year, we have been actively working with our colleagues at the University of Helsinki to adapt ViQUF to TGS. This adaptation uses the assembly graph with a flow inferred from it. Then, this flow is translated to the set of most parsimonious paths by formulating a problem of integer linear programming (ILP). To do so, we have designed three different scenarios: the standard, based on the network structure, and the flow conservation rule; the inexact, based on the network structure, and the flow conservation rule, but it requires a preflow, not a flow; and the subpath constraints version, based on the network structure, flow conservation rule, and including the solid k-mers in the TGS as subpath constraints. The term subpath constraints has been introduced by Tomescu et al. in [WL22].

In conclusion, *haplotype reconstruction* challenges are still under active development with several solutions that work incredibly well in simulated data but they struggle with real information. In our field of study, viral *haplotype reconstruction*, several solutions have been proposed during the last years. Although, all of them have some flaws when working on real data, basically because there is no real data benchmark posted, they work extremely well in simulated datasets. Therefore, the next required step would be to get some real data information to assay all the available tools and see how they really perform. Furthermore, this makes it possible to replace synthetic data that produces unexplained gaps with real information.

1.4 Articles published during the thesis span

The results of the research carried out throughout this doctoral thesis have been published in the following international journals:

- ViQUF: de novo Viral Quasispecies reconstruction using Unitig-based Flow networks. Borja Freire Castro, Susana Ladra, José R. Paramá, Leena Salmela, July 2022, *IEEE/ACM Transactions on computational biology and bioinformatics* DOI: 10.1109/TCBB.2022.3190282.

- Memory-Efficient Assembly using Flye. Borja Freire Castro, Susana Ladra, José R. Paramá, September 2021, *IEEE/ACM Transactions on computational biology and bioinformatics*, DOI: 10.1109/TCBB.2021.3108843.

- Inference of viral quasispecies with a paired de Bruijn graph. Borja Freire Castro, Susana Ladra, José R. Paramá, Leena Salmela, February 2021, *Bioinformatics* 37(4), pp. 473-481. DOI: 10.1093/bioinformatics/btaa782.

Other publications during my doctoral period:

- Compact and Efficient Representation of General Graph Databases. Sandra Álvarez-García,Borja Freire Castro, Susana Ladra, Oscar Pedreira, September 2019, *Knowledge and Information Systems*, DOI:10.1007/s10115-018-1275-x.

- Parallel Feature Selection for Distributed-Memory Clusters. Jorge González-Domínguez, Verónica Bolón-Canedo, Borja Freire Castro, Juan Touriño, September 2019, *Information Sciences*, DOI:10.1016/j.ins.2019.01.050.

1.5 Projects not published yet

Apart from the previous work already published, some work other work has been developed during these past four years. Nevertheless, they have not been published yet. These works are:

- Singular spectrum drugs design. During my stay in Chile, we started working on unique pattern discoveries to allow automatic drug design. The recent discovery of the existence of a unique pattern in the bacteria *Helicobacter pylori* has raised the question of "Is there a unique pattern in all bacteria?". By following this idea, we aimed to answer the topic of antibiotic resistance. To do so, we used some self-polished versions of recently microbiome databases published in indexed journals. The objective is, by giving the genomic information for a bacterium, raw reads or complete genome assembled, detect a unique pattern. If there are unique patterns in the bacteria, then there may be patterns directly related to the survivability of the bacteria. Therefore, a unique antibiotic can be designed to target only that bacteria. This project was conducted in a joint work with Cecilia Hernández and Alexis Salas, researchers at the School of Medicine and Engineering at the University of Concepción (Chile). Some good results were achieved and the project is still under development.

- Analysis over the conservation index in the mitochondrial genomes in cancer and healthy cells. This is a joint work with Anton Vila Sanjurjo and Jose Antonio Vilar Fernández, researchers at University of A Coruña. The goal of the analysis was to verify the hypothesis that the number of variations in places with high conservation index increases in cases of cancer. As a final

result, we adjusted both splines and polynomial regressions over the original data, the first and the second derivatives and for all an effect existed. At the moment, we are waiting for the biological explanation of these results.

As a side study of this work, we hypothesized to study the concurrence of mutations in the mitochondrial genome by using collaborative filtering methods. Since the mitochondrial genome is short, around 40000 bps, and there are several already fully assembled. It is feasible to build a matrix M of rows the assemblies and the columns each base with a stored variation. Therefore, a Singular Value Decomposition can be performed over this matrix, $M = U\Sigma V^T$, and distance metrics can be calculated in this matrix to build clusters that associate variants by similarity. As a result, we were able to associate multiple mutations in the mitochondrial genome; however, the correctness of these is pending to be assayed by experts in the domain.

- Lyndon word enumeration. During the last year, we started a collaboration with Hideo Bannai and Dominik Köppl, researchers at the Dental University of Tokyo. In this collaboration, we focused on pure theoretical computer science topics such as *bijective Burrows Wheeler* transformation and *Lyndon* word enumeration. As a result, we have written a conference paper which is pending for publication.

Chapter 2

Articles

Bioinformatics, 37(4), 2021, 473–481
doi: 10.1093/bioinformatics/btaa782
Advance Access Publication Date: 14 September 2020
Original Paper

Sequence analysis

Inference of viral quasispecies with a paired de Bruijn graph

Borja Freire[1], Susana Ladra [1], Jose R. Paramá [1,*] and Leena Salmela [2]

[1]Department of Computer Science and Information Technologies, Facultade de Informática, Universidade da Coruña, Centro de investigación CITIC, A Coruña, Spain and [2]Department of Computer Science, Helsinki Institute for Information Technology, University of Helsinki, Helsinki, Finland

*To whom correspondence should be addressed.

Associate Editor: Bonnie Berger

Received on June 13, 2019; revised on March 11, 2020; editorial decision on August 31, 2020; accepted on September 2, 2020

Abstract

Motivation: RNA viruses exhibit a high mutation rate and thus they exist in infected cells as a population of closely related strains called viral quasispecies. The viral quasispecies assembly problem asks to characterize the quasispecies present in a sample from high-throughput sequencing data. We study the *de novo* version of the problem, where reference sequences of the quasispecies are not available. Current methods for assembling viral quasispecies are either based on overlap graphs or on de Bruijn graphs. Overlap graph-based methods tend to be accurate but slow, whereas de Bruijn graph-based methods are fast but less accurate.

Results: We present viaDBG, which is a fast and accurate de Bruijn graph-based tool for *de novo* assembly of viral quasispecies. We first iteratively correct sequencing errors in the reads, which allows us to use large *k*-mers in the de Bruijn graph. To incorporate the paired-end information in the graph, we also adapt the paired de Bruijn graph for viral quasispecies assembly. These features enable the use of long-range information in contig construction without compromising the speed of de Bruijn graph-based approaches. Our experimental results show that viaDBG is both accurate and fast, whereas previous methods are either fast or accurate but not both. In particular, viaDBG has comparable or better accuracy than SAVAGE, while being at least nine times faster. Furthermore, the speed of viaDBG is comparable to PEHaplo but viaDBG is able to retrieve also low abundance quasispecies, which are often missed by PEHaplo.

Availability and implementation: viaDBG is implemented in C++ and it is publicly available at https://bitbucket.org/bfreirec1/viadbg. All datasets used in this article are publicly available at https://bitbucket.org/bfreirec1/data-viadbg/.

Contact: jose.parama@udc.es

Supplementary information: Supplementary data are available at *Bioinformatics* online.

1 Introduction

RNA viruses such as the human immunodeficiency virus (HIV), the Zika virus (ZIKV) and the hepatitis C virus (HCV) exhibit a high mutation rate (Duffy *et al.*, 2008). Thus, their populations in a host organism consist of a number of different strains, which are differentiated from each other by mutations in the genome. In the context of viruses, the collection of these strains is called a viral quasispecies (Domingo *et al.*, 2012; Holmes, 2009). Each of the strains in the viral quasispecies can be characterized by its haplotypic sequence. When studying a viral sample, it is important to capture all strains present in the sample, because different viral strains may have a different response to the available treatments and drugs (Domingo *et al.*, 2012).

High-throughput sequencing has provided a way to investigate viral samples in detail to characterize the different strains present in the sample and their abundances. However, although viral genomes are short, there are challenges that are specific to the analysis of viral quasispecies data. First, the presence of similar strains in the data makes it difficult to assign the reads to different haplotypic sequences. Secondly, viral samples are typically sequenced to a much deeper coverage than e.g samples for genomic or metagenomic sequencing. This presents a challenge for developing computationally efficient tools for reads that frequently overlap each other. Therefore on viral samples, standard tools for genome assembly or metagenomics produce fragmented assemblies that do not properly capture all strains present in the sample [see e.g. Baaijens *et al.* (2017)].

Methods for assembling viral quasispecies from high-throughput sequencing data are classified into two approaches, reference-based and *de novo* approaches (Posada-Cespedes *et al.*, 2017). The reference-based approaches first align the reads to the reference sequence. Many of these approaches then cluster the reads to haplotypes by enumerating maximum cliques (Töpfer *et al.*, 2014), assembling the reads (Jayasundara *et al.*, 2015), using Hidden Markov Models (Töpfer *et al.*, 2013), or using probabilistic

modelling (Ahn and Vikalo, 2018; Barik *et al.*, 2018; Prabhakaran *et al.*, 2014; Zagordi *et al.*, 2011). Instead of clustering reads, Knyazev *et al.* (2019) cluster the observed variants to haplotypes. Prosperi and Salemi (2012) divide the reference into overlapping intervals, construct local haplotypes for each interval, and finally merge them to global haplotypes. These reference-based approaches can be effective if a good quality reference is available. However, it has been shown that using reference genomes can bias the reconstruction significantly (Baaijens *et al.*, 2017; Töpfer *et al.*, 2014). Thus, a number of *de novo* viral quasispecies assemblers, which do not need a reference sequence, have been developed. We are aware of three tools fitting this category, MLEHaplo (Malhotra *et al.*, 2015), SAVAGE (Baaijens *et al.*, 2017) and PEHaplo (Chen *et al.*, 2018). The *de novo* assemblers typically cannot assemble each strain into a single haplotype but instead produce a set of contigs. Baaijens *et al.* (2019) have recently proposed a method that takes as input contigs produced by a *de novo* viral quasispecies assembler and uses frequency information to further merge these into global haplotypes. In this work, we focus on the *de novo* contig assembly of viral quasispecies data.

Similar to the most successful genome assemblers for bacterial and eukaryotic genomes, *de novo* viral quasispecies assemblers use either an overlap graph or a de Bruijn graph (DBG) to represent the sequencing data. See e.g. Nagarajan and Pop (2013) for a discussion on genome assembly approaches. An overlap graph is constructed by finding all pairwise overlaps between the sequencing reads. Given the deep sequencing of viral data, the number of actual overlaps between the reads approaches the quadratic worst-case limit, and thus this step could be computationally expensive. On the other hand, methods based on overlap graphs, such as SAVAGE, produce very accurate assemblies, because the overlap graph captures well the long-range similarities between the reads. PEHaplo introduces a different trade-off for overlap graph-based approaches by introducing a technique to reduce the number of reads. It is thus much faster, but unfortunately also less accurate. The DBG-based methods, such as MLEHaplo, do not need to perform computationally intensive overlap computations between the reads. Instead, they decompose the reads into k-mers and construct a DBG, where the $k-1$-mers are the nodes of the graph and an edge is added between two nodes if the corresponding k-mer is present in the read set. Because k-mers can be extracted by a linear scan over the reads, these approaches are computationally efficient. However, they are not able to optimally use long-range information available in full-length reads and thus the assemblies they produce tend to be more fragmented and less accurate.

High-throughput sequencing reads, such as Illumina reads, are typically paired-end reads. Many of the viral quasispecies assemblers use heuristics to incorporate the paired-end information. SAVAGE merges read pairs when the pairs overlap each other and it accepts overlaps involving paired-end reads only if both pairs are involved in the overlap and their orientation in the overlap is the same. PEHaplo uses heuristics to prune the overlap graph based on paired-end information and it uses paired-end information as a guidance for finding paths in the overlap graph. PEHaplo also includes a post-assembly step where contigs are split based on paired-end alignments. MLEHaplo formulates the viral quasispecies assembly problem as finding a path cover with maximum score from paired-end reads in a DBG. This problem is shown to be NP-hard and thus MLEHaplo implements a heuristic path finding algorithm for this problem.

We present viral assembly with paired DBG (viaDBG), a fast and accurate tool for viral quasispecies assembly. Our method is based on DBGs, which we augment with several techniques to improve the accuracy of the reconstructed haplotypic sequences. First we employ an iterative error correction method with increasing k-mer sizes. This allows us to use large k-mers in the final assembly enabling the use of long-range information in the DBG. Furthermore, we adapt the approximate paired de Bruijn graph (APDB) (Medvedev *et al.*, 2011) to viral quasispecies assembly. Whereas almost every assembler nowadays applies paired-end information in the post-processing phase, where contigs are merged and/or topologically sorted to create scaffolds, in the APDB the paired-end information is added to the DBG.

Our experiments show that on both synthetic and real data, viaDBG is among both the most accurate methods and the fastest methods, whereas previous tools are either accurate or fast but not both. For example, viaDBG is up to 43 times faster than SAVAGE and produces assemblies with comparable accuracy. Furthermore, viaDBG is able to recover also low-abundance strains, which are lost or inaccurately assembled by PEHaplo, while matching the speed of PEHaplo. On real sequencing data, viaDBG produces assemblies with three times as high $N50$ values as SAVAGE while being nine times faster. The speed of viaDBG is comparable to PEHaplo on this dataset but depending on whether we look at polished or unpolished contigs, PEHaplo either mixes the strains or produces a 30% lower $N50$ value than viaDBG, while viaDBG produces accurate results. The DBG-based approach makes viaDBG efficient, whereas the accuracy of viaDBG is due to using a large k in the DBG and the systematic use of paired-end information.

2 Materials and methods

2.1 Background

2.1.1 Error correction by LoRDEC

LoRDEC (Salmela and Rivals, 2014) is a hybrid error correction method for correcting sequencing errors in *third-generation sequencing* reads with the help of accurate short reads. LoRDEC defines a k-mer as solid if it occurs at least t times in the short-read data, where t is the abundance threshold. The solid k-mers are then used to build a DBG. The third-generation sequencing reads are then processed one at a time. First, solid k-mers in the read are identified and the regions between solid k-mers are called weak. Then, for each weak region between two solid k-mers, LoRDEC finds the best matching path in the DBG between the two solid k-mers. This path is used to correct the weak region in the read. Finally, the weak ends of the read are aligned to the DBG starting from the extremal solid k-mer and the weak ends are corrected according to the found paths. To limit the runtime of the method, LoRDEC abandons the search for the best alignment if there are too many branches in the DBG.

2.1.2 Approximate paired DBG

Medvedev *et al.* (2011) presented the APDB to leverage paired-end information directly in contig assembly. To build the APDB, they first extract all bilabels from the paired-end reads. A bilabel is a pair of k-mers (A, B) such that A occurs in position p in a left-hand read and B occurs in position p in the corresponding right-hand read. Two bilabels (A, B) and (C, D) are merged if $A = C$ and B is reachable from D or vice versa. The merged bilabels form the edges of the APDB and thus the edges have the form (A, S), where A is a k-mer and S is a set of k-mers. An edge (A, S) connects two nodes, $(\mathrm{pref}(A), \mathrm{pref}(S))$ and $(\mathrm{suf}(A), \mathrm{suf}(S))$, where $\mathrm{pref}(A)$ $(\mathrm{suf}(A))$ is the $k-1$ length prefix (suffix) of the k-mer A and $\mathrm{pref}(S)$ $(\mathrm{suf}(S))$ is the set of $k-1$ length prefixes (suffixes) of all the k-mers in the set S.

Unfortunately, APDB is not directly applicable to viral quasispecies assembly. Consider a case where the left-hand reads derive from a region of the genome where two strains are equal and the right-hand reads derive from a region where the strains differ by a single SNP. When we extract bilabels from these reads, the left k-mers will be the same in both strains but the extracted right k-mers can be the same or different depending on whether they cover the SNP or not. Let us suppose that we have extracted bilabels (A, B), (A, B') and (A, C) from the reads, where B and C occur in the first strain and B' and C in the second strain. Because B is reachable from C and B' is reachable from C, all these bilabels are merged into a single edge in APDB. This is acceptable if the goal is to construct a single genomic sequence but not for viral quasispecies assembly where we need to construct all the strains. Here, we have devised a method to differentiate such bilabels correctly. The key idea is to merge a set of bilabels only if all right-hand k-mers are pairwise reachable from each other.

2.2 Overview of our method

Figure 1 shows the main steps followed by viaDBG. The main difference with respect to typical assembly methods based on DBGs is the use of paired-end information in an early stage. Paired-end reads are composed by two reads, which are the two extremes (left- and right-hand) of a sequencing fragment. The insert size is the number of base pairs between the two reads. We will use Δ to denote the maximum error in the insert size.

2.3 Error correction

The error rate of paired-end short reads is low, which makes them suitable for assembly methods based on the DBG. It has been shown that longer k-mers lead to better assembly, but the probability of getting erroneous k-mers also increases. Therefore, we devote the first step to remove sequencing errors from reads, to obtain longer correct k-mers, and thus reducing the erroneous information that would lead to shorter contigs, lower genome fraction recovered and/or more misassemblies.

The error correction involves two steps: (i) selection and classification of solid k-mers that, as in the case of LoRDEC, are the k-mers whose abundance in the reads is higher than a threshold and (ii) reads correction.

2.3.1 Selection of solid k-mers

A k-mer is genomic if it appears in at least one strain and a non-genomic k-mer does not appear in any of the strains in the sample. As in LoRDEC, we select solid k-mers based on their frequency of appearance, assuming that genomic k-mers are more frequent than the non-genomic ones. Then, the whole read set is traversed and each k-mer is classified as solid or not solid. Because of the conservative selection of the threshold, we expect the non-solid regions to be compounded with erroneous information. This is simple, but the problem is to determine the threshold.

In our work, the search of that value is based on the following idea. Let us first consider the histogram of the number of different k-mers that occur at each frequency, i.e. for each frequency f_i, we plot $n(f_i)$, which is the number of different k-mers occurring f_i times. Then, we expect to find a change in the trend in the histogram among the number of different k-mers having low frequencies (non-genomic k-mers) and those having higher frequencies (genomic k-mers). On one hand, the number of different non-genomic k-mers decreases as the frequency increases. On the other hand, the number of different genomic k-mers, which have higher frequencies, starts outnumbering the number of different non-genomic k-mers. Thus, we will use the starting position of that change of trend in the histogram as the threshold to detect solid k-mers.

More concretely, we will search for a region in the histogram where there is a frequency whose count is lower than most of the counters for the frequencies in the succeeding zone. We make use of a fixed window size N to find the frequency f_t where that change of trend starts. We define t as the smallest i such that $f_i \geq 1$ and

$$|\{f_j | f_i \leq f_j \leq f_{i+N} \text{ and } n(f_j) > n(f_i)\}| \geq N/2.$$

In the Supplementary Material, we show that the choice of N is easy, by showing that with a wide range of different window values, the performance of viaDBG does not differ significantly. By default, we use $N = 16$.

2.3.2 Error correction algorithm

To correct sequencing errors in the read, we adapted the LoRDEC algorithm (Salmela and Rivals, 2014) for viral quasispecies data. We use the solid k-mers identified above to build a DBG and then align all the reads to the DBG to correct them. If the abundance of a strain is such that the corresponding k-mers are solid, the strain is present as a path in the DBG. The reads are corrected by aligning them to this graph and choosing the alignment with the smallest edit distance between the read and path in the graph. The reads are expected to align best against the path representing the strain they derive from and thus most of them are corrected towards the correct haplotypic sequence. Therefore, this algorithm is well suited for correction of viral quasispecies data.

We made three further changes to better adapt the algorithm for viral quasispecies data. First, after building the DBG using the solid k-mers, we polish it by removing short tips, i.e. short paths where the first node has out-degree larger than one and the last node has out-degree zero. Secondly, we only correct the part of the reads between the leftmost and rightmost solid k-mer because the algorithm is less accurate on the read ends when only one end of the alignment is anchored on solid k-mers. Third, in viral quasispecies data, it is not necessary to abandon the search for best alignment if there are too many branches in the DBG because the genomes are smaller and the DBG is less tangled. Therefore, this limitation was removed from the algorithm.

Once reads have been corrected, the k-mer size is doubled and the reads are corrected again. This process is repeated three times. In the first iteration when k is small, the set of solid k-mers contains most genomic k-mers and some non-genomic k-mers that are caused by the same sequencing error occurring in the same locus in several reads. Still our method can correct most errors already at this stage because most k-mers including an error are unique even for a small k. Once most errors have been corrected in the first iteration, we can increase k because longer k-mers are now expected to be correct. Because longer k-mers span more variants, k-mers originating from different strains are separated better from each other. Thus, also the abundance of erroneous k-mers becomes lower and now less of the erroneous k-mers are classified as solid. This allows us to further correct some sequencing errors in the next iterations.

2.4 Haplotype inference using paired-end reads

As seen in Figure 1, the haplotype inference is applied in several steps. To illustrate them, Figure 2 shows an example workflow.

First, a regular DBG is built with the solid k-mers obtained in the previous step. Then, we retrieve the unitigs of the DBG, and for each unitig, we assign a representative k-mer, which will be used later in the process. Next, each k-mer is associated with a set of paired k-mers. Given a k-mer A and a paired-end read where A appears at position p of the left-hand read, B is a paired k-mer of A if B occurs at position p of the right-hand read. Next, we polish the paired-end information, and finally we modify the DBG. If all occurrences of the k-mer A are from the same strain, then all paired k-mers of A occur along some path in the DBG. Thus, they are all reachable from each other. On the other hand, if the k-mer A occurs in several strains, then the paired k-mers are likely to span some site containing a mutation. Note that, the paired k-mers span an area larger than k in the haplotypic sequences. Therefore, they are not all reachable from each other but it still holds that the paired k-mers

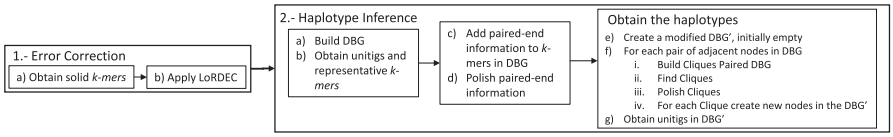


Fig. 1. Overview of viaDBG. Our method has two main steps, error correction and haplotype inference. The error correction step aims to correct the sequencing errors in the reads by first identifying solid k-mers in the reads and then applying the LoRDEC algorithm. The haplotype inference step starts by building a DBG and obtaining unitigs. The paired-end information is then added to the DBG and some heuristics are used to polish the paired-end information. Finally, the haplotypes are obtained by splitting the DBG nodes based on the paired-end information and obtaining unitigs from this modified DBG

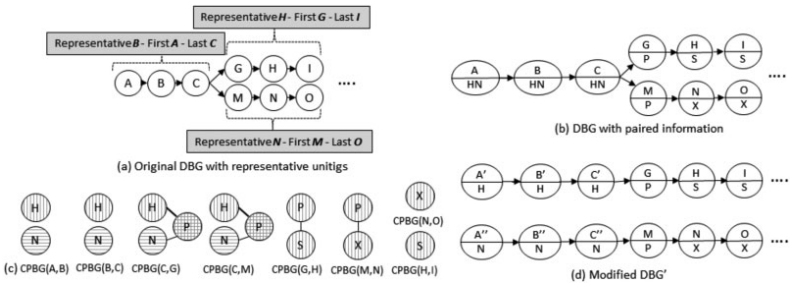


Fig. 2. Example of the different steps of haplotype inference. (**a**) First we build a DBG using all solid k-mers in the reads. Unitigs are then identified and a representative k-mer is assigned to each unitig. (**b**) Next, we augment the graph with the paired-end information. P, S and X are representative k-mers of unitigs not shown in the figure. (**c**) A CPBG is built for each adjacent pair of k-mers in the DBG. Nodes of CPBG are the paired k-mers of the adjacent pair of k-mers and edges between the nodes are added if there is a path between the corresponding k-mers in the DBG. (**d**) Finally for each CPBG, we find cliques and each clique is used to split the nodes of the DBG

originating from the same strain are all reachable from each other. We will use this reachability information to split the DBG nodes into different strains. Finally, the contigs are retrieved from this new DBG. Next, we explain each of these steps in detail.

2.4.1 Getting unitigs and representative k-mers

A unitig is a unary path in the DBG, i.e. a path where all nodes have in-degree and out-degree equal to one except for the first and last nodes. Unitigs always belong to the final genome/s. Therefore, some assemblers, such as SPAdes (Bankevich *et al.*, 2012), condense unitigs into single nodes to compact the graph without losing information.

In our case, for each unitig, we extract three elements: first, middle and last k-mers. The middle k-mer serves as a *representative* of the unitig, whereas the first and last k-mers are used to determine if there is a path from one unitig to another.

Figure 2a shows the DBG of our running example where unitigs are delimited by a brace. The representative k-mer and the first and last k-mers are also displayed in a grey box.

2.4.2 Adding paired-end information for each k-mer

One of the key features of our method is the use of the paired-end information, as it provides additional clues of the actual strains during the traversal of the DBG.

Let U be the set of paired-end reads. Given a read $r \in U$, $L(r)$ is the left-hand, $R(r)$ is the right-hand read and $L(r)[l \dots m]$ ($R(r)[l \dots m]$) are the base pairs at positions $l \dots m$ of $L(r)$ ($R(r)$). For each k-mer A, our method needs to compute its set of paired k-mers $P(A) = \{M | M$ is a solid $k-$mer and, $\exists\ r_x \in U$ and a position j such that $L(r_x)[j \dots j+k-1] = A$ and $R(r_x)[j \dots j+k-1] = M\}$.

Figure 3 shows an example where the k-mer A appears in the left-hand part in two reads (r_x and r_y). Then, the solid k-mers M and L, which appear in the same positions of the right-hand parts, form $P(A)$.

To avoid excessive memory usage, we do not store all paired k-mers. Instead, for each k-mer in $P(A)$, we find the unitig to which it belongs and replace that k-mer with the representative k-mer of the unitig.

Observe in Figure 2b, e.g. that k-mer A has two paired k-mers H and N, which are the representative k-mers of the unitigs GHI and MNO, respectively.

2.4.3 Polishing paired-end information

In this step, for each solid k-mer A, its $P(A)$ is polished. This is needed since sometimes the variance of the insert size can be larger than the used Δ, as the insert size distribution can be modelled with a normal distribution. Therefore, we design a paired-end polishing method that removes outliers with large variance in the insert size, while avoiding the removal of low abundance strains.

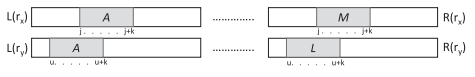


Fig. 3. Extracting paired k-mers from paired-end reads. *P(A)=(M, L)*

Let $\text{freq}(A, M)$ be the frequency of the appearance of the k-mer pair (A, M) in paired-end reads $r \in U$ such that $M \in P(A)$. Because the insert size is normally distributed, the frequency of a k-mer pair with insert size close to the mean is expected to have a high frequency, whereas a k-mer pair with insert size far from the mean is expected to have a low frequency. Furthermore, if the insert size of a k-mer pair (A, M) is close to the mean, then within a short distance from the node corresponding to M in the DBG, we expect to see many other k-mers L that are also in $P(A)$ and have a frequency $\text{freq}(A, L) \geq 1$. Again, this is not expected for a k-mer pair whose insert size is far from the mean. We combine these two effects into a smoothed frequency $\text{freq}'(A, M)$, which is defined as follows:

$$\text{freq}'(A, M) = \min \begin{cases} \text{freq}(A, M) + \\ \\ \max - \text{threshold} \end{cases} \begin{array}{l} \\ |\{L|\ \text{freq}(A, L) \geq 1 \\ \text{and}\, d(M, L) < \max - \text{path} - \text{len}\}|, \end{array}$$

where $d(M, L)$ denotes the distance between M and L in the DBG and we set max-threshold to 40 and max-path-len to 20. Finally, we keep only those paired k-mers whose frequency is within top 85%. We experimentally found that those values work well in practice for all cases.

We limit the smoothed frequency by max-threshold to preserve low abundance haplotypes. Without such limit, the frequency of paired k-mers of high abundance strains with higher divergence from the mean insert size often gets higher than the frequency of paired k-mers of low abundance strains with insert size close to the mean value. This is especially important when relative abundances are around 1–2%.

Currently, this step is the bottleneck of the algorithm. We need to compute the distance between $\frac{n(n-1)}{2}$ pairs per node, where n is the number of pairs in the list of paired k-mers of a given k-mer.

2.4.4 Obtaining the haplotypes

This subsection describes in detail the third block of Step 2 of Figure 1 [steps labelled 2.(e), 2.(f) and 2.(g)]. The haplotypes are obtained by splitting the nodes of the DBG built in Step 2.(a), based on the paired-end information and obtaining unitigs from this modified DBG. Therefore, the Step 2.(e) starts by creating a new empty DBG'.

- *Step 2.(f) i*: for each pair of adjacent nodes *(A, B)* of the DBG, a *Cliques Paired de Bruijn Graph* (CPBG) graph is built. *CPBG(A,*

B) is an undirected graph. The paired *k*-mers of *A* and *B* are the nodes of *CPBG(A, B)*, i.e. the set of nodes is $P(A) \cup P(B)$. There is an edge between two nodes *U*, *V* in the *CPBG(A, B)* if there is a path of length $\leq 2\Delta$ in the DBG from *last*(*U*) to *first*(*V*) or from *last*(*V*) to *first*(*U*) [*first*(*K*) is the first *k*-mer of the unitig of which *K* is the representative, while *last*(*K*) is the last *k*-mer].

Observe that we are computing the CPBG of *k*-mers *A* and *B* that are adjacent in the DBG. Therefore, their paired *k*-mers (separated from *A* and *B*, on average, by the insert size) would also be neighbours in the DBG since they ideally differ by one base pair as well, or they would be very close to each other, due to the insert size error, forward and backward, i.e. 2Δ. Therefore, in *CPBG(A, B)*, we link the paired *k*-mers of *A* and *B* that are connected by a path of the DBG of size at most 2Δ.

Figure 2c shows the CPBG of all pairs of adjacent nodes in the DBG of our example. For example, observe in *CPBG(A, B)* that the nodes are the paired *k*-mers of *A* and *B*, which are *H* and *N* in both cases. However, there is no path in the DBG connecting *H* and *N*, and thus, in the CPBG, there is not an edge linking them. In the case of *CPBG(C, G)*, there is an edge between *H* and *P*, since there is a path of length at most 2Δ in the DBG that connects them (not shown in the DBG of Fig. 2 to avoid cluttering the figure). Similarly, there is an edge connecting *N* and *P*.

Here, we can see the other important benefit of using representatives. Observe that in order to determine whether there is an edge connecting a pair of nodes *U* and *V* of a CPBG, the algorithm has to find paths in the DBG, between the unitigs of the DBG corresponding to *U* and *V*. Therefore, decreasing the number of nodes of the CPBG speeds up this process.

- *Step 2.(f) ii*: for each CPBG, we obtain all its maximal cliques. A clique is a set of nodes of the graph where all nodes are connected to each other.

  In Figure 2c, observe the *CPBG(C, G)*. There are two cliques; the first one is formed by *H* and *P*, and the other by *N* and *P*. Conceptually, cliques are sets of *k*-mers that belong to the same haplotypic sequence. Since all the paired-end *k*-mers in the clique reach or are reached by others in the DBG, it means that there is one strain that gathers them together.

  However, when the graph is tangled, it is possible to find paths in the DBG for two *k*-mers that do not belong to the same strain, and this may produce fake cliques. Therefore, we select the cliques that are supported by the frequency of appearance of their *k*-mers, more precisely, we select those cliques whose nodes appear in more reads and are more linked to other nodes in the DBG. Full details of this process are given in the Supplementary Section S2. Choosing a large value of *k* makes the graph less tangled and thus alleviates this problem. Also using a small Δ helps because even in a tangled graph, shorter paths are less likely to be incorrect.

  We obtain another benefit by using representative *k*-mers, since the CPBG is not a large graph, maximal cliques can be found with lower computational cost than in the case of using all *k*-mers.
- *Step 2.(f) iii*: because of errors in reads, repetitive sections and shared strain regions, wrong cliques can be created. We use several heuristics to polish the cliques.
  - We remove small cliques because they often rise from erroneous *k*-mers.
  - Shared strain regions produce cliques where all *k*-mers are paired with *A* while a subset of them is also paired with *B*, i.e. all nodes of the clique are in $P(A)$, and some nodes, but not all, are in $P(B)$. To keep strains with shared regions separate, we also remove these cliques.
  - Let $\mathcal{SC}$ be the set of all cliques found so far. When two cliques $\mathcal{C}\ell_x,\ \mathcal{C}\ell_y \in \mathcal{SC}$ are almost the same, we remove the smallest one because such cliques can arise from sequencing errors. More precisely two cliques are considered almost the same when $|(\mathcal{C}\ell_x \cap \mathcal{C}\ell_y)| \geq R * \min(|\mathcal{C}\ell_x|, |\mathcal{C}\ell_y|)$, where *R* is a threshold value. By default, we use $R = 90\%$.
- *Step 2.(f) iv*: for each pair of adjacent nodes *A* and *B* in DBG, we take the set of cliques $\mathcal{SC}_{CPBG(A,B)}$ of *CPBG(A, B)* and, for each clique $\mathcal{C}\ell_x \in \mathcal{SC}_{CPBG(A,B)}$: if $\mathcal{C}\ell_x$ has nodes of $P(A)$ and $P(B)$, then the nodes $A_{P_A \cap \mathcal{C}\ell_x}$ and $B_{P_B \cap \mathcal{C}\ell_x}$ are added to *DBG'*, unless they are already in *DBG'*. $A_{P_A \cap \mathcal{C}\ell_x}$ is a node corresponding to the *k*-mer *A* having paired-end information $P_A \cap \mathcal{C}\ell_x$, similarly $B_{P_B \cap \mathcal{C}\ell_x}$ is a node corresponding to *B* having paired-end information $P_B \cap \mathcal{C}\ell_x$.

  In the case of nodes *C* and *G* of the example of Figure 2, their *CPBG(C, G)* has two cliques $\mathcal{C}\ell_1 = \{H, P\}$ and $\mathcal{C}\ell_2 = \{N, P\}$. Then, a new node C' is created due to the existence of $\mathcal{C}\ell_1$, with paired information $P(C) \cap \mathcal{C}\ell_1 = \{H, N\} \cap \{H, P\} = \{H\}$, as seen in Figure 2d. C" is derived from the clique $\mathcal{C}\ell_2$, thus, this new node has as paired information $P(C) \cap \mathcal{C}\ell_2 = \{H, N\} \cap \{N, P\} = \{N\}$. Next, *G* is processed accordingly, producing only one version with paired info *P*. These nodes are added to *DBG'*.

  Observe that, in *DBG'*, C' and C" correspond to the same *k*-mer but those nodes have different paired information, which means that they correspond to different strains.
- *Step 2.(g)*: the last step of the algorithm enumerates the unitigs in the new *DBG'*. As a result of the adaptations based on the CPBG analysis, unitigs are expected to be much longer than in the previous DBG.

3 Results

We compare viaDBG with previous methods for *de novo* viral quasispecies assembly. We also include SPAdes (Bankevich *et al.*, 2012) and metaSPAdes (Nurk *et al.*, 2017) in the comparison to show that viaDBG improves upon general approaches for genome assembly and metagenomic assembly in the case of viral data. We omit some comparisons, such as the reference-based approaches PredictHaplo (Prabhakaran *et al.*, 2014) and ShoRAH (Zagordi *et al.*, 2011), as Baaijens *et al.* (2017) have shown that SAVAGE outperforms both of them. We perform experiments both on simulated and real Illumina MiSeq data.

In the case of *de novo* viral quasispecies assemblers, we compared viaDBG with SAVAGE (Baaijens *et al.*, 2017), which has proven to be the most precise tool among the whole *de novo* assemblers for viral quasispecies, and with PEHaplo (Chen *et al.*, 2018), which is the last released state-of-the-art tool. Real data were trimmed using CutAdapt (Martin, 2011), removing primers, low quality and extremely short reads. In the case of SAVAGE, whose authors highly encourage the usage of PEAR (Zhang *et al.*, 2014), it was only applied to the dataset HIV-real described in Section 3.1.1 and to the dataset HCV-10 described in Section 3.1.2, since for the rest of the datasets, SAVAGE could not complete the assembly—due to memory crash—when running on the result of applying PEAR. In the case of PEHaplo, PEAR was not applied since their authors discourage its usage. PEAR was not applied on synthetic datasets for viaDBG. However, we used PEAR on the real datasets (HIV-real and the real ZIKV and HCV samples) for viaDBG because the reads were shorter in these datasets and using PEAR ensured that we could use a large *k* for constructing the DBG.

3.1 Benchmarking data

In our experimental evaluation, we used both simulated and real MiSeq sequenced data. We have followed the methodology and datasets used by Baaijens *et al.* (2017), which are described next.

3.1.1 Real data with ground truth

We used a gold standard benchmark for viral assembly (Giallonardo *et al.*, 2014). The reads were produced from 5 HIV strains using Illumina MiSeq (2×250 bp with error around 0.3% and mean insert size 371 bp) with 20 000× coverage. As the five strains contained in the sample are known, it is possible to validate the achieved results. Table 1 includes the main characteristics of this dataset (HIV-real).

3.1.2 Synthetic benchmarks

Five different simulated datasets were used, consisting of 2×250 bp Illumina reads from different virus strains, namely HIV, HCV and ZIKV. The HIV-5, ZIKV-3 and HCV-10 datasets are the datasets generated by Baaijens *et al.* (2017). The read length in these datasets is 2×250 bp and the insert size is 450 bp. The ZIKV-15 dataset was regenerated by us using SimSeq with default configuration for Illumina MiSeq reads (read length 2×250 bp and insert size 500 bp). Table 1 also shows the main characteristics of these datasets.

3.1.3 Divergence ratio and relative abundance benchmarks

We used synthetic datasets for measuring the algorithm bounds. To analyse when the algorithm loses its effectiveness, we used datasets with extreme properties that differ from the real data or the synthetic datasets used in usual experiments, which are generally simulated using realistic properties. Thus, we used 36 datasets from HIV-86.9 strain, varying the divergence ratio (0.5%, 0.75%, 1%, 2.5%, 5% and 10%) and the relative abundance (1:1, 1:2, 1:5, 1:10, 1:50 and 1:100) of each haplotype. These datasets also correspond to the datasets used by Baaijens *et al.* (2017), in an effort to avoid any bias in the experiments.

3.1.4 Real data without ground truth

We have included two real patient samples. More concretely, (i) *ZIKV sample*: an Asian-lineage ZIKV sample consisting of Illumina MiSeq 2×300 bp reads with ∼30 000× coverage sequenced from a rhesus macaque after 4 days of infection (Dudley *et al.*, 2016) and publicly available in NCBI under the accession code SRR3332513 and (ii) *HCV sample*: an HCV sample consisting of Illumina MiSeq reads with ∼80 000× coverage, sequenced from an Australian human patient after 135 days of infection, publicly available in NCBI under the accession code SRR1056035.

3.2 Evaluation scenarios

We ran several experiments under different scenarios. First, we analysed the behaviour of our method when the target genome is known, such that we can evaluate the obtained results. More concretely, we used the evaluator MetaQUAST (Mikheenko *et al.*, 2016) with the option '-unique-mapping', which allows us to assay metagenomic results getting the best unique alignment for each contig to the objective genomes, avoiding one contig to cover more than one genome fragment. We obtained several statistics, such as the largest contig, mismatches/indels/N-Rate, misassemblies, *N*50, genome fraction, etc. Furthermore, we measured the time spent and the memory used during the whole assembly process. As in previous work (Baaijens *et al.*, 2017), we only considered contigs above 500 bp.

To further analyse the performance of our method, we ran some experiments to check the algorithm bounds. We used the synthetic data with different abundance and divergence ratios, and compared the obtained results in terms of percentage of genome retrieved and percentage of mismatches.

Finally, we ran some experiments over those datasets with no available ground truth. This is the case for real virus sample HCV and ZIKV, which may have mixed data of other organisms different from the considered virus, making the discovery even more challenging.

3.3 Results comparison—overall performance

Table 2 shows a summary of the results obtained when applying each assembler over the benchmarking datasets. The complete table, including also the results for the datasets ZIKV-3 and HCV-10, and the additional values of number of contigs larger than 500 bp, length of the largest contig, percentage of indels, *N*-rate and total user CPU time, can be seen in the Supplementary Material.

Overall, the results show, as expected, that tools specifically designed for viral quasispecies inference obtain the best results in genome fraction and largest alignment for all datasets. SAVAGE, PEHaplo and viaDBG show a good performance on the average length of the retrieved contigs. SAVAGE generally retrieves a higher genome fraction and obtains larger contigs than viaDBG and PEHaplo. When the datasets are more complex, namely large differences in genome abundances or high number of strains, PEHaplo fails. For example, we could not get meaningful results for PEHaplo on the ZIKV-15 dataset and thus these are missing in Table 2. After correcting the reads, PEHaplo removes all of those that do not have a large enough number of duplications or substrings. Ideally, when the dataset has high coverage (around 20 000×), every position of the genome will have a significant number of reads starting on it. However, when the number of strains is high, the coverage for each strain is reduced. Furthermore, if abundance for each strain is large, then the impact over the coverage for each strain is even higher. On the other hand, in accordance with the results reported by Baaijens *et al.* (2017), SPAdes gets results comparable with tools specifically designed for viral assembly on some of the simulated datasets, such as HIV-5, but poor performance over the real dataset (HIV-real). Exactly the opposite happens with metaSPAdes, which obtains low genome fractions, large number of mismatches and low *N*50 values for simulated data, whereas it improves the genome fraction retrieved for HIV-real (while keeping high rates of mismatches and misassemblies).

Table 2 also shows that SPAdes' performance decreases when the number of strains increases, the relative abundance decreases, and similarity ratio increases. This is due to the fact that SPAdes does not implement any strategy to deal with this situation. As commented before, PEHaplo also encounters problems when the number of strains increases. In contrast, viaDBG and SAVAGE obtain similar performance, SAVAGE being more sensitive to the strain relative abundance.

On the HIV-real and HIV-5 datasets, PEHaplo achieves the highest *N*50 but the contigs reported have much more errors than viaDBG (four times more mismatches on HIV-5 and 17 times more mismatches on HIV-real than the contigs produced by viaDBG). This indicates that some of the strains have been mixed in the contigs produced by PEHaplo. For the HIV-real dataset, we also report the results of PEHaplo without the polishing step (PEHaplo** in the table) and see that the mismatch rate is much lower without the polishing step but also the *N*50 drops below the *N*50 of viaDBG.

Focusing in the case of the real dataset (HIV-real in Table 2), viaDBG has the overall best performance. Although the genome

Table 1. Main characteristics for the datasets with ground truth available

	Virus type	Genome length (bp)	Average coverage	Num. strains	Abundance (%)	Divergence (%)
HIV-real	HIV-1	9487–9719	20 000×	5	10–30	1–6
HIV-5	HIV-1	9487–9719	20 000×	5	5–28	1–6
ZIKV-3	ZIKV	10 251–10 269	20 000×	3	16–60	3–10
ZIKV-15	ZIKV	10 251–10 269	20 000×	15	1–13	1–12
HCV-10	HCV-1a	9273–9311	20 000×	10	5–19	6–9

Table 2. Assembly results per method on the benchmarking datasets when ground truth is known

Dataset	Method	% genome	*N*50	Misassemblies	% mismatches	Elap time (min)	Memory (GB)
HIV-real	viaDBG*	87.25	1813	0	0.197	4.48	3.74
	viaDBG	89.53	1986	0	0.204	20.01	3.74
	SAVAGE	91.79	611	0	0.684	218.30	49.12
	PEHaplo	87.96	2995	0	3.521	12.74	3.48
	PEHaplo**	91.43	1262	0	0.074	7.56	3.48
	SPAdes	20.15	660	1	2.091	12.74	5.52
	metaSPAdes	83.10	1432	3	9.291	9.06	4.29
HIV-5	viaDBG	97.50	8046	2	0.151	5.01	2.89
	SAVAGE	98.22	6001	3	0.014	204.40	26.11
	PEHaplo	78.59	9328	2	0.690	23.93	4.86
	SPAdes	90.91	5097	2	0.051	3.31	4.12
	metaSPAdes	35.87	6385	6	5.322	3.86	2.99
ZIKV-15	viaDBG	86.06	1759	0	0.002	18.26	3.71
	SAVAGE	82.72	1632	0	0.002	352.98	9.03
	PEHaplo	—	—	—	—	—	—
	SPAdes	38.97	2063	0	0.147	6.17	4.42
	metaSPAdes	16.03	3863	0	2.273	4.49	3.19

Note: viaDGB, * omits the correction step and PEHaplo, ** omits the polishing step.

fraction retrieved is slightly lower than SAVAGE (2.26% points), viaDBG is able to get the largest contig, a longer average contig and a lower number of mismatches. Moreover, using the standard pipeline of PEHaplo, viaDBG obtains a larger genome fraction retrieved and a lower number of mismatches and indels. The high number of mismatches and indels obtained by the standard PEHaplo pipeline indicates that some of the haplotypes have been mixed. With this dataset, PEHaplo obtains higher genome fraction and lower number of mismatches and indels, but also a lower *N*50 and shorter largest contig, if the polishing step is omitted (indicated as PEHaplo** in the table).

We will closely compare the behaviour of each method by taking into account the results for ZIKV-15 datasets, which can be considered the most challenging simulated dataset. We omit PEHaplo from the comparison, as we were not able to run this tool and produce reliable results for this dataset. Figure 4 shows the percentage of genome recovered for each strain of the ZIKV-15 dataset. Table 2 shows that viaDBG and SAVAGE have a similar overall performance. However, a deeper comparison reveals that the behaviour of each method is rather different. SAVAGE retrieves the highest percentage of genome for most of the strains. This can be caused by the fact that viaDBG systematically removes the beginning and the end of the genomes due to lack of coverage in these regions. Despite SAVAGE outperforming viaDBG in most cases, SAVAGE fails at assembling the genome for two strains. This is not happening with viaDBG, which retrieves a significant portion of the genome in all cases, being close to SAVAGE in most cases, and even sometimes outperforming its results. More concretely, one of the genomes lost by SAVAGE, HQ234501.1 (Mutant 1%) (Abundance 6%), is almost fully recovered by viaDBG. The performance of metaSPAdes and SPAdes in this particular example was quite bad: they could only recover one of the 15 strains completely. Moreover, metaSPAdes did not retrieve any portion from 11 of them.

3.4 Efficiency analysis

We measured the runtime and peak memory usage required by all the algorithms when applied to each dataset. All algorithms were given access to 32 cores in all experiments. Error correction, adding paired-end information and polishing the paired-end information have been parallelized in viaDBG. The Supplementary Material includes an experiment evaluation of the effects of the number of cores used by each of the tools.

As shown in Table 2, SAVAGE needs much more time than the rest of the methods, ranging from 204.40 min in the fastest case to 352.98 min in the slowest one. This is caused by the computations made by SAVAGE during the overlap graph construction, which requires the enumeration of all approximate suffix–prefix overlaps among the reads. PEHaplo, despite following also an overlap graph approach, obtains much better execution times, as it removes a high percentage of repeated reads, thus, alleviating the construction of the graph. On the other hand, the time performance of the methods based on the DBG is better: SPAdes and metaSPAdes were around 1.5 times faster than viaDBG. However, when the correction step is omitted, viaDBG outperforms both of them on the real dataset, HIV-real. Results show that viaDBG worsens its time efficiency when including the error correction step, as the CPU times are around 4.5 times higher. This is an expected result, since the time complexity of the error correction performed by viaDBG is $O(n * m)$ where n is the number of reads and m is the maximum length of the reads. PEHaplo is faster than viaDGB with correction step for HIV-real dataset, but viaDBG obtains more accurate results. In the Supplementary Material, we can also see that PEHaplo obtains slightly better time efficiency and also better accuracy than viaDBG for HCV-10.

We also measured the peak memory required by each tool. Among the de Bruijn methods, SPAdes and metaSPAdes require the highest memory resources, reaching 5.52 GB, whereas viaDBG requires at most 3.74 GB per execution. Only for ZIKV-15 dataset, metaSPADES obtains lower memory consumption, but also much lower accuracy. On the other hand, if we consider the overlap methods, the memory requirements of SAVAGE are much higher, from 9.03 to 49.12 GB, depending on the file size, whereas PEHaplo requires for their worst tested case, HIV-5, 4.86 GB (8.99 GB if we consider HCV-10, as seen in the Supplementary Material).

3.5 Testing viaDBG limits

In this section, we will explore the algorithm bounds. We will follow the methodology used in the experimental evaluation of Baaijens *et al.* (2017), using 36 simulated datasets that vary their abundance and divergence.

Figure 5 shows the results obtained by viaDGB, SAVAGE and PEHaplo for each of these datasets in terms of percentage of retrieved genome and percentage of mismatches. In this experiment, in the case of SAVAGE, PEAR was applied over the input datasets. As we can see, viaDBG has a surprising behaviour with 10%, 5% and 2.5%, as it is able to retrieve almost the complete genome until the 1:50 abundance relation. Furthermore, on 1:50 relation, it is able to retrieve around 60% of the genome for the minor strain.

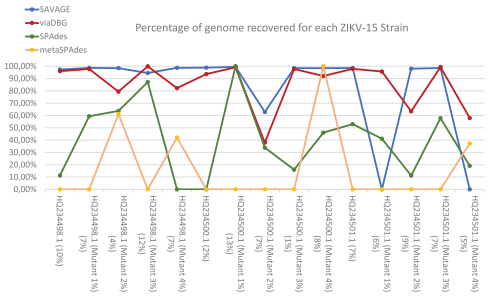
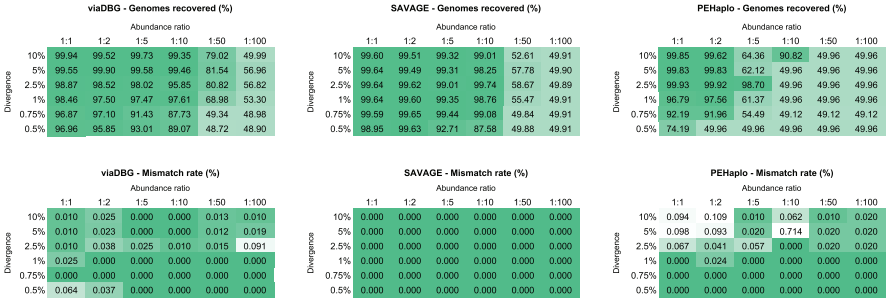


Fig. 4. Comparison between the four tools and the ZIKV-15 strains dataset



viaDBG - Genomes recovered (%)

Divergence \ Abundance ratio	1:1	1:2	1:5	1:10	1:50	1:100
10%	99.94	99.52	99.73	99.35	79.02	49.99
5%	99.55	99.90	99.58	99.46	81.54	56.96
2.5%	98.87	98.52	98.02	95.85	80.82	56.82
1%	98.46	97.50	97.47	97.61	68.98	53.30
0.75%	96.87	97.10	91.43	87.73	49.34	48.96
0.5%	96.96	95.85	93.01	89.07	48.72	48.90

SAVAGE - Genomes recovered (%)

Divergence \ Abundance ratio	1:1	1:2	1:5	1:10	1:50	1:100
10%	99.60	99.51	99.32	99.01	52.61	49.91
5%	99.64	99.49	99.31	98.25	57.78	49.90
2.5%	99.64	99.62	99.01	99.74	58.67	49.89
1%	99.64	99.60	99.35	98.76	55.47	49.91
0.75%	99.59	99.65	99.44	99.08	49.84	49.91
0.5%	98.95	99.63	92.71	87.58	49.88	49.91

PEHaplo - Genomes recovered (%)

Divergence \ Abundance ratio	1:1	1:2	1:5	1:10	1:50	1:100
10%	99.85	99.62	64.36	90.82	49.96	49.96
5%	99.83	99.83	62.12	49.96	49.96	49.96
2.5%	99.93	99.92	98.70	49.96	49.96	49.96
1%	96.79	97.56	61.37	49.96	49.96	49.96
0.75%	92.19	91.96	54.49	49.12	49.12	49.12
0.5%	74.19	49.96	49.96	49.96	49.96	49.96

viaDBG - Mismatch rate (%)

Divergence \ Abundance ratio	1:1	1:2	1:5	1:10	1:50	1:100
10%	0.010	0.025	0.000	0.000	0.013	0.010
5%	0.010	0.023	0.000	0.000	0.012	0.019
2.5%	0.010	0.038	0.025	0.010	0.015	0.091
1%	0.025	0.000	0.000	0.000	0.000	0.000
0.75%	0.000	0.000	0.000	0.000	0.000	0.000
0.5%	0.064	0.037	0.000	0.000	0.000	0.000

SAVAGE - Mismatch rate (%)

Divergence \ Abundance ratio	1:1	1:2	1:5	1:10	1:50	1:100
10%	0.000	0.000	0.000	0.000	0.000	0.000
5%	0.000	0.000	0.000	0.000	0.000	0.000
2.5%	0.000	0.000	0.000	0.000	0.000	0.000
1%	0.000	0.000	0.000	0.000	0.000	0.000
0.75%	0.000	0.000	0.000	0.000	0.000	0.000
0.5%	0.000	0.000	0.000	0.000	0.000	0.000

PEHaplo - Mismatch rate (%)

Divergence \ Abundance ratio	1:1	1:2	1:5	1:10	1:50	1:100
10%	0.094	0.109	0.010	0.062	0.010	0.020
5%	0.098	0.093	0.020	0.714	0.020	0.020
2.5%	0.067	0.041	0.057	0.000	0.020	0.020
1%	0.000	0.024	0.000	0.000	0.000	0.000
0.75%	0.000	0.000	0.000	0.000	0.000	0.000
0.5%	0.000	0.000	0.000	0.000	0.000	0.000

Fig. 5. Performance of viaDBG, SAVAGE and PEHaplo for different divergence and abundance ratios

Comparing our results with those achieved by SAVAGE and PEHaplo, we can see that viaDBG behaves better than either of them in these scenarios with higher differences of abundance rates. For example, in the 1:50 case, neither SAVAGE nor PEHaplo is able to retrieve more than 10–20% of the minor strain, and in most cases they retrieve 0% of the minor strain. On the divergence bound behaviour, viaDBG's results decrease in comparison to SAVAGE, retrieving a bit less genome fraction when divergence is below 1%. A possible reason for this is the length of the processed *k*-mers. SAVAGE uses the full-length reads (>200 bp) and extends them, which produces much longer reads, thus more accuracy. On the other hand, viaDBG uses fixed *k*-mer length, which produces a slight loss in accuracy when divergence is below 1%. Nevertheless, it seems that both SAVAGE and viaDBG have a more robust behaviour than PEHaplo, which suffers when divergence is below 0.75%. In conclusion, viaDBG can properly handle different ranges of divergence levels and of relative abundances, especially for extreme differences in the abundance ratio.

3.6 Real datasets with unknown target genome

In this section, we show the results obtained for a real virus sample from patients infected by the Asian-lineage ZIKV. To evaluate the results, we use as references the complete genome sequence of the Asian-lineage ZIKV (KU681081.3). The result for the HCV can be found in the Supplementary Material.

We obtain 10 contigs above 1000 bp covering 9578 out of 10 677 bases, with a *N*50 of 1975 bp and a largest contig of 2445 bp. Additionally, 17 809 bases were aligned, thus, it is obvious that more than one strain is contained in the sample. According to these results, it seems that there are two different, but highly similar strains, in the sample. Besides, in our analysis, we have not discovered any local misassembly, which means that there is no contig that does not align with the reference at some point.

4 Conclusion

We present viaDBG, a time- and memory-efficient *de novo* multi-assembler for viral quasispecies. Viral samples generally contain several haplotypes, which have evolved from the same genome through multiple mutations and recombination events. Additionally, not all viral genomes within the sample have exactly the same frequency, namely each viral genome has its own level of abundance. Experimental results have shown that general purpose and metagenomic assemblers, such as SPAdes and metaSPAdes, are not able to retrieve the viral genomes in the sample. This motivates the research on new specific tools that can overcome all these limitations.

Our experimental evaluation shows that viaDBG is able to get competitive, sometimes even better, results in comparison to state-of-the-art *de novo* viral quasispecies assemblers, such as SAVAGE and PEHaplo. Furthermore, the runtime of viaDBG is much lower than SAVAGE and also to PEHaplo in most cases, and its memory usage is also lower than its counterparts, making viaDBG an attractive alternative. One of the main drawbacks of PEHaplo is that its behaviour is highly dependent on the parameter configuration, which is not easy to determine for each particular dataset. Furthermore, in some extremely complex cases, such as 15 ZIKV strains where genomes are extremely close and abundance is extremely low, viaDBG is able to retrieve information for the whole set of strains and the overall genome fraction is higher than for other tools. Despite of these successful results, our method shows a weaker behaviour than SAVAGE for datasets with extreme divergence ratios.

The main reasons for the good performance of viaDBG are the error correction step and the systematic use of the paired-end information. On the one hand, error correction enables the usage of extremely large *k*-mers (120-mers), as the veracity of the *k*-mers is improved due to the adjustment of their frequency distribution. Additionally, a side effect of the correction is that it reduces the possibility of having wrong pairs for genomic *k*-mers and vice versa.

On the other hand, the cliques retrieved by using the paired-end information for every pair of *k*-mers allow us to change the original DBG into a much less tangled graph. Applying paired information as a post-processing step is more restrictive than adding the information directly in the graph. When used during the post-processing step, only reads that align entirely with one contig are going to be used, whereas all reads with genomic information can improve the results when they are considered during the graph construction. Therefore, there is always more information when using paired *k*-mers than when using the overlap between reads and contigs.

The main advantage of viaDBG is its efficiency, both in terms of execution time and memory usage. Our method benefits from the better efficiency of DBG approaches, which avoids computing overlaps between all reads. Despite the computations of the paired information, viaDBG has proven to be much faster than overlap based methods.

As a future work, we plan to reduce the memory footprint of viaDBG by taking full advantage of compacted DBGs. This will also allow us to study the suitability of our approach for metagenomics assembling, which is a more demanding task. Another line of improvement is to enhance the current parallelization of viaDBG by taking into account some relevant issues, such us disk accesses, thread synchronization and data interchanges. In parallel, we will consider the possibility of integrating our approach with Virus-VG (Baaijens *et al.*, 2019) to produce larger contigs.

Acknowledgements

We want to thank J. Baaijens and J. Chen for the help in the execution and in the search for the right parameters on SAVAGE and PEHaplo, respectively.

Funding

This research has received funding from the European Union's Horizon 2020 research and innovation programme under the Marie Sklodowska-Curie [grant agreement number 690941]; Ministerio de Ciencia, Innovación y Universidades [TIN2016-78011-C4-1-R, TIN2016-77158-C4-3-R and FPU17/02742]; Xunta de Galicia [ED431C 2017/58, ED431G/01, IN848D-2017-2350417 and IN852A 2018/14]; and from Academy of Finland [308030, 314170 and 323233]. We also wish to acknowledge the support received from the Centro de Investigación de Galicia "CITIC", funded by Xunta de Galicia and the European Union (European Regional Development Fund- Galicia 2014-2020 Program), by grant ED431G 2019/01.

Conflict of Interest: none declared.

Data availability

The data underlying this article are available in Bitbucket Repository, at https://bitbucket.org/bfreirec1/data-viadbg/.

References

Ahn,S. and Vikalo,H. (2018) aBayesQR: a Byesian method for reconstruction of viral populations characterized by low diversity. *J. Comput. Biol.*, **25**, 637–648.

Baaijens,J.A. *et al.* (2017) De novo assembly of viral quasispecies using overlap graphs. *Genome Res.*, **27**, 835–848.

Baaijens,J.A. *et al.* (2019) Full-length de novo viral quasispecies assembly through variation graph construction. *Bioinformatics*, **35**, 5086–5094.

Bankevich,A. *et al.* (2012) SPAdes: a new genome assembly algorithm and its applications to single-cell sequencing. *J. Comput. Biol.*, **19**, 455–477.

Barik,S. *et al.* (2018) QSdpR: viral quasispecies reconstruction via correlation clustering. *Genomics*, **110**, 375–381.

Chen,J. *et al.* (2018) *De novo* haplotype reconstruction in viral quasispecies using paired-end read guided path finding. *Bioinformatics*, **34**, 2927–2935.

Domingo,E. *et al.* (2012) Viral quasispecies evolution. *Microbiol. Mol. Biol. Rev.*, **76**, 159–216.

Dudley,D.M. *et al.* (2016) A rhesus macaque model of Asian-lineage Zika virus infection. *Nat. Commun.*, **7**, 12204.

Duffy,S. *et al.* (2008) Rates of evolutionary change in viruses: patterns and determinants. *Nat. Rev. Genet.*, **9**, 267–276.

Giallonardo,F.D. *et al.* (2014) Full-length haplotype reconstruction to infer the structure of heterogeneous virus populations. *Nucleic Acids Res.*, **42**, e115.

Holmes,E.C. (2009) *The Evolution and Emergence of RNA Viruses*. Oxford University Press, Oxford.

Jayasundara,D. *et al.* (2015) ViQuaS: an improved reconstruction pipeline for viral quasispecies spectra generated by next-generation sequencing. *Bioinformatics*, **31**, 886–896.

Knyazev,S. *et al.* (2019) CliqueSNV: scalable reconstruction of intra-host viral populations from NGS reads. *bioRxiv*. doi: 10.1101/264242.

Malhotra,R. *et al.* (2015) Maximum likelihood de novo reconstruction of viral populations using paired end sequencing data. *arXiv e-Prints*.

Martin,M. (2011) Cutadapt removes adapter sequences from high-throughput sequencing reads. *EMBnet J.*, **17**, 10–12.

Medvedev,P. *et al.* (2011) Paired de Bruijn graphs: a novel approach for incorporating mate pair information into genome assemblers. *J. Comput. Biol.*, **18**, 1625–1634.

Mikheenko,A. *et al.* (2016) MetaQUAST: evaluation of metagenome assemblies. *Bioinformatics*, **32**, 1088–1090.

Nagarajan,N. and Pop,M. (2013) Sequence assembly demystified. *Nat. Rev. Genet.*, **14**, 157–167.

Nurk,S. *et al.* (2017) metaSPAdes: a new versatile metagenomic assembler. *Genome Res.*, **27**, 824–834.

Posada-Cespedes,S. *et al.* (2017) Recent advances in inferring viral diversity from high-throughput sequencing data. *Virus Res.*, **239**, 17–32.

Prabhakaran,S. *et al.* (2014) HIV haplotype inference using a propagating Dirichlet process mixture model. *IEEE/ACM Trans. Comput. Biol. Bioinform.*, **11**, 182–191.

Prosperi,M.C.F. and Salemi,M. (2012) QuRe: software for viral quasispecies reconstruction from next-generation sequencing data. *Bioinformatics*, **28**, 132–133.

Salmela,L. and Rivals,E. (2014) LoRDEC: accurate and efficient long read error correction. *Bioinformatics*, **30**, 3506–3514.

Töpfer,A. *et al.* (2013) Probabilistic inference of viral quasispecies subject to recombination. *J. Comput. Biol.*, **20**, 113–123.

Töpfer,A. *et al.* (2014) Viral quasispecies assembly via maximal clique enumeration. *PLoS Comput. Biol.*, **10**, e1003515.

Zagordi,O. *et al.* (2011) ShoRAH: estimating the genetic diversity of a mixed sample from next-generation sequencing data. *BMC Bioinformatics*, **12**, 119.

Zhang,J. *et al.* (2014) PEAR: a fast and accurate Illumina Paired-End reAd mergeR. *Bioinformatics*, **30**, 614–620.

"output" — 2020/3/11 — 9:08 — page 1 — #1

Supplementary Material

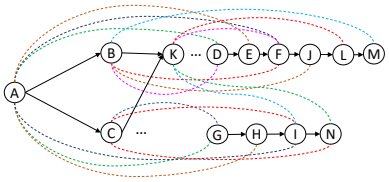


Fig. 1. DBG with the paired-end information.

Contents

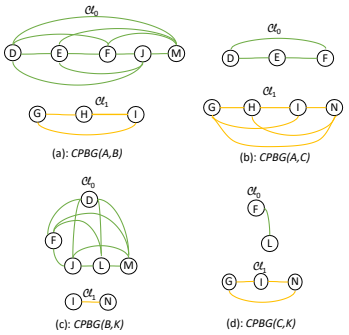


Fig. 2. The four different clique graphs.

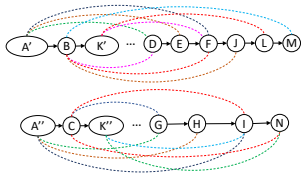


Fig. 3. The final form of the graph.

1 Algorithm for obtaining the haplotypes

Algorithm 1 shows the pseudocode of the algorithm for obtaining the haplotypes, described in Section 2.4.4 of the main paper, that is, steps 2(e), 2(f), and 2(g) of the Figure 1 of the main paper.

Figure 1 shows a portion of a DBG used to illustrate this process. Solid black arrows are the edges of DBG, whereas the coloured dotted lines show the pair-end information of each node.

The algorithm receives as input the DBG obtained through the steps 2(a) until 2(d) of the Figure 1 of the main paper. In Line 4, for each pair of adjacent nodes (A, B) of the DBG, a CPBG (*CPBG(A,B)*) is built. Lines 6 and 7 create the nodes of *CPBG(A,B)*. Lines 8–10 add the edges of the CPBG.

Figure 2(a) shows *CPBG(A,B)*.[1] It is composed of the nodes in *P(A)={D, E, F, G, H, I}* and *P(B)={M, D, F, J}*. There is an edge between two nodes if there is a path in the DBG of length smaller than 2Δ. In this case, function $reach_{DBG}(A, B)$ returns *true*.

Line 13 obtains the set of maximal cliques in the CPBG (denoted as $\mathcal{SC}$). This algorithm is shown in Section 2 of this Supplementary Material.

Figure 2 shows the four different CPBGs obtained for the DBG shown in Figure 1. Each clique in the graphs is highlighted by its own color.

Line 14 polishes the cliques in $\mathcal{SC}$. For example, in *CPBG(A,B)*, the yellow clique will be discarded because it only has nodes from *P(A)*. This step avoids having short tips (1 bp) and having several contigs which lead to exactly the same strain.

Lines 15–28 create the nodes of the new DBG. Line 15 is a loop that processes all cliques of the treated pair. If the processed clique has nodes of $P(A)$ and $P(B)$, then that pair is added to the new DBG, otherwise it is discarded (Line 16).

In Figure 2(a), only $\mathcal{C}\ell_0$ passes the polish step. Therefore the pair (A,B) is added to the output (see Figure 3): A with paired information $P(A) \cap \mathcal{C}\ell_0$ *={D, E, F}* and B with paired information $P(B) \cap \mathcal{C}\ell_0 =$ *{D, F, J}*.

Processing $\mathcal{C}\ell_1$ of *CPBG(A,C)* (see Figure 2(b)) produces the output of A and C. However, that A is *different*[2] from that obtained from *CPBG(A,B)*, and thus, to differentiate them, we use A' for the one obtained from *CPBG(A,B)* and A'' for the one obtained from *CPBG(A,C)*. Continuing the process, we obtain the new DBG of Figure 3.

2 Obtaining the maximal cliques in the CPBG

Classic algorithms to find cliques in undirected graphs, such as those of Johnson *et al.* (1988) or Tomita *et al.* (2006), have a considerable computational cost, for example, $O(3^{n/3})$ in the case of Tomita *et al.*

[1] We do not use representative k-mers to facilitate the understanding of the example.

[2] In the sense that they belong to different strains.

Algorithm 1 Obtain Haplotypes (DBG)

1: **let** new_DBG be a DBG
2: **let** $new\_DBG_{nodes} = \emptyset$
3: **let** $new\_DBG_{edges} = \emptyset$
4: **for all** (A, B) a pair of adjacent nodes of the DBG **do** {Builds the CPBG}
5: **let** $CPBG(A, B)_{edges} = \emptyset$
6: **let** $CPBG(A, B)_{nodes} = P(A)$ {Adds the representative k-mers in P(A)}
7: **let** $CPBG(A, B)_{nodes} = CPBG(A, B)_{nodes} \cup P(B)$ {Adds the representative k-mers in P(B)}
8: **for all** (U, V) nodes in $CPBG(A, B)_{nodes}$ **do**
9: **if** $reach_{DBG}(U, V)$ **then**
10: $CPBG(A, B)_{edges} = CPBG(A, B)_{edges} \cup \{(U, V)\}$
11: **end if**
12: **end for**
13: **let** $\mathcal{SC} = \{\mathcal{C}\ell_0, \mathcal{C}\ell_1, ..., \mathcal{C}\ell_n\}$ the maximal cliques in $CPBG(A, B)$
14: **polish** ($\mathcal{SC}$)
15: **for all** $\mathcal{C}\ell_i \in \mathcal{SC}$ **do**
16: **if** $\mathcal{C}\ell_i \cap P(A) \neq \emptyset$ **and** $\mathcal{C}\ell_i \cap P(B) \neq \emptyset$ **then**
17: **let** $new\_DBG_{nodes} = new\_DBG_{nodes} \cup \{A_{P(A)\cap\mathcal{C}\ell_i}\}$ {Adds version A with paired information $P(A) \cap \mathcal{C}\ell_i$}
18: **let** $new\_DBG_{nodes} = new\_DBG_{nodes} \cup \{B_{P(B)\cap\mathcal{C}\ell_i}\}$ {Adds version B with paired information $P(B) \cap \mathcal{C}\ell_i$}
19: **let** $new\_DBG_{edges} = new\_DBG_{edges} \cup \{(A_{P(A)\cap\mathcal{C}\ell_i}, B_{P(B)\cap\mathcal{C}\ell_i})\}$ {Both nodes are connected}
20: **end if**
21: **end for**
22: **end for**
23: **search** unitigs in new_DBG

Algorithm 2 Obtain maximal Cliques ($DBG, CPBG(A, B)$)

1: **for all** Node n in $CPBG(A, B)$ **do**
2: **for all** Edge e of DBG that reaches n **do**
3: **let** $n.weight$ =+ $e.weight$
4: **end for**
5: **end for**
6: **for all** Node n in $CPBG(A, B)$ **do**
7: **for all** Node n' in $CPBG(A, B)$ adjacent to n **do**
8: **let** $n.weight$ =+ $n'.weight$
9: **end for**
10: **end for**
11: **let** $i = 0$
12: **let** $\mathcal{C}\ell_i = \emptyset$
13: **repeat**
14: **let**=n be the node of $CPBG(A, B)$ with the highest value of $degree \times n.weight$
15: **repeat**
16: **let** $\mathcal{C}\ell_i = \mathcal{C}\ell_i \cup n$
17: **let** $n_{used} = true$
18: **let** $n' = n$
19: **let**=n be the node of $CPBG(A, B)$ adjacent to n' and also connected to all nodes in c_i with the highest value $degree \times n.weight$ and not in c_i
20: **until** $n = \emptyset$
21: **let** $lowest$ be the lowest weight of all nodes in c_i
22: **for all** Node n in $\mathcal{C}\ell_i$ **do**
23: **let** $n.weight$ =- $lowest$
24: **if** $n.weight$==0 **then**
25: **let** $n.weight = 1$
26: **end if**
27: **end for**
28: **let** i=i+1
29: **let** $\mathcal{C}\ell_i = \emptyset$
30: **until** All nodes in $CPBG(A, B)$ are used
31: **return** $\mathcal{C}\ell_0, \mathcal{C}\ell_1, \ldots, \mathcal{C}\ell_v$

(2006). Therefore, instead, we use a faster heuristic method shown in Algorithm 2, which is based on the work by Pattabiraman *et al.* (2015).

It receives as input the DBG and the CPBG of a given pair of nodes. Observe in Figure 4, that the edges of the DBG corresponding to paired information are now labeled with the number of paired reads that contain the corresponding pair of k-mers. That frequency is used to determine which cliques correspond to real strains.

The first loop of Lines 1–4 enriches the nodes of the CPBG with a number which results from adding the weight of all edges which reach that node in the DBG. Figure 5(a) shows the result of this process for the *CPBG(A,B)* of our example in Figure 4. For example, observe the node D, its weight (18) is obtained by adding the weight of the edge of the DBG connecting A and D, with weight 10, and that connecting B and D, with weight 8.

"output" — 2020/3/11 — 9:08 — page 3 — #3

The next loop in Lines 6–10 adds more weight to each node of the CPBG, specifically, the weight of all its adjacent nodes. Figure 5(b) shows the result. The weight of D is 45, resulting from adding its weight (18), and those of its adjacent nodes (J (8), F (9), and E (10)).

The loop of Lines 13–29 is the main part of the algorithm. Line 14 selects the node with the highest value resulting from multiplying its degree (number of adjacent nodes) and its weight. In our example, that node is F.

Then, the loop of Lines 15–20 builds the clique containing that node. In our example, first F is added to the clique and marked as "used". Then, line 19 obtains the adjacent node of F with the highest value resulting from multiplying its degree and its weight. In our example, all nodes are adjacent to F, but E and J are those with the highest value ($46 * 4$). So, the process continues adding, let say, E and then J and marking them as "used". From J, the adjacent node with highest value is D, which is also added to the clique. After processing D, all its adjacent nodes are already in the clique, and thus the process ends.

Next, Lines 21–27 decrease the weight of all the nodes of the recently created clique by subtracting the weight of the node of the clique with the lowest value, except if the result of that subtraction is 0, which is changed to 1. The resulting clique of this process is shown in Figure 5(c) highlighted in yellow. Observe that its nodes remain now with a low weight, then it is unlikely that these nodes will be part of subsequent cliques. The idea is that we have concluded that those nodes correspond to a given strain, it is unlikely that they would be part of another different strain.

The process continues until all nodes have been marked as "used". In our example, another clique is created, as shown in Figure 5(d), but this clique is later removed by the polishing process, as explained in the main manuscript.

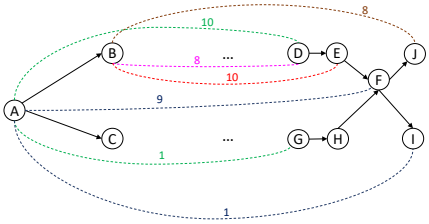


Fig. 4. DBG with the frequency of appearance of paired $k-$mers.

3 Parameter selection

The most important parameters of our method are *k-mer solid threshold* and the variance of the insert size which we denote by Δ. The Δ value can be configured using a higher value than needed without endangering the assembly results. Nevertheless, if some further information is known, then using a more precise Δ will provide better results avoiding some false cliques that may appear. On the other hand, the *k-mer solid threshold* is automatically selected by using the histogram of the k-mers frequencies, where a change in the trend of the frequency count intuitively means that from there on k-mers having higher frequencies are genomic. For better identifying this point of change, we use a window of size N.

We have conducted several experiments for analysing the effect of this window size in the results. We obtained that the threshold selection for the simulated data is quite stable, whereas it is much more variable for the real data set HIV-real. A possible explanation is given by Figure 6, which shows

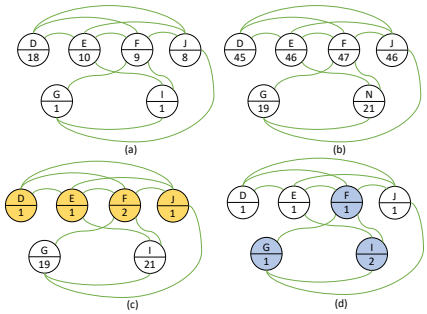


Fig. 5. Obtaining the cliques of the *CPBG(A,B)* of the DBG of Figure 4.

two k-mer frequency histograms: one for the real data set HIV-real and the other for the simulated data set HIV-5. Nevertheless, results in terms of genome fraction retrieved have been consistent throughout the whole evaluation. This information is provided in Figures 7 and 8. In Figure 7, we can see that for HIV-real, the threshold suggested by our algorithm increases when the windows size grows, whereas for the simulated data sets, the threshold is practically stable. On the other hand, we can see in Figure 8 that the windows size does not affect the percentage of the retrieved genome for any data set.

4 Description of Zika virus simulated data sets (ZIKV-3 and ZIKV-15)

SAVAGE was assayed by using a 15-strain ZIKV data set. Unfortunately, ground truth criteria, namely reference, was not available. Therefore, we simulated our own references and the data set.

Twelve extra references were produced from only 3 strains, all of African lineage: one from Uganda (accession HQ234498), one from Nigeria (accession HQ234500), and one from Senegal (accession HQ234501). For each reference 4 extra sequences were build by inserting 1%, 1%, 2% and 2% mutations to the reference.

The data set was simulated by using the whole set of 15 references with abundances from 1% to 13% at most.

5 Complete results comparison - overall performance

We include here the complete results of the benchmarking performed in Section 3 of the main paper. Thus, Table 1 contains the results for all the data sets, included ZIKV-3 and HCV-10, which were omitted in Table 2 of the main paper. We also include the values for the number of contigs larger than 500 bp, the length of the largest contig, the percentage of indels, N-rate, and total user CPU time. We measured peak memory usage using gnu time commnad (with option -v), and measured time performance using perf profiler (command perf stat -d), which reports both total CPU user time and elapsed time.

5.1 Analysis of the parallelisation

We conducted an experiment varying the number of cores available to the methods to see how the number of cores affects the running times and the

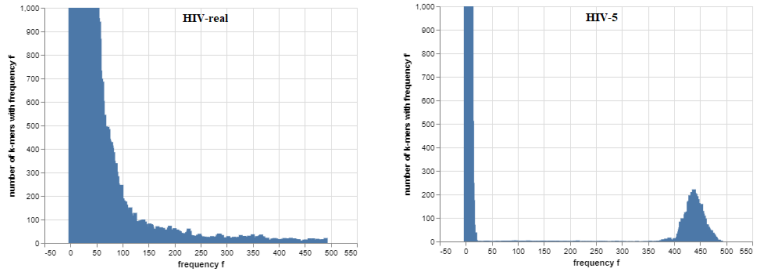


Fig. 6. HIV-real vs HIV-5 frequency histograms.

Table 1. Assembly results per method on the benchmarking data sets when ground truth is known.

data set	method	contigs >500	% genome	N50	largest contig	mis-assemb.	% mis-matches	% indels	% N-Rate	elapsed time (min)	CPU user time (min)	peak mem (GB)
HIV-real	viaDBG - without correction	88	87.25%	1813	8596	0	0.197	0.285	0.000	4.48	17.24	3.74
	viaDBG - with correction	57	89.53%	1986	8966	0	0.204	0.240	0.000	20.01	389.13	3.74
	SAVAGE	459	91.79%	611	2511	0	0.684	0.149	0.104	218.30	4803.06	49.12
	PEHaplo - with polishing	35	87.96%	2995	8674	0	3.521	0.245	0.000	12.74	101.89	3.74
	PEHaplo - without polishing	31	91.43%	1262	6383	0	0.074	0.083	0.000	7.56	50.47	3.74
	SPAdes	1	20.15%	660	2952	1	2.091	0.089	0.000	12.74	99.63	5.52
	metaSPAdes	16	83.10%	1432	2986	3	9.291	0.405	0.000	9.06	111.43	4.29
HIV-5	viaDBG	45	97.50%	8046	9667	2	0.151	0.008	0.000	5.01	63.56	2.89
	SAVAGE	18	97.69%	3305	9645	4	0.120	0.004	0.000	204.40	3618.10	26.11
	PEHaplo	7	78.59%	9328	9656	2	0.690	0.037	0.000	23.93	68.58	4.86
	SPAdes	22	90.91%	5097	9557	2	0.051	0.002	0.000	3.31	25.89	4.12
	metaSPAdes	8	35.87%	6385	6561	6	5.322	0.104	0.000	3.86	51.52	2.99
ZIKV-3	viaDBG	10	99.76%	10203	10267	0	0.000	0.000	0.000	7.56	62.10	3.66
	SAVAGE	3	99.77%	10243	10258	0	0.003	0.000	0.003	332.15	6527.90	42.37
	PEHaplo	2	99.89%	10247	10269	0	0.000	0.000	0.000	20.11	68.19	4.40
	SPAdes	3	99.56%	9851	10269	0	0.000	0.000	0.000	4.05	33.05	4.59
	metaSPAdes	6	33.34%	2890	8675	0	1.919	0.009	0.000	4.96	58.69	3.33
ZIKV-15	viaDBG	185	86.06%	1759	9483	0	0.002	0.000	0.000	18.26	82.22	3.71
	SAVAGE	231	82.72%	1632	10199	0	0.002	0.000	0.002	352.98	8329.14	9.03
	PEHaplo	-	-	-	-	-	-	-	-	-	-	-
	SPAdes	247	38.97%	2063	10251	0	0.147	0.000	0.000	6.17	35.34	4.42
	metaSPAdes	11	16.03%	3863	5261	0	2.273	0.264	0.000	4.49	57.98	3.19
HCV-10	viaDBG	27	97.72%	8934	9293	0	0.005	0.000	0.000	13.03	69.06	2.81
	SAVAGE	20	99.33%	9204	9290	0	0.0975	0.000	1.043	50.01	451.63	26.13
	PEHaplo	10	99.66%	9297	9311	0	0.032	0.000	0.000	29.01	64.79	8.99
	SPAdes	26	90.59%	8690	9311	0	0.002	0.000	0.000	4.10	26.79	4.09
	metaSPAdes	42	49.37%	2742	3475	0	4.534	0.000	0.000	3.73	49.86	2.97

memory usage. We used HIV-real data set with the same configuration for each tool as the previous experiments. Table 2 shows that only SAVAGE approaches an ideal speedup, as times are practically divided by 2 when doubling the number of cores. In the case of viaDBG, using 16 cores instead of 8 only implies an improvement of 2%, whereas the usage of 32 cores yields a speedup of 1.78. Still, that value is around the same as that obtained by metaSPAdes and better than the rest, except for SAVAGE.

In Table 3, we can observe that memory usage grows for all methods when increasing the number of cores. Again, viaDBG obtains similar results using 8 and 16 cores, but the memory usage growth is worse for 32 cores. This increase in memory consumption is moderate in the case of PEHaplo, which is the tool obtaining the best result for 32 cores, but larger for metaSPAdes and SAVAGE.

From the results, we can see that SAVAGE is the tool that best takes advantage of parallelisation, as its running time considerably decreases with the number of cores, at the expense of greater memory requirements. In any case, those time results are still much higher than those obtained by the rest of the techniques. viaDBG parallelisation obtains a slight speedup at the expense of a moderate growth in memory consumption.

"output" — 2020/3/11 — 9:08 — page 5 — #5

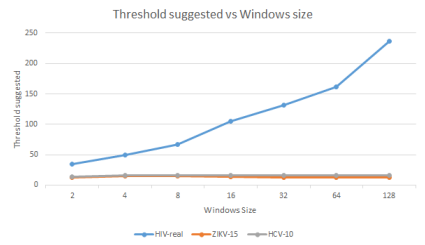


Fig. 7. Threshold value suggested by our approach, varying the windows size.

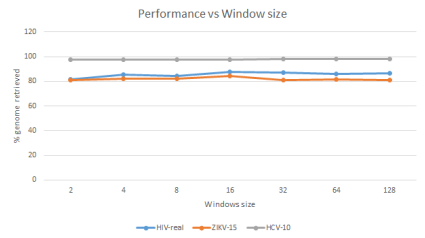


Fig. 8. Genome fraction retrieved, varying the windows size.

Table 2. Running times in minutes, varying the number of cores.

# of cores	viaDBG	SPAdes	metaSPAdes	PeHaplo	SAVAGE
8	5.96	15.51	15.73	11.00	1255.91
16	5.82	12.83	9.72	11.18	468.58
32	3.34	12.74	9.06	7.56	218.30

Table 3. Memory usage in Gigabytes, varying the number of cores.

# of cores	viaDBG	SPAdes	metaSPAdes	PeHaplo	SAVAGE
8	2.85	4.97	6.53	4.12	7.26
16	2.96	7.28	11.57	4.00	10.25
32	6.69	11.35	19.03	5.50	20.27

6 Assembly of real data sets with unknown target genome

6.1 BAC clones in the Zika virus sample

Baaijens *et al.* (2017) discovered human BAC clones within the same real Zika virus data set that we have also analysed. In our results we did not find these. This could be caused by two possible explanations. On the one hand, it is probable that BAC information was not complete, creating several isolated connected components that were removed during the polishing step of the graph. On the other hand, results can differ due to the preprocessing step, which is not exactly the same as the one used by SAVAGE. In fact, results obtained by SAVAGE method with our preprocessing step did not show BAC clones either.

6.2 Hepatitis C Human Sample

Here, we show the results of the Australian human patient infected with Hepatitis C virus. For ground truth, we will use the complete genome of Hepatitis C virus (NC_004102.1).

Obviously, the results are much more non-specific than in the Zika sample. This is probably because of the Australian warm environment, the time that the virus spent in the carrier, plus the initial number of strains was as well unknown.

7 Commands executed

We run the tools over the same data sets, but customising the configuration with the best one for each data set. In the case of PEHaplo, and following authors' indications, input was only trimmed but PEAR was not used.

7.1 Specialized assembly tools

- ***viaDBG***: *Version 1.0*
  Current version of viaDBG only allows dsk as counter. Furthermore, preprocessing included removing duplicated reads, as this boosts efficiency with no accuracy impact.
  - ./bin/viaDBG -p ../PairEndDir/ -o Output -u ../OutputUnitig -k 1...192 -c dsk -n -t 1 --postprocess
  - ./bin/viaDBG -s SingleEndFile -p ../PairEndDir/ -o Output -u ../OutputUnitig -k 1...192 -c dsk -n -t 1 --postprocess
- ***SAVAGE***: *Version: 0.4.0*
  Although last version of SAVAGE has made the selection of minimum overlap automatic, it is recommended to use the value 150.
  - python savage.py -p1 forward.fastq -p2 reverse.fastq -t 32 --split 30
  - python savage.py -s single-end.fastq -p1 forward.fastq -p2 reverse.fastq -t 32 --split 30
- ***PEHaplo***: *Version 1.0*
  Data sets for PEHaplo were modified to fit the software requirements:
  - Reads ids must be all different.
  - Reads names must end with /1.
  - Only fasta format is allowed.
  - python pehaplo.py -f1 forward.fasta -f2 reverse.fasta -l 210 -l1 220 -correct yes -n 3 -r 250 -F 450 -t 32

7.2 Generic assembly tools

- ***SPAdes***: *Version 3.13.1*
  SPAdes has automated the selection of multiple parameters such as *k*-mer size. Therefore, we relied on SPAdes for the parameters selection.
  - python spades.py -s pear.assembled.fastq -1 forward.fastq -2 reverse.fastq -t 32
- ***metaSPAdes***: *Version 3.13.1*
  metaSPAdes has also automated the selection of multiple parameters thus we again relied on metaSPAdes parameters selection.
  - python metaspades.py -s assembled.pear.fastq -1 forward.fastq -2 reverse.fastq -t 32

7.3 PEAR

- ./pear -f forward.fasta -r reverse.fasta -o output_preffix

7.4 Divergence-Abundance Comparison

- ***viaDBG***: viaDBG needs to use longer k-mers when the divergence ratio decreases. Abundance is not managed by any parameter.
  - Divergence above 1%: ./bin/viaDBG -p exp_0.1_1/ ../OutputDir/ -u ../OutputUnitigs -k 120 -c dsk -n -t 32
  - Divergence below 1%: ./bin/viaDBG -p exp_0.1_1/ ../OutputDir/ -u ../OutputUnitigs -k 180 -c dsk -n -t 32
- ***SAVAGE***:
  - python savage.py -p1 forward.fastq -p2 reverse.fastq -t 32 –split 1
- ***PEHaplo***:
  - Relative abundance 50%: python pehaplo.py -f1 forward.fasta -f2 read2.fasta -l 210 -l1 220 -correct yes -n 3 -r 250 -F 450 -t 32 -std 150
  - Relative abundance 33%: python pehaplo.py -f1 forward.fasta -f2 read2.fasta -l 210 -l1 220 -correct yes -n 2 -r 250 -F 450 -t 32 -std 150
  - Relative abundance below 10%: python pehaplo.py -f1 forward.fasta -f2 read2.fasta -l 210 -l1 220 -correct yes -n 1 -r 250 -F 450 -t 32 -std 150
  - Low divergence ratio: python pehaplo.py -f1 forward.fasta -f2 read2.fasta -l 170 -l1 200 -correct yes -n 1 -r 250 -F 450 -t 32 -std 150

References

Baaijens, J. A., Aabidine, A. Z. E., Rivals, E., and Schönhuth, A. (2017). De novo assembly of viral quasispecies using overlap graphs. *Genome Research*, **27**, 835–848.

Johnson, D. S., Yannakakis, M., and Papadimitriou, C. H. (1988). On generating all maximal independent sets. *Information Processing Letters*, **27**(3), 119 – 123.

Pattabiraman, B., Patwary, M. M. A., Gebremedhin, A. H., keng Liao, W., and Choudhary, A. (2015). Fast algorithms for the maximum clique problem on massive graphs with applications to overlapping community detection. *Internet Mathematics*, **11**(4–5), 421–448.

Tomita, E., Tanaka, A., and Takahashi, H. (2006). The worst-case time complexity for generating all maximal cliques and computational experiments. *Theoretical Computer Science*, **363**(1), 28 – 42. Computing and Combinatorics.

ViQUF: de novo Viral Quasispecies reconstruction using Unitig-based Flow networks

Borja Freire, Susana Ladra, José R. Paramá, and Leena Salmela

Abstract—During viral infection, intrahost mutation and recombination can lead to significant evolution, resulting in a population of viruses that harbor multiple haplotypes. The task of reconstructing these haplotypes from short-read sequencing data is called viral quasispecies assembly, and it can be categorized as a multiassembly problem. We consider the *de novo* version of the problem, where no reference is available. We present ViQUF, a *de novo* viral quasispecies assembler that addresses haplotype assembly and quantification. ViQUF obtains a first draft of the assembly graph from a de Bruijn graph. Then, solving a min-cost flow over a flow network built for each pair of adjacent vertices based on their paired-end information creates an *approximate paired assembly graph* with suggested frequency values as edge labels, which is the first frequency estimation. Then, original haplotypes are obtained through a greedy path reconstruction guided by a min-cost flow solution in the *approximate paired assembly graph*. ViQUF outputs the contigs with their frequency estimations. Results on real and simulated data show that ViQUF is at least four times faster using at most half of the memory than previous methods, while maintaining, and in some cases outperforming, the high quality of assembly and frequency estimation of overlap graph-based methodologies, which are known to be more accurate but slower than the de Bruijn graph-based approaches.
Availability: ViQUF is freely available at: https://github.com/borjaf696/ViQUF

Index Terms—*de novo* viral quasispecies assembly, genome assembly, de Bruijn graphs.

1 Introduction

A host organism infected by an RNA virus, such as human immunodeficiency virus (HIV-1), typically carries a population of related but distinct viral haplotypes, i.e., viral quasispecies [1], [2], due to their high mutation rate [3]. However, it is possible to use short-read sequencing data to reconstruct the set of viral haplotypes in a sample with their relative abundance. In this work, we focus on *de novo* viral quasispecies assembly, which does not require a reference sequence. There are reference-based methods for viral quasispecies reconstruction [4]–[6]; however, this approach is constrained to well-known viruses [7] or can be biased for certain studies [8], [9].

De novo viral quasispecies assembly from shotgun reads has been tackled either using overlap graphs [8], [10] or de Bruijn graphs (DBGs) [11], [12]. Overlap graph-based approaches have traditionally been considered accurate but require many resources, whereas DBG-based approaches are usually efficient but less accurate. Recently, we showed that when properly taking advantage of paired-end reads using a paired DBG [13], a DBG-based approach can be as accurate as an overlap graph-based method [12].

Network flows have been used to solve many assembly problems in computational biology [14]–[19]. These approaches first construct an assembly graph (e.g., an overlap graph, DBG, or string graph), where each vertex and edge is assigned a coverage based on the number of reads mapping to it. Then, a network flow that explains the coverage of each vertex and edge is found. Finally, the flow is split into paths, which are the contigs reported by the methods. Pevzner *et al.* [14] and Myers [15] first suggested using network flows to solve genome assembly. Later, Westbrooks *et al.* [17] used network flows in the quasispecies assembly problem. Recently, several other works have also used network flows to address the same problem [18]–[21].

In this work, we present ViQUF, an efficient method for assembly and abundance quantification of viral quasispecies. ViQUF builds an assembly graph using the DBG paradigm and uses network flows to formulate the problem of using paired-end reads to split the assembly graph into an approximate paired assembly graph. We obtain the contigs and their frequencies from the approximate paired assembly graph by solving a network flow problem. Furthermore, we present a mathematically rigorous formulation based on kernel density estimation to determine the threshold for solid k-mer abundance when building the DBG. The experimental results show that ViQUF is faster and more memory efficient than previous methods while maintaining the good quality of the resulting assembly and frequency estimation.

2 Preliminaries

Given an individual carrying a virus with reference genome G and a set of haplotypes $H = \{h_1, h_2, \ldots, h_{n_h}\}$, each one with a relative frequency $f(h_i)$ (or abundance), a

- *B. Freire, S. Ladra, and J.R. Paramá are with Universidade da Coruña, Centro de investigación CITIC, Facultad de Informática, 15071, A Coruña, Spain.*
  E-mail: {borja.freire1,susana.ladra,jose.parama}@udc.es
- *Leena Salmela is with Department of Computer Science, Helsinki Institute for Information Technology, University of Helsinki, Helsinki, Finland*
  E-mail: leena.salmela@cs.helsinki.fi

Manuscript received xxxxx xx, 2020; revised xxxxx xx, xxxx.

shotgun sequencing process produces a set of reads $D = \{r_1, \ldots, r_{n_r}\}$, each read of length l.

The genome G is a sequence of base pairs $G = \{bp_1, \ldots bp_n\}$. Each $h_i \in H$ is G but with three types of changes with respect to the reference genome: substitutions, insertions, and deletions.

Reads are of *paired-end* type; thus, each read $r_j \in D$ has two parts: $L(r_j)$ is the left-hand read and $R(r_j)$ is the right-hand read. $L(r_j)$ covers the base pairs $bp_b, \ldots, bp_e$ and $R(r_j)$ covers the base pairs $bp_{b'}, \ldots, bp_{e'}$ of a haplotype $h_t \in H$. The average number of base pairs between bp_b and bp'_e is called the *insert size*.

Due to sequencing errors, reads might include false changes of base pairs, insertions, and deletions. Given a base pair bp_s of the reference genome G, there are on average C reads covering it; this value is the *coverage* of the sequencing.

From the set of reads D, our goal is to reconstruct the haplotypes in H as complete as possible and estimate the relative frequency of each haplotype in the sample.

Our method is based on a DBG $G = (V, E)$. The vertices $V = \{v_1, ..., v_n\}$ are k-mers, i.e., all subsequences of length k of the reads in D, and there is an edge $e = (v_i, v_j)$ between two vertices v_i and v_j if the last $k-1$ bases of v_i match the first $k-1$ bases of v_j. k is a parameter of the assembly.

3 Our method: ViQUF

Figure 1 shows an overview of the steps of ViQUF. The method selects the k-mers found in the reads in D whose frequency is above a threshold to build a DBG. In that graph, the nonbranching paths are compacted into single nodes, unitigs, producing an assembly graph (AG). For each unitig, we associate a set of unitigs linked to it by paired-end reads. Then, AG is processed by taking every pair of adjacent nodes. For each pair, a new directed acyclic graph (DAG) is built including all unitigs linked to those two nodes by paired-end reads. We then determine a minimum path cover of DAG, where each path represents a haplotype. Therefore, the two nodes for which we computed DAG are divided as many times as paths were found.

For this to work properly, each DAG must be carefully processed to achieve a more reliable graph. This includes transforming each DAG into an offset flow network and solving a min-cost flow problem. With the detected flows, the DAG is corrected, yielding a more reliable graph. Finally, the nodes of the AG are divided based on the path covers of its DAGs, and this results in a new AG, called approximate paired AG (APAG). Once this graph is polished, it is used to derive the contigs and their abundances. Figure 2 shows a running example of this process, which will be used in this section to illustrate how to build the final APAG from the original DBG.

3.1 Obtaining the AG

First we will explain in detail how AG is produced.

The nodes of AG are unitigs, i.e., compacted nonbranching paths of a DBG. Two unitigs are connected with an edge if the corresponding edge also exists in the DBG. AG is augmented with paired-end information by associating a set of paired unitigs $P(U)$ to each unitig U. The paired unitigs are inferred based on the paired-end reads.

3.1.1 Obtaining solid k-mers

We denote a k-mer as *genomic* if it appears in a haplotype of the sample and *nongenomic* if it does not. Recall that, in D, reads contain erroneous changes of base pairs, insertions, and deletions; thus, there can be nongenomic k-mers. Therefore, we use the notion of *solid* k-mer, which denotes a k-mer that occurs at least t times in the reads in D, where t is an abundance threshold. The selection of solid k-mers based on the frequency of appearance [14] works well due to the high coverage used in viral quasispecies assembly. Therefore, it is possible to establish a threshold to separate correct and erroneous information with high level of accuracy. The process is simple; the entire read set is traversed, and those k-mers whose frequency is above the threshold are classified as solid, whereas the rest are considered nongenomic.

There are several methods for selecting the threshold for solid k-mers. Chaisson and Pevzner [22] approximated the number of reads covering a k-mer using a Poisson distribution. They selected a threshold such that few k-mers are expected to be covered by several reads below that threshold. Chaisson *et al.* [23] applied a Poisson and Gaussian mixture model to the k-mer frequency distribution, and the first local minimum is selected as the threshold. Zhao *et al.* [24] used a clustering method based on the variable-bandwidth mean-shift algorithm. In our recent study [12], we used a window over the k-mer frequency histogram to detect the threshold value where no frequency reduction is detected.

We propose a new approach based on estimating the k-mer frequency density function through kernel density estimation using the Gaussian kernel. A brief introduction of kernel density estimation can be found in the supplementary material. The frequency is expected to decrease systematically, corresponding to nongenomic k-mers, up to a certain point where it stabilizes. Then, it subsequently begins to increase again, corresponding to genomic k-mers. Therefore, a good threshold would be the middle point between the minimum point and the previous maximum value of the function. The problem of possible local minima can be mitigated by increasing the suggested bandwidth; thus, producing an intended over-smoothing effect in the density function, as shown in Figure 3.

3.1.2 Building and polishing the AG

The solid k-mers obtained in the previous step are used to build a DBG. We join all the nodes of the unitigs in just one node, as shown in Figure 2(b), to reduce the number of nodes of the graph. The resulting graph is called AG. We use the tool BCALM2 [25] to construct AG.

First, the graph is polished by removing isolated unitigs and tips shorter than a given threshold. Then, we remove filigree edges, an approach previously employed in metagenomics [26]. Filigree edges are the connections between unitigs that are not strongly supported. The idea is not to remove edges based on an overall threshold, which may remove edges corresponding to low-frequent haplotypes. Instead, edges in the graph are tagged as weak or strong based on the connection they support. For instance, the abundance of an edge e, $abu(e)$, denotes the number of reads that support the edge. Then, the abundance of a node

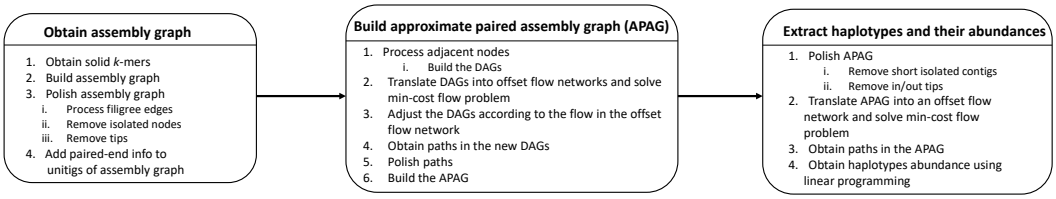


Fig. 1. Overview of the proposed method process.

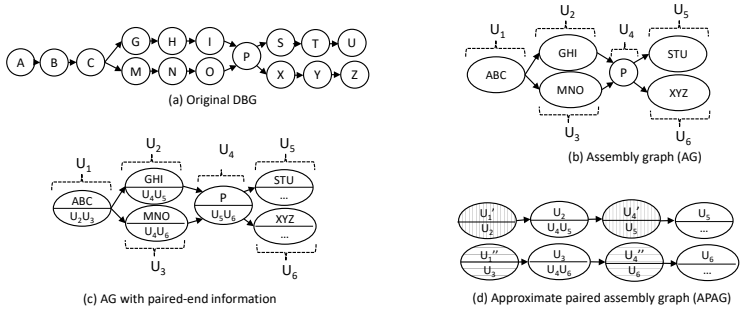
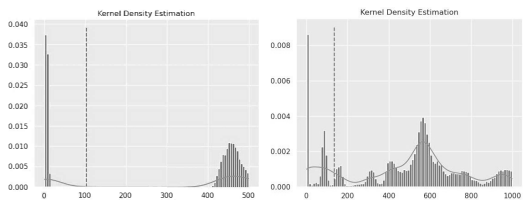


Fig. 2. Example illustrating different steps of the proposed method from the original DBG to APAG.



(a) HCV-10 (b) HIV-real

Fig. 3. Kernel density estimation of k-mer abundance for two datasets of the experimental evaluation (HCV-10 and HIV-real). We indicate with a dotted line the threshold selected for each of the datasets.

v, $abu(v)$, is defined as the maximum $abu(e)$ over all edges e incident to v. Then, every edge $e = (v_1, v_2)$ is tagged as weak if $abu(e) * ratio < min(abu(v_1), abu(v_2))$ and strong otherwise. We set the $ratio$ to a default value of 5, lower than the typical value used in metagenomics, where abundances are lower.

3.1.3 Adding paired-end information to the unitigs

Paired-end information is usually employed in a post-assembly step to join contigs into scaffolds such that the assembly can be extended or repetitive sections can be solved. However, this information has also been used in the previous stages of the assembly process. For instance, the approximate paired DBG (APDB) [13] is a modification of the DBG to use paired-end information directly in contig assembly. The same ideas were also employed for assembling viral quasispecies [12]. The aim is to use the paired-end information before building the contigs to untangle AG. Thus, the paired-end information is included during the graph-building step. This addition requires time and increases space consumption. However, it does not represent a problem for viral datasets in practice.

More precisely, given a k-mer k_d, its paired k-mers are $P(k_d) = \{k_g \mid k_g$ is a solid k-mer and $\exists\, r_x \in D$ and a position j such that $L(r_x)[j \dots j+k-1] = k_d$ and $R(r_x)[j \dots j+k-1] = k_g\}$.

Let K_y be a set of k-mers, $\mathcal{U}(K_y)$ denotes the unitigs where the k-mers in K_y appear. Then, given a unitig U_i formed by k-mers $k_1, k_2, \dots, k_m$, its paired-end information is $P(U_i) = \{\mathcal{U}(P(k_1)), \mathcal{U}(P(k_2)), \dots, \mathcal{U}(P(k_m))\}$. Therefore, to each vertex U_i of the AG, we attach its $P(U_i)$.

An example of paired unitigs is shown in the supplementary material. Figure 2(c) shows our AG with paired-end information.

3.2 Building the APAG

Next, we will transform AG augmented with paired-end information to APAG. Similar to AG, the nodes of APAG are unitigs augmented with paired unitigs. However, in APAG, unitigs that occur in more than one haplotype will be replicated. Each instance of the unitig ideally corresponds to one haplotype and the paired unitigs in the same haplotype. The edges of APAG will connect the unitigs belonging to the same haplotype.

To construct APAG, we take each pair of adjacent nodes in AG and build a DAG, where its nodes are those two nodes and their paired unitigs. The DAG captures the connectivity between the unitigs in AG, by adding edges in DAG between the unitigs linked in AG through short paths.

Therefore, each edge of DAG indicates that the connected nodes probably form part of the same haplotype. Then, we detect the haplotypes in that group of unitigs by finding paths in the DAG. We use that information to split the edge and the two adjacent nodes of AG according to the haplotypes found in DAG.

Therefore, DAGs are the core elements of the proposed method. This implies that they must be built with care to obtain reliable paths, including a correction based on a min-cost problem built on DAG.

3.2.1 Processing adjacent nodes: building the DAGs

Observe that if all occurrences of a unitig U_i are due to a unique haplotype, then all paired unitigs of U_i occur along a path in AG. Thus, they are all reachable from each other. This occurs in our example of Figure 2(c) with, e.g., U_2, with paired unitigs U_4 and U_5. We employ this information to detect haplotypes. For example, the pairing between U_2 and U_4/U_5 highlights one of the two haplotypes in Figure 2: $\{A, B, C, G, H, I, P, S, T, U\} = \{U_1, U_2, U_4, U_5\}$. The other haplotype is $\{A, B, C, M, N, O, P, X, Y, Z\} = \{U_1, U_3, U_4, U_6\}$.

However, if the unitig U_j occurs in several haplotypes, the paired k-mers will span some site containing a mutation. In our example, U_1 appears in two haplotypes; thus, its paired information contains U_2 and U_3, which are not reachable from each other. However, the paired unitigs originating from the same haplotype are reachable from each other. We will use this reachability information to split the nodes of AG into different nodes. Each node corresponds to a different haplotype; thus, producing the new APAG. The new APAG will be used to obtain the contigs.

Let $AG = (V_{ag}, E_{ag})$ be the AG. Our target is to divide the nodes in V_{ag} into as many nodes as the number of haplotypes they belong to. Thus, we follow the next steps:

1) For each pair of adjacent nodes of V_{ag}, say U_i and U_j, a directed acyclic graph $DAG_{ij} = (V_{ij}, E_{ij})$ is built, where
   - $V_{ij} = U_i \cup U_j \cup P(U_i) \cup P(U_j)$.
   - There is an edge $e \in E_{ij}$ from $u \in V_{ij}$ to $v \in V_{ij}$, if and only if there is a path in AG from u to v that does not include any other node in V_{ij} and has a length shorter than 2Δ, where Δ is the maximum error in the insert size. When searching for a path, we also limit the number of traversed branches by a threshold T to ensure that the path search remains tractable. By default, we set T to 10, which is a conservative value in order to avoid removing low abundance haplotypes.

   Observe that the nodes of DAG are the two adjacent (in AG) unitigs U_i and U_j, and their paired unitigs. U_i and U_j overlap by $k-1$ bp[1]; thus, their paired unitigs (separated from U_i and U_j, on average, by the insert size) should also be close by in AG because their distance in AG should not exceed 2Δ, twice the insert size error if they belong to the same haplotype.
2) Next, we will find the source to sink paths in DAG such that the paths correspond to the haplotypes. However, not all paths correspond to haplotypes. To identify the correct paths, we will use the coverage of the DAG nodes and edges along with the paired-end information of the reads.
   The coverage of the DAG edges is computed based on the coverage of the unitigs on the path in AG that the edge represents, where the coverage of a unitig is the average abundance of the k-mers of the unitig. When computing the coverage of an edge (u, v) of DAG, the process analyzes the paths in AG connecting them. It means that if there is only one path p, the edges that reach nodes of p from nodes that are not in p and those coming from nodes in p and reaching nodes that are not in p correspond to haplotypes different from that containing u and v. Thus, we analyze the paths connecting U_i, U_j, u, and v. We traverse those paths. At each node, we add to a bag the coverages of edges coming from nodes that do not belong to p. Then, we distribute those *excess* coverages in the bag among the nodes forming a fork when we find one. Therefore, we combine the coverage and paired-end information of the reads.
   The final coverage is that of the last edge of p minus the incoming coverages that were not assigned to nodes leaving p. The exact method for computing the coverages is described in more detail in the supplementary material.
   Once the edge and node coverages are computed, we want to find a set of paths in DAG and associated abundances to explain the coverage of edges and nodes.
   Figure 4 shows the process of building a reliable DAG for detecting the haplotypes that pass through the adjacent nodes U_2 and U_4. Figure 4(a) shows AG of our running example with the coverages of nodes and edges, i.e., the number of reads supporting those nodes and edges. Figure 4(b) shows the first version of DAG DAG_{24}. Observe that there could be two paths U_2, U_4 and U_5, and U_2, U_4, and U_6; thus, we must check if they are correct using the coverages of edges in AG and the paired information. Figure 4(b) shows that, even after applying the previous method for computing the coverage of the edges of DAG, the coverages may not be consistent. Observe that U_4 has two exits, adding a total output coverage of 12. This is not possible since U_4 only receives an input coverage of 10.

Therefore, the coverages of the graph are inconsistent. Moreover, there can be false edges due to false paths in AG. Their origin is due to the shared k-mers between different but similar haplotypes. Additionally, there can be several real edges and thus paths, but the observed coverages may not be consistent with each other.

1. Recall that AG is a compaction of the DBG, so edges between unitigs are actually edges of the DBG, and thus the end of the source unitig overlaps $k-1$ bp with the beginning of the target unitig.

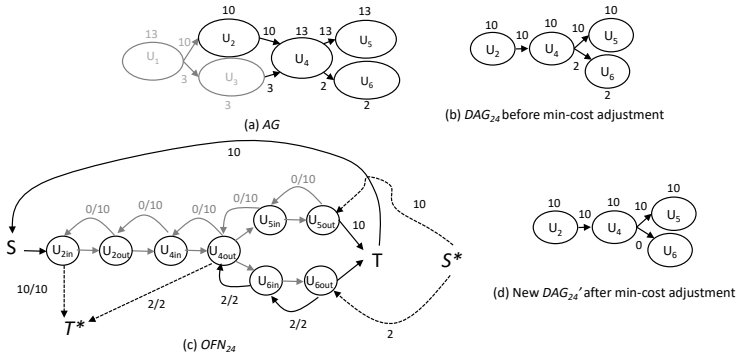


Fig. 4. Example of the construction and coverage adjustment of DAG corresponding to the adjacent nodes U_2 and U_4. We show the original AG, DAG_{24}, OFN_{24}, and DAG'_{24}

To solve these problems, we employ the coverage of the unitigs (nodes of DAG), i.e., the number of reads supporting them and their connections.

We need to assign a flow to the graph to satisfy the *conservation* property of the flow.

It means that the amount of flow reaching each node must be the same as those leaving the node. A simple heuristic, such as taking first the heaviest paths, will work accurately with a graph in that form. However, conservation is not enough; the flow assigned to each node/edge must be as close as possible to the observed coverage.

Our example, DAG_{24}, is simple, but in a very complex graph, a decision on the flow distribution must be taken at each fork; however, the flow that reaches the fork might depend on several previous decisions. Therefore, the optimal solution is very complex. We will use a min-cost flow that will solve the two problems using a cost function, i.e., it will modify DAG such that i) satisfies the conservation law, ii) the new flow assigned to the edges is close to the observed coverage.

To assign a flow minimizing the difference from the observed coverage of the unitigs in DAG, the min-cost flow uses a cost function that increases when the difference grows. Therefore, we obtain our solution by minimizing that cost. The main advantage of this method is that it decides at all forks using a polynomial-time algorithm, minimizing the difference between the assigned flow and observed coverage *globally*. It means that it does not compute the best cost in the fork but computes the cost as the sum of costs at all edges of the graph to decide the flow in each edge of a fork.

With the corrected graph, the next step is to decompose the flow into paths in the graph in the most parsimonious way. Finally, the corrected flow and its decomposition into paths will give us the relative abundance of the haplotypes.

Consider the following: let $cov(v)$ be the coverage of the nodes $v \in V_{ij}$, $cov(u, v)$ be the coverage of the edges $(u, v) \in E_{ij}$, and $c(y)$ a cost function that can be applied to nodes $v \in V_{ij}$ or edges $(u, v) \in E_{ij}$. Following the approach proposed by [16], we use, for our problem, the same cost function for nodes and edges $c(x) = x^2$. Here, x is the error in the frequency estimation for nodes and edges.

The objective is to find the set of paths $\mathcal{P}_{ij}$ from the sources of DAG_{ij} to the sinks of DAG_{ij} that minimizes:

$$err = \sum_{u \in V_{ij}} c_u\big(|cov(u) - \sum_{p \in \mathcal{P}_{ij} | u \in p} a(p)|\big) + \sum_{(u,v) \in E_{ij}} c_{uv}\big(|cov(u,v) - \sum_{p \in \mathcal{P}_{ij} | (u,v) \in p} a(p)|\big),$$

where $a(p)$, for each path $p \in \mathcal{P}_{ij}$, is the estimate abundance level of the path obtained by the solution of the min-cost flow problem and the decomposition of the flow into paths.

3.2.2 Translating DAGs into offset flow networks

In this section, we explain how we use the min-cost network flows to first determine an optimal flow in DAG and then split the flow into paths. A brief introduction to network flows can be found in the supplementary material.

To determine such an optimal flow, we translate DAGs into offset flow networks and solve the min-cost problem in those offset flow networks. The min-cost flow in the offset flow network models how the coverages in DAG need to be modified to create an optimal feasible flow.

For each $DAG_{ij} = (V_{ij}, E_{ij})$, we build the offset flow network OFN_{ij} as follows:

- Replace every node $v \in V_{ij}$ by two nodes v_{in} and v_{out} and two arcs: (v_{out}, v_{in}), with capacity $b(v_{out}, v_{in}) = cov(v)$ and cost function $c_{v_{out},v_{in}}(x) = c_v(x)$, and (v_{in}, v_{out}) with infinite capacity and the same cost function. The flow on the arc (v_{out}, v_{in}) will be nonzero if the coverage of v in DAG needs to be decreased in the optimal solution. However, the flow on the arc (v_{in}, v_{out}) will be nonzero if the coverage of v needs to be increased.
- Replace the arc between each pair of original nodes $(u, v) \in E_{i,j}$ by two edges: (v_{in}, u_{out}) with capacity $b(u, v) = cov(u, v)$, and cost function $c_{v_{in},u_{out}}(x) = c_{u,v}(x)$, and (u_{out}, v_{in}) with infinite capacity, and the same cost function. Similarly, a nonzero flow on the forward arcs indicates that the coverage of the arc (u, v) should be increased. In contrast, a nonzero flow on the backward arcs indicates that the coverage should be decreased.
- Add start (S) and target (T) nodes. For each source node s in DAG_{ij}, add an arc from S to s with cost

$c_{S,s}(x) = 0$ and infinite capacity. Similarly, for each target node t in DAG_{ij}, add an arc from t to T with $c_{t,T}(x) = 0$ and infinite capacity. Source nodes are the nodes with no incoming arcs, and target nodes are the nodes with no outgoing arcs.

- Add an artificial source (S^*) and sink (T^*) to manage the exogenous flow and initialize the exogenous flow of each node of OFN_{ij} to $q_v = 0$. These will be used to balance the flow on nodes where the total coverage of the incoming arcs and the total coverage of the outgoing arcs do not match.
- For any node v with

$$q_v = \sum_{u \in V} cov(v, u) - \sum_{u \in V} cov(u, v) < 0$$

add an arc from S^* to the node v with capacity set to q_v and cost $c_{S^*,v}(x) = 0$. Update the exogenous flow of S^*: $q_{S^*} = q_{S^*} + q_v$.

- For any node v with

$$q_v = \sum_{u \in V} cov(v, u) - \sum_{u \in V} cov(u, v) > 0$$

add an arc from v to T^* with capacity set to q_v and cost $c_{v,T^*}(x) = 0$. Update the exogenous flow of T^*: $q_{T^*} = q_{T^*} + q_v$.

- Add an edge from T to S to close the offset network with cost $f(T, S) = 0$ and infinite capacity.

Figure 4(c) shows the resulting network flow for the offset network of DAG_{24} of Figure 4(b). On the edges, the label x/b means that b is the node's capacity, and x is the flow that passes through it when solving the min-cost problem. Edges with only a value x means that the edge has ∞ capacity, and the flow passing through it is x.

3.2.3 Adjusting DAGs according to the flow in the offset flow network

Once the min-cost problem in the offset flow network is solved, the original DAG is modified as follows.

Let us consider DAG_{ij} and its OFN_{ij}. Let $cov(u, v)$ be the coverage of the edge $(u, v) \in DAG_{ij}$, $x_{u_{out}v_{in}}$ be the flow through the edge (u_{out}, v_{in}) in OFN_{ij}, and $x_{u_{in}v_{out}}$ be the flow through the edge (u_{in}, v_{out}) in OFN_{ij}. Then, the new coverage $cov(u, v)'$ of the edge (u, v) in the modified DAG'_{ij} is: $cov(u, v)' = cov(u, v) - x_{u_{in}v_{out}} + x_{u_{out}v_{in}}$.

Observe in Figure 4(d) the new DAG'_{24}, the output flows of U_4 match the input flows.

3.2.4 Obtaining paths in the new DAGs

For each adjusted DAG'_{ij}, we build a set of paths $\mathcal{P}_{ij} = p_1, p_2, p_3, \ldots, p_p$ from the sources to the sinks of DAG'_{ij} with weights $w_1, w_2, w_3, \ldots, w_p$ satisfying:

$$cov(u, v)' = \sum_{\forall p_t \in \mathcal{P}_{ij} | u \in p_t \text{ and } v \in p_t} w_t$$

for all edges (u, v) of DAG'_{ij}.

Analogously to prior research to solve RNA-seq problems [16], we are interested in *parsimoniously* explaining the abundance distribution over the sample. We are interested in finding the set of paths $\mathcal{P}_{ij}$ with the lowest number of paths. A well-known result is that decomposing a flow into a minimum number of paths is an NP-hard problem in a strong sense. Therefore, only heuristic methods can be run in a reasonable time. In this work, we use a mixture of two heuristics to build the final set of paths.

We iteratively find the source to sink paths in the graph until the paths are a decomposition of the flow. We resolve branches primarily using paired-end information and secondarily choosing the heaviest path when the paired-end information is unavailable or does not point to a unique solution. To achieve this, we proceed in two steps as follows:

- For each node $u \in V_{ij}$, we store the list of nodes in V such that u is part of their paired-end information, i.e., $paired\_with[u] = \{v_1, \ldots, v_n\}$ iff $u \in P(v_i), i = 1, \ldots, n$.
- When traversing the graph to find a path, we remember the parent of the last node on the path with an indegree greater than one. We call this saved node haplotype reference node. When we find a node with an outdegree greater than one, we must decide which branch to follow. Then, we take the haplotype reference node and employ the map built in the previous step. There are three possibilities: none of the branches of the fork is pointed by any node in the paired-end information of the haplotype reference node, then we use the heaviest path heuristic; only one branch of the fork is pointed, then we follow that path; more than one branch is pointed, then we follow the heaviest path of the pointed branches.

In the example of Figure 4(d), the only resulting path is $p_1 = U_2, U_4, U_5$, with weight 10. We have discarded the path U_2, U_4, U_6, with weight 0, which, if not removed, would be responsible for mixing haplotypes.

Additionally, in DAGs, we have information about the relative abundance of the haplotypes.

3.2.5 Polishing the paths

It is necessary to solve one min-cost flow problem for each pair of adjacent nodes in AG. Ideally, the number of paths reported for each pair coincides with the number of haplotypes to which the pair belongs. However, this does not happen in practice due to, for instance, shared regions or coverage inconsistencies; thus, a wrong number of paths might be reported. Therefore, a soft polishing is performed to overcome this problem. This polishing involves tasks, such as removing paths that do not contain both nodes under study, paths with suggested flow below the k-mer solid threshold, and paths with low simultaneously support from the given nodes.

The expected output once the polishing has been completed is to have for each pair of nodes both the haplotypes it belongs to and the estimation of the abundance of each haplotype given by the suggested flow.

3.2.6 Building the APAG

For each pair of adjacent nodes U_i and U_j in AG, we take the set of paths $\mathcal{P}_{ij}$, and for each path $p \in \mathcal{P}_{ij}$, unless they are already in APAG, we create the nodes:

- $U_{iP(U_i) \cap p}$, which is the node with label U_i with paired information $P(U_i) \cap p$.

- $U_{j_{P(U_j)\cap p}}$, which is the node with label U_j with paired information $P(U_j) \cap p$.

In our example, DAG_{24} produced only one path $\mathcal{P}_{24} = \{p_1\}$, and $p_1 = \{U_2, U_4, U_5\}$. Then, we create two nodes:

- $U_{2_{P(U_2)\cap p_1}}$. Since $P_{U_2} = U_4, U_5$, we create the node labeled U_2 in Figure 2(d) with paired information U_4, U_5.
- $U_{4_{P(U_4)\cap p_1}}$. Since $P_{U_4} = U_5, U_6$, we create the node labeled U_4' in Figure 2(d), with paired information U_5.

As shown in Figure 2(d), *APAG* of our running example and our nodes U_2 and U_4' are in the upper haplotype; the remaining nodes of the haplotype are computed from DAGs DAG_{12} and DAG_{45}.

The nodes U_4' and U_4'' correspond to the same unitig, but those nodes have different paired information, meaning that they correspond to different haplotypes.

3.3 Extracting the haplotypes and their abundances

Finally, we polish APAG by removing isolated nodes, which can be wrongly built when splitting one node more than necessary. We also remove short tips and paths with lengths below 500 bp, which are not expected to be correct contigs. After polishing APAG, the only remaining step is to traverse it and report the correct paths.

For this, we solve a min-cost flow problem on APAG. APAG is transformed into an offset network using the same method used previously for DAGs. Once the offset network is built, a min-cost flow is solved on it using cost function $c_{uv}(x) = (x - cov(u, v)')^2$. Therefore, the first paths to be flooded will be the ones with more flow capacity. Then, we obtain the haplotypes and their abundances by decomposing the flow into weighted paths using a greedy algorithm to extract the heaviest path.

A similar approach has been previously introduced by VG-Flow, which solves a min-cost flow problem over a variation graph, using the number of paired-end reads mapping a particular edge as edge labels. Then, it performs a greedy path extraction to obtain the set of paths P and solves a linear programming problem to polish the results. In contrast, ViQUF skips the variation graph using APAG. It also avoids mapping the reads using the suggested flows as labels for the edges. Although ViQUF could obtain the haplotype abundance estimation based on the estimated flows in APAG, the estimation for some haplotypes may not be accurate due to the lack of precision on the abundances for heads/tails plus some wrong estimations that can appear on the split computation. Therefore, like VG-Flow, ViQUF performs a flow polishing based on a simple linear programming problem. Given $APAG = G = (V, E)$ a graph, $P = \{p_1, p_2, \ldots, p_n\}$ a set of paths on G, and $f(p_i)$ a flow over the paths, we define

$$\begin{aligned} \min_{x} \quad & \sum_{u \in V} |ab_u - \sum_{p_i, u \in p_i} f(p_i) x_{p_i}| (\frac{L_u}{k} - 1) \\ \text{s.t.} \quad & x = [x_{p_1}, \ldots, x_{p_n}] \in R^n_{\geq 0} \end{aligned} \tag{1}$$

where L_u and ab_u are the length and abundance of the unitig, respectively; k is the k-mer size, and $f(p_i)x_{p_i}$ is the flow of the path p_i after polishing. Equation 1 minimizes the accumulated difference for each vertex and the assigned flow for each path it belongs to. The second term of the expression is a correction factor that gives a higher weight to the longest; thus, the most reliable, unitigs. The aim is to obtain, for each path, a weight x_{p_i} that minimizes the difference between the abundance of each unitig and the given flow. Although simple, it is quite effective in practice and, even for complex cases, the frequency estimation improves significantly compared to the previous estimation.

3.4 Theoretical complexity analysis

3.4.1 Theoretical time complexity

To build the initial AG, we must build and traverse a DBG from the reads. We use GATB library [27] for this step, which takes $\mathcal{O}(N)$ time, where N denotes the total length of all reads.

To build APAG, we need to associate each unitig with the unitigs through the paired-end information. To do so, we traverse the read set and, for each pair of k-mers in the left and right reads, we locate the corresponding unitigs and add the right unitig to the set of paired unitigs of the left unitig. Assuming constant time access to the unitigs using a hash function, this takes $O(N)$ time.

To build DAG_{ij} for two nodes U_i and U_j of AG, we traverse AG from the first node U_i until we reach a specific distance (we use 2Δ by default) without finding any node from $pe = P(U_i) \cup P(U_j)$. In the worst case, the number of iterations of this step is $n_h(k+2\Delta)$, where n_h is the number of haplotypes. This leaves us a complexity of $\mathcal{O}(n_h(k+2\Delta))$.

After building DAG, it is polished. For this, DAG is traversed. This costs $\mathcal{O}(|V_{ij}| + |E_{ij}|)$, where $|E_{ij}|$ and $|V_{ij}|$ are the number of edges and vertices of DAG_{ij}, respectively. Furthermore, in the worst case $|V_{ij}| = n_h(k + 2\Delta)$, indicating that the insert size is maximum and every k-mer within the insert size belongs to pe. In DAGs, the $|E_{ij}|$ is bounded to $\binom{|V_{ij}|}{2}$. However, even in the worst case scenario, this is impossible, and the upper limit is $4|V_{ij}| = 4n_h(k + 2\Delta)$, meaning that each node is at most connected to four other nodes. Again, the complexity of this step is also $\mathcal{O}(n_h(k + 2\Delta))$.

The third and hardest step is solving the min-cost flow problem. There are two possibilities: if the cost function is concave or convex. If it is concave, the problem is NP-hard; however, the problem can be solved in polynomial time if the cost function is convex [28]. Given a circulation problem on a network $N = (G, l, u, c)$, where G is a directed graph, l and u are non-negative functions for demand and capacity for every arc, respectively, and c is a convex cost function, the minimum-cost circulation, flow, problem can be solved through cycle-canceling algorithm in $\mathcal{O}(nm^2CU)$, where $n = |V(G)|$, $m = |E(G)|$, C is the maximum value of the costs, and U is the maximum capacity in an edge. As described earlier, $n \leq n_h(k + 2\Delta)$ and $m \leq 4n$; thus, the algorithm runs in $\mathcal{O}(n^3CU) = \mathcal{O}((n_h(k + 2\Delta))^3CU)$.

Finally, building and polishing DAG and solving the min-cost flow problem is repeated for each edge or pair of adjacent nodes in AG, which is four times the number of k-mers in the sample in the worst case scenario. Thus,

the theoretical complexity of these steps is $\mathcal{O}((N)(n_h(k+2\Delta))^3 CU)$.

The final step is to solve a minimum-cost flow over APAG, which in the worst case is the same as AG, which in the same scenario is DBG. Therefore, we can use the same previous complexity $\mathcal{O}(nm^2CU)$, but in this case $n \leq N$ and $m \leq 4n = 4(N)$, resulting in a complexity of $\mathcal{O}(N^3CU)$.

As summary, the entire algorithm takes $\mathcal{O}((N)(n_h(k+2\Delta))^3 CU + (N^3CU))$. If we assume k and Δ are constants, we obtain $\mathcal{O}(N^3CU)$. According to the theoretical analysis, building APAG and solving the min-cost flow in APAG are the most time-consuming steps. However, building APAG takes the longest in practice.

3.4.2 Theoretical space complexity

The three main structures of ViQUF are the *de Bruijn* graph, the assembly graph and the APAG. In the worst case scenario where no k-mers have been filtered, the *de Bruijn* graph can be represented in $\mathcal{O}(N)$ space, where N denotes the total length of all reads. However, for the assembly graph we need to store the paired-end information of each node and in the APAG we have potentially split every node in the assembly graph n_h times. The assembly graph is a compaction of the de Bruijn graph and thus the graph itself can be represented in $O(N)$ space. The total number of k-mer pairs we extract from the reads is $O(N)$ and thus also the paired-end information can be represented in $O(N)$ space. The representation of the APAG will need $O(N * n_h)$ space. Thus, we have a final space complexity of $\mathcal{O}(N * n_h)$. However, n_h has a bounded value, thus we can treat it as a constant and therefore the final space complexity is $\mathcal{O}(N)$.

4 Experimental evaluation

We compare ViQUF with the most recent solutions for *de novo* haplotype-aware full-length viral quasispecies assembly: Virus-VG [19] and VG-Flow [29]. We also include in this comparison viaDBG [12], a related solution for *de novo* assembly of viral quasispecies, and PEHaplo [10] (which do not provide haplotype abundance estimations). Additionally, we compare ViQUF with two reference-based viral quasispecies construction methods: CliqueSNV [6] and PredictHaplo [5]. We followed the recommendations given by the authors in their papers to set the parameters of the methods. For ViQUF, we set the k-mer size to 121.

4.1 Benchmarking data

In our experimental evaluation, we used simulated and real MiSeq sequencing data. We employed the methodology and datasets used by related work [8], [19], [29], which are described below.

Real data with ground truth. We used a gold standard benchmark for viral assembly [30]. The reads were produced from five HIV haplotypes using Illumina MiSeq (2×250 bp with an error of about 0.5% [31]) with 20000x coverage. Since the five haplotypes in the sample are known, it is possible to validate the achieved results. Table 1 presents the main characteristics of this dataset (HIV-real).

Synthetic benchmarks. Four simulated datasets were used, consisting of 2×250 bp Illumina reads from different viruses:

TABLE 1
Main characteristics for the datasets used in the experiments.

	Virus Type	Genome Length (bp)	Average Coverage	# Haplo-types	Abun-dance	Diver-gence
HIV-real	HIV-1	9487–9719	20000×	5	10%–30%	1%–6%
HIV-5	HIV-1	9487–9719	20000×	5	5%–28%	1%–6%
ZIKV-15	ZIKV	10251–10269	20000×	15	1%–13%	1%–12%
HCV-10	HCV-1a	9273–9311	20000×	10	5%–19%	6%–9%
POLIO-6	Poliovirus	7428–7460	20000×	6	1.6%–51%	1.2%–7%

Human immunodeficiency virus (HIV), Poliovirus (POLIO), Hepatitis C virus (HCV), and Zika virus (ZIKV). These datasets correspond to those employed by [29] in their experiments. We denote these datasets by HIV-5, POLIO-6, HCV-10, and ZIKV-15, respectively. Table 1 also shows the main characteristics of these datasets.

4.2 Evaluation metrics

Ground truth sequences are available for all out datasets. Thus, we evaluate the assemblies by comparing the obtained contigs against these. We used MetaQUAST [32] for this evaluation. MetaQUAST gives multiple alignment statistics and an overall assembly evaluation. We used MetaQUAST with the option "–unique-mapping" meaning that each contig can map to only one haplotype. For each dataset, we report genome fraction retrieved, N50, and errors (misassemblies and percentage of mismatches/indels/Ns). The genome fraction is defined as the percentage of the target haplotypes contained in the set of the obtained contigs (all contigs with lengths larger than 500 bp). We used N50 to measure the fragmentation level of the assembly. The N50 is defined as the length of the shortest contigs in the assembly of at least the length of half of the total assembly. We show misassemblies to measure the behavior on repetitive sections, and error rate, which is calculated as the sum of mismatch rate, indel rate, and N-rate, measures the correctness of the assembly, showing if a set of contigs is correct or just a random graph extension. The supplementary material includes the results for the measures of precision and recall, following the extension proposed by [33] for *de novo* methods.

4.3 Performance analysis

In this section, we present several evaluations to verify the quality of the obtained contigs. Table 2 presents the overall results of the five tools on the different datasets. Section 5 of the supplementary material shows the results for the main statistics of this table using graphs for a better view.

The *de novo* reconstruction methods, Virus-VG, VG-Flow, viaDBG, and PEHaplo, and the reference-based methods, PredictHaplo and CliqueSNV, were configured with the settings given by their authors [5], [6], [10], [12], [19], [29]. We use the optimal values to determine the solid k-mers for viaDBG and ViQUF. A discussion on the impact of these values can be found in the supplementary material.

The results of the reference-based methods show high variability in terms of genome fraction and error rate; however, N50 is almost the length of the entire haplotype for all datasets. For example, on HCV-10, using the same

TABLE 2
Results for the seven viral quasispecies assembly tools. We show percentage genome (the fraction of all haplotypes retrieved by each method), N50 (the length of the shortest contig needed to be included to cover at least half of the total assembly), the number of misassemblies, the error rate of the assembly (the sum of mismatch rate, indel rate, and N-rate), elapsed time, peak memory usage, mean estimated error (MEE) of haplotype frequencies, and the standard quasideviation of the estimated error of haplotype frequencies ($\hat{S}_{EE}$). For Virus-VG and VG-Flow, we show the elapsed time and memory usage separated into the time for contig assembly using SAVAGE (first value) and for the full haplotype reconstruction (second value). Similarly, for the reference-based methods, PredictHaplo and CliqueSNV, we show the elapsed time and memory usage to align the reads to the reference (first value) and run the method (second value) separately. The best values for each data set in each column are highlighted.

dataset	method	% Genome	N50	misass-emblies	% error rate	elap time (min)	memory (GB)	MEE (%)	$\hat{S}_{EE}$
HCV-10	Virus-VG	99.30%	9231	0	0.002	913.48 + 1009.08	26.13 — 8.35	**0.05**	**0.04**
	VG-Flow	**99.79%**	**9293**	0	**0.001**	913.48 + 559.56	26.13 — 8.29	**0.05**	**0.04**
	viaDBG	97.72%	8934	0	0.005	69.10	2.81	-	-
	PEHaplo	94.78%	8661	0	0.013	68.45	8.94	-	-
	PredictHaplo	89.79%	9273	0	0.044	4.11+175.73	0.08 — 1.14	6.74	6.44
	CliqueSNV	9.97%	9273	0	2.100	4.11+3494.09	0.08 — 17.24	17.6	25.10
	ViQUF	97.37%	8911	0	0.008	**3.51**	**1.09**	0.15	0.13
HIV-5	Virus-VG	96.85%	9632	2	0.332	1619.34 + 312.68	26.83 — 0.64	6.12	5.57
	VG-Flow	96.87%	9625	2	0.331	1619.34 + 312.20	26.83 — 0.65	5.25	4.79
	viaDBG	97.50%	8046	2	**0.151**	62.34	2.89	-	-
	PEHaplo	78.59%	9328	2	0.690	73.33	4.84	-	-
	PredictHaplo	**99.90%**	**9663**	0	0.591	4.00+120.13	0.09 — 1.05	6.26	5.59
	CliqueSNV	99.86%	9649	0	1.152	4.00+93.67	0.09 — 8.51	7.02	4.06
	ViQUF	99.71%	9237	2	0.321	**3.26**	**1.07**	**3.09**	**1.88**
POLIO-6	Virus-VG	89.96%	**7436**	0	0.141	3455 + 201.23	17.30 — 0.73	**1.56**	**1.18**
	VG-Flow	**99.49%**	7388	2	0.137	3455 + 532.33	17.30 — 0.30	2.18	2.48
	viaDBG	73.81%	1760	0	**0.018**	49.21	2.52	-	-
	PEHaplo	98.15%	7428	0	0.125	107.96	3.63	-	-
	PredictHaplo	49.81%	7428	0	0.646	3.80 + 82.35	0.10 — 0.92	11.08	11.65
	CliqueSNV	83.07%	7428	0	1.844	3.8 + 27.95	0.10 — 8.45	4.74	5.27
	ViQUF	97.40%	7428	0	0.247	**2.61**	**1.06**	3.05	1.79
ZIKV-15	Virus-VG	**99.56%**	10212	0	0.077	706 + 407.51	13.45 — 1.37	0.94	0.70
	VG-Flow	83.05%	10210	0	0.144	706 + 406.22	13.45 — 0.62	2.00	1.85
	viaDBG	89.85%	1398	0	0.110	65.48	3.25	-	-
	PEHaplo	98.32%	10247	0	2.05	321.53	8.80	-	-
	PredictHaplo	46.65%	**10251**	0	0.133	4.06+149.68	0.08 — 1.11	6.33	7.68
	CliqueSNV	66.66%	**10251**	0	**0.036**	4.06 + 126.28	0.08 — 8.38	1.18	1.34
	ViQUF	99.08%	10140	0	0.142	**4.05**	**1.11**	**0.25**	**0.29**
HIV-real	Virus-VG	83.36%	8637	0	3.384	3550 + 440.71	26.85 — 0.80	-	-
	VG-Flow	89.99%	5950	0	1.100	3550 + 1499.61	26.85 — 1.47	-	-
	viaDBG	87.25%	1813	0	0.197	17.24	3.74	-	-
	PEHaplo	91.43%	1262	0	**0.074**	68.34	3.48	-	-
	PredictHaplo	90.21%	**8702**	0	0.287	4.71 + 100.75	0.09 — 0.87	-	-
	CliqueSNV	72.17%	8676	0	1.125	4.71 + 136.68	0.09 — 9.03	-	-
	ViQUF	**90.85%**	2267	0	0.292	**3.73**	**1.07**	-	-

alignment file, CliqueSNV only retrieves one genome, obtaining 9.97% of genome fraction, whereas PredictHaplo retrieves nine out of 10 genomes. The opposite situation occurs in POLIO-6 and ZIKV-15. On HIV-real, PredictHaplo performs well in all metrics, whereas CliqueSNV retrieves less than four of five genomes, 72.17% with a higher error rate.

The results of the *de novo* approaches show that although all tools have an overall good performance in terms of genome fraction, on average ViQUF achieved the highest level of genome fraction retrieved. However, there are some cases where large differences appear. The first case is POLIO-6, where viaDBG with the optimal configuration retrieves only 73% of the genome fraction, whereas the remaining methods range from 89% to 99%. In ZIKV-15, VG-Flow achieves a genome fraction of 83%, whereas the rest cover more than 89% of the genome. This difference may seem small, but the loss of one complete haplotype causes a 6% loss in genome fraction for 15 haplotypes. Finally, for HIV-5, PEHaplo achieves only 78% of the genome fraction with twice the error rate compared to the remaining *de novo* tools.

Analyzing the length of the obtained contigs, N50 shows that viaDBG loses by far against the rest of the tools. However, ViQUF and PEHaplo are slightly below Virus-VG and VG-Flow, but they can obtain competitive results. We note that DBG methods usually produce more fragmented assemblies than overlap methods because of using k-mers instead of full-length reads.

We also evaluated the number of misassemblies and the error rate. All tools perform well in these aspects. There are only a few examples where VG-Flow, Virus-VG, and PEHaplo have quite high error rates. Virus-VG and VG-Flow have an extremely good performance on N50 but with many errors in HIV-real. Therefore, we assume that the large N50 is due to the mixing of multiple haplotypes. A similar situation occurs for PEHaplo in ZIKV-15 where the error level is very high, but N50 and genome fraction are almost perfect.

The previously published values [19], [29] did not show this anomaly. The reason is probably that Virus-VG and VG-Flow were evaluated with QUAST version 4.3, whereas we

used MetaQUAST from QUAST version 5.0. Furthermore, in previous evaluations of Virus-VG and VG-Flow, contigs were allowed to align to several haplotypes, whereas we only allowed one contig to align at one place. Although one can think that this evaluation method may impact the genome fraction level, making it smaller, this is not the case since our experiment shows similar values.

Another interesting result is the misassemblies for HIV-5, where all *de novo* tools have at least two misassemblies. The reason for these strange results is the terminal repeats of the HIV genome. Therefore, no matter the graph methodology, if the shared region between the start and end of the genome is longer than the k-mer, for DBG methods, or the average read, for overlap-based methods, they will probably extend longer than the actual genome.

The overall comparison between reference-based and *de novo* methods supports *de novo* approaches as the most reliable ones. However, the divergence levels in these datasets range from 1% to 12%, which might be high for a short-term virus infection. We have performed two extra experiments in Section 4.6 by simulating two new datasets with a divergence between 0.1% to 2%.

4.4 Haplotype abundance estimation

This section evaluates the precision of the haplotype relative frequency estimations. PEHaplo and viaDBG are excluded since they do not provide this information, as stated before. Again, we used the last version of MetaQUAST with the "–unique-mapping" set. If it is not set, one contig might be aligned to several haplotypes; thus, misplacing it and making frequency estimation harder or even impossible.

To evaluate the haplotype relative frequency estimation error, we use two different measures. More specifically, the Mean Estimation Error (MEE), which is the average error per haplotype, and the estimation error standard quasideviation ($\hat{S}_{EE}$), which measures the amount of dispersion in the frequency error estimations.

We will only consider the set of contigs with N50 larger than 75% of the genome length; thus, the contigs assigned to each haplotype should be long enough to estimate the haplotype frequency. Additionally, more than one contig could cover a particular haplotype. This can happen in complex or simple datasets at the beginning and end of the haplotype, where frequencies are expected to be less reliable. For a fair comparison, the estimated frequency for one haplotype belongs to the longest contig in these situations. Besides, some short contigs sometimes appear in assemblies because of repetitive sections or failures in the assembly. These contigs are expected to have lower or inaccurate frequencies; thus, they will not be considered in the evaluation.

In this experiment, from the *de novo* perspective, there is no clear winner. Virus-VG and VG-Flow perform better than ViQUF for HCV-10 and POLIO-6. However, ViQUF outperforms these techniques for ZIKV-15 and HIV-5, obtaining lower estimation errors and less variability. Overall, the three tools can properly estimate the haplotype frequencies for the datasets. These estimations are accurate for HCV-10 and ZIKV-15, with estimation errors per haplotype below 2% on average. However, the three techniques obtain lower levels of accuracy for POLIO-6 and HIV-5. One possible explanation is the high level of similarity between haplotypes. This high level of similarity (Table 1) means that the paired-end information of unitigs shared between two or more haplotypes point to other sets of unitigs that are also shared. Therefore, this complicates the estimation of the abundance frequencies. Nevertheless, all tools can assemble haplotypes with high levels of accuracy. However, for POLIO-6, the differences in abundance create even more problems on low abundance haplotypes with lower levels of paired-end information and a lower level of reliability on their frequencies. Finally, the reference-based methods achieve worse results, especially when they do not retrieve all haplotypes. The supplementary material contains individual estimations per haplotype and dataset for further details about which haplotypes have been retrieved and the error in each estimation.

4.5 Time performance and memory consumption

We measured the running time and peak memory usage required by the seven tools. For evaluation, we used only one core to execute the different techniques. For VG-Flow, Virus-VG, PredictHaplo and CliqueSNV, elapsed time and memory consumption are split into two terms separated by "+" and "—" symbols, respectively. For VG-Flow and Virus-VG, the first time term refers to the time required to build the preassembled contigs that the tools need, while the second term is the actual time for running the tool. The reference-based methods, PredictHaplo and CliqueSNV, require an alignment file; thus, the first term refers to the time employed to perform the alignment process. We used SAVAGE [8] to obtain the preassembled contigs as suggested by the authors of VG-Flow and Virus-VG. Additionally, we used BWA [34] to obtain the alignment files for reference-based methods. For running time, we used the symbol "+" to report the results since the entire running time is the sum of the time required by SAVAGE or BWA and the one required by a specific assembly tool. We use the symbol "—" for memory consumption, since peak memory consumption of the entire process is the maximum of both results. We decided to show the information separately because one could use another multiassembly, such as viaDBG or ViQUF, or alignment tool, such as Bowtie or Bowtie2, for this first step, which might be beneficial in terms of running times or memory consumption.

First, running times for *de novo* methods in Table 2 show that ViQUF is the fastest approach, faster than viaDBG, which is also a DBG-based method, and PEHaplo, which (although it is an overlap-based method) can correctly reduce the input data several orders of magnitude. Finally, it is significantly faster than Virus-VG and VG-Flow, based on overlap graphs. ViQUF is between 275 and 1070 times faster than SAVAGE, followed by Virus-VG, and between 275 and 1354 times than SAVAGE, followed by VG-Flow.

These results were expected since the DBG-based methods skip reads alignment, which takes quadratic time. Furthermore, ViQUF and viaDBG are faster because VG-Flow and Virus-VG require aligning reads to contigs (which takes $O(nm)$, where m is the number of contigs and n is

the number of reads) even without considering SAVAGE execution time, thus removing reads alignment. Compared with viaDBG and PEHaplo, ViQUF is between 5 and 80 times faster. The current implementation of ViQUF is not parallelized. The most time-consuming part of ViQUF is building APAG, which could, in principle, be computed in parallel for each pair of adjacent nodes. However, the practical impact of parallelization would be negligible since even without parallelization, ViQUF finishes in less than 5 minutes on all datasets.

For the reference-based methods, running times include the time required for alignment, even though it is negligible compared with the actual assembly time. Running times for reference-based methods are larger than those for the DBG-based methods; specifically two to four times slower than viaDBG and 10 to 40 times slower than ViQUF. These are significantly smaller than the times of Virus-VG and VG-Flow, and they are comparable with those of PEHaplo.

Finally, our entire approach is more efficient in terms of memory consumption since SAVAGE+Virus-VG and SAVAGE+Virus-Flow require 12 to 25 times the space required by ViQUF, and ViQUF requires less than half of the memory required by viaDBG and PEHaplo. However, the two reference-based methods show different behavior. While PredictHaplo is comparable to ViQUF, even outperforming its results for HIV-real, CliqueSNV requires from eight to 10 times more memory than PredictHaplo and ViQUF.

4.6 Testing *de novo* approaches boundaries

The experiments shown so far reveal an advantage of *de novo* methods against the reference-based ones. However, all datasets used in those experiments have relatively high divergence ratio between the haplotypes. Therefore, the previous sections do not include a scenario where the viruses have not evolved a lot from the common ancestor.

To test this scenario and, at the same time, test the *de novo* algorithms boundaries, we used FAVITES [35] and DWGSIM to simulate two different datasets with five haplotypes with low divergence ratios, from 0.1% to 2.0%, and an average divergence of 1.08%. Furthermore, we have established two different abundance levels to test two different situations. In the first case we have the same abundance, 20%, for each simulated haplotype and a coverage of 20000×. In the second case, we have a more realistic situation with different abundances, 33.3%, 27.6%, 20%, 12.5%,5%, and 2.5% but with a very high coverage of around 40000×. Through these experiments, our goal is to test the limits of *de novo* and reference-based approaches in terms of sensitivity and scalability.

Table 3 summarizes the results of these experiments. In the first dataset, where abundance for each strain is the same, the reference-based methods, PredictHaplo and CliqueSNV, slightly outperform the *de novo* approaches. The percentage of genome retrieved is almost the same, except for CliqueSNV, which is able to obtain the five entire haplotypes with almost no error. However, mismatches here are really important, for example, PEHaplo's 0.8 might be really problematic when maximum difference between strains is around 2%, in the case of ViQUF this number is smaller, 0.314, but it is still higher than that of the reference-based methods. In the second dataset, both reference and *de novo* based methods retrieve the entire set of haplotypes with a N50 as long as the actual genomes. However, in this case, there is no advantage of using reference-based methods because ViQUF retrieves the same percentage of the genome with no errors, as PredictHaplo. For the second dataset, we were not able to get results for SAVAGE+VG-Flow and CliqueSNV. SAVAGE could not produce long enough contigs in any of the experiments, whereas CliqueSNV requires more than 64Gb and thus, we could not run it on the same machine we used for the rest of the experiments. Table 3 also shows the time needed to run all the experiments. While the first dataset has the same coverage as the ones used in previous sections, the second has twice the coverage. This second dataset was simulated in this way to be able to measure the scalability of the approaches. As the results show, with the exception of CliqueSNV, the methods are scalable. PredictHaplo has the best scalability ratio, whereas ViQUF is the one with the lowest absolute runtimes.

Overall, the results exposed in Table 3 show that reference-based methods are slightly more reliable than *de novo* approaches when the divergence between the strains is very low, specially CliqueSNV, which has an astonishing performance. However, *de novo* approaches are on average faster and use less memory than the reference-based methods, even if we do not take into consideration the required time for the alignment.

5 Conclusions

In this work, we proposed ViQUF to *de novo* assemble viral quasispecies. The methodology of ViQUF bears some similarity to our previous work, viaDBG [12], which also uses paired-end information to split nodes of a DBG into haplotypes. However, ViQUF addresses the problem differently, obtaining important result improvements and the possibility of estimating haplotype abundances.

With respect to ViaDBG, these are the main differences. First, it employs a mathematically rigorous way of determining solid k-mers using kernel density estimation, resulting in better estimates for the abundance threshold of solid k-mers as shown by our experiments. Second, ViQUF uses path cover to split the adjacent nodes into haplotypes, whereas viaDBG finds cliques in the reachability graph of paired-end nodes. Third, ViQUF uses path cover to find haplotypes and their abundances; thus, considering the coverage in this stage. However, viaDBG only reports nonbranching paths as contigs. Our experiments show that these improvements allow us to build longer contigs with comparable accuracy fast.

In the overall results, all methods show remarkable performance. In comparison, Virus-VG, VG-Flow,and the two reference-based methods (PredictHaplo and CliqueSNV) perform slightly better in average contig length than their counterparts. However, the error rate is higher than expected for some cases, and the reference-based methods lose some haplotypes. This is probably because of the high divergence between the haplotypes in each sample. However, viaDBG and ViQUF exhibit a more

TABLE 3
Results for the viral quasispecies assembly tools, de novo and reference-based, for small divergence ratios and extremely high sequencing coverage. For reference-based methods, the elapsed time column shows the time required for alignment plus the tool execution time.

		% Genome	N50	Misassemblies	Mismatches	Time elapsed (mins)
Same abundance - 20000× coverage	ViQUF	83.97%	9975	0	0.314	4.53
	SAVAGE+VG-Flow	*	*	*	*	*
	PEHaplo	80.00%	10000	0	0.807	227.96
	PredictHaplo	80.00%	10000	0	0.205	10.53 + 442.78
	CliqueSNV	100.00%	10000	0	0.100	10.53 + 34.93
Different abundances - 50000× coverage	ViQUF	99.94%	9995	0	0.000	25.56
	SAVAGE+VG-Flow	*	*	*	*	*
	PEHaplo	100.00%	10000	0	0.604	1051.56
	PredictHaplo	100.00%	10000	0	0.100	23.73 + 1475.23
	CliqueSNV	*	*	*	*	*

robust behavior without an unexpectedly high error rate for any dataset. Furthermore, ViQUF shows high levels of sensitivity, retrieving more than 90% of the genome fraction. ViQUF outperforms all methods when comparing memory consumption and runtime, especially VG-Flow and Virus-VG. It is also remarkable that PredictHaplo exhibits a great performance in running time and memory usage, especially in memory usage, where it is comparable with ViQUF and even better for real data. For this dataset, PEHaplo achieves the best performance in terms of genome fraction and error rate. ViQUF is also faster than viaDBG, using noticeably less memory. Moreover, ViQUF can provide haplotype abundance, whereas viaDBG cannot.

In summary, ViQUF obtains long accurate contigs consuming fewer resources than other methods. Furthermore, ViQUF can produce competitive haplotype frequency estimations compared with current state-of-the-art tools like Virus-VG and VG-Flow. However, our experiments were mostly conducted on synthetic data. Therefore, it is highly desirable to obtain new real datasets to conduct more thorough experimental evaluations in our future studies.

As a comparative conclusion, the viral quasispecies assembly and quantification have several good but different solutions. Some solutions are better than others depending on constraints, such as runtime, memory resources, accuracy required, and properties of a specific dataset. For example, when the sequencing depth is sufficiently high and uniform over each haplotype, and all haplotypes are completely present in the sample, ViQUF or viaDBG can be the best solution when running time and memory consumption are severe constraints. However, under the same conditions but without constraints on running time and memory, using a smart grid of parameters with SAVAGE+VG-Flow could lead to the most accurate results. Furthermore, if the average similarity between haplotypes is below 1%, the most accurate and consistent results are provided by reference-based methods.

ACKNOWLEDGMENTS

This work has received funding from the EU H2020 under the Marie Sklodowska-Curie [GA 690941]. We wish to acknowledge the support received from the Centro de Investigación de Galicia "CITIC", funded by Xunta de Galicia and the European Union (European Regional Development Fund- Galicia 2014-2020 Program), by grant ED431G 2019/01. This work was also supported by Xunta de Galicia/FEDER-UE under Grants [ED431C 2021/53; IG240.2020.1.185; IN852A 2018/14]; Ministerio de Ciencia e Innovación under Grants [TIN2016-78011-C4-1-R; FPU17/02742; PID2019-105221RB-C41; PID2020-114635RB-I00]; and the Academy of Finland [grants 308030 and 323233 (LS)]. The authors also thank David Posada for his advice on viral evolution.

REFERENCES

[1] E. Domingo, J. Sheldon, and C. Perales, "Viral quasispecies evolution," *Microbiology and Molecular Biology Reviews*, vol. 76, no. 2, pp. 159–216, 2012.
[2] E. C. Holmes, *The Evolution and Emergence of RNA Viruses*. Oxford University Press, 2009.
[3] S. Duffy, L. A. Shackelton, and E. C. Holmes, "Rates of evolutionary change in viruses: patterns and determinants," *Nature Reviews Genetics*, vol. 9, pp. 267–276, 2008.
[4] O. Zagordi, A. Bhattacharya, N. Eriksson, and N. Beerenwinkel, "ShoRAH: estimating the genetic diversity of a mixed sample from next-generation sequencing data," *BMC Bioinformatics*, vol. 12, p. 119, 2011.
[5] S. Prabhakaran, M. Rey, O. Zagordi, N. Beerenwinkel, and V. Roth, "HIV haplotype inference using a propagating Dirichlet process mixture model," *IEEE/ACM Transactions on Computational Biology and Bioinformatics*, vol. 11, no. 1, pp. 182–191, 2014.
[6] S. Knyazev, V. Tsyvina, A. Melnyk, A. Artyomenko, T. Malygina, Y. B. Porozov, E. Campbell, W. M. Switzer, P. Skums, and A. Zelikovsky, "CliqueSNV: scalable reconstruction of intra-host viral populations from ngs reads," *bioRxiv*, 2019.
[7] R. Malhotra, S. Prabhakara, M. Poss, and R. Acharya, "Estimating viral haplotypes in a population using k-mer counting," in *Pattern Recognition in Bioinformatics*, A. Ngom, E. Formenti, J.-K. Hao, X.-M. Zhao, and T. van Laarhoven, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 265–276.
[8] J. A. Baaijens, A. Z. E. Aabidine, E. Rivals, and A. Schönhuth, "De novo assembly of viral quasispecies using overlap graphs," *Genome Research*, vol. 27, pp. 835–848, 2017.
[9] N.-C. Chen, B. Solomon, T. Mun, S. Iyer, and B. Langmead, "Reference flow: reducing reference bias using multiple population genomes," *Genome biology*, vol. 22, no. 1, pp. 1–17, 2021.
[10] J. Chen, Y. Zhao, and Y. Sun, "*De novo* haplotype reconstruction in viral quasispecies using paired-end read guided path finding," *Bioinformatics*, vol. 34, no. 17, pp. 2927–2935, 2018.
[11] R. Malhotra, M. M. S. Wu, A. Rodrigo, M. Poss, and R. Acharya, "Maximum likelihood de novo reconstruction of viral populations using paired end sequencing data," *arXiv e-prints*, p. arXiv:1502.04239, 2015.
[12] B. Freire, S. Ladra, J. Paramá, and L. Salmela, "Inference of viral quasispecies with a paired de Bruijn graph," *Bioinformatics*, vol. 37, no. 4, pp. 473–481, 2021.
[13] P. Medvedev, S. Pham, M. Chaisson, G. Tesler, and P. Pevzner, "Paired de Bruijn graphs: a novel approach for incorporating mate pair information into genome assemblers," *Journal of Computational Biology*, vol. 18, no. 11, pp. 1625–1634, 2011.

[14] P. A. Pevzner, H. Tang, and M. S. Waterman, "An eulerian path approach to dna fragment assembly," *Proceedings of the National Academy of Sciences*, vol. 98, no. 17, pp. 9748–9753, 2001.
[15] E. W. Myers, "The fragment assembly string graph," *Bioinformatics*, vol. 21, no. suppl_2, pp. ii79–ii85, 09 2005.
[16] A. I. Tomescu, A. Kuosmanen, R. Rizzi, and V. Mäkinen, "A novel min-cost flow method for estimating transcript expression with RNA-Seq," *BMC Bioinformatics*, vol. 14, p. S15, 2013.
[17] K. Westbrooks, I. Astrovskaya, D. Campo, Y. Khudyakov, P. Berman, and A. Zelikovsky, "Hcv quasispecies assembly using network flows," in *Bioinformatics Research and Applications*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 159–170.
[18] P. Skums, N. Mancuso, A. Artyomenko, B. Tork, I. Mandoiu, Y. Khudyakov, and A. Zelikovsky, "Reconstruction of viral population structure from next-generation sequencing data using multicommodity flows," *BMC Bioinformatics*, vol. 14, no. 9, pp. 1–13, 2013.
[19] J. A. Baaijens, B. Van der Roest, J. Köster, L. Stougie, and A. Schönhuth, "Full-length de novo viral quasispecies assembly through variation graph construction," *Bioinformatics*, 05 2019, btz443.
[20] N. Mancuso, "Algorithms for viral population analysis." Ph.D. dissertation, Georgia State University, 2014.
[21] I. Astrovskaya, N. Mancuso, B. Tork, S. Mangul, A. Artyomenko, P. Skums, L. Ganova-Raeva, I. Mandoiu, A. Zelikovsky, and M. Park, "Inferring viral quasispecies spectra from shortgun and aplicon next-generation sequencing reads," in *Genome analysis: current procedures and applications*, M. S. Poptsova, Ed. Caister Academic Pres, 2014.
[22] M. J. Chaisson and P. A. Pevzner, "Short read fragment assembly of bacterial genomes," *Genome Research*, vol. 18, no. 2, pp. 324–330, 2008.
[23] M. J. Chaisson, D. Brinza, and P. A. Pevzner, "De novo fragment assembly with short mate-paired reads: Does the read length matter?" *Genome Research*, vol. 19, no. 2, pp. 336–346, 2009.
[24] X. Zhao, L. E. Palmer, R. Bolanos, C. Mircean, D. Fasulo, and G. M. Wittenberg, "Edar: an efficient error detection and removal algorithm for next generation sequencing data," *Journal of computational biology*, vol. 17, no. 11, pp. 1549–1560, 2010.
[25] R. Chikhi, A. Limasset, and P. Medvedev, "Compacting de Bruijn graphs from sequencing data quickly and in low memory," *Bioinformatics*, vol. 32, no. 12, pp. i201–i208, 06 2016.
[26] S. Nurk, D. Meleshko, A. Korobeynikov, and P. Pevzner, "metaSPAdes: a new versatile metagenomic assembler," *Genome Research*, vol. 27, pp. 824–834, 2017.
[27] E. Drezen, G. Rizk, R. Chikhi, C. Deltel, C. Lemaitre, P. Peterlongo, and D. Lavenier, "GATB: genome assembly & analysis tool box," *Bioinformatics*, vol. 30, no. 20, pp. 2959–2961, 2014.
[28] A. Tomescu, V. Mäkinen, F. Cunial, and D. Belazzougui, "Genome-scale algorithm design," 2015.
[29] J. A. Baaijens, L. Stougie, and A. Schönhuth, "Strain-aware assembly of genomes from mixed samples using flow variation graphs," *bioRxiv*, 2020.
[30] F. D. Giallonardo, A. Töpfer, M. Rey, S. Prabhakaran, Y. Duport, C. Leemann, S. Schmutz, N. K. Campbell, B. Joos, M. R. Lecca, A. Patrignani, M. Däumler, C. Beisel, P. Rusert, A. Trkola, H. F. Günthard, V. Roth, N. Beerenwinkel, and K. J. Metzner, "Full-length haplotype reconstruction to infer the structure of heterogeneous virus populations," *Nucleic Acids Research*, vol. 42, no. 14, p. e115, 2014.
[31] N. Stoler and A. Nekrutenko, "Sequencing error profiles of Illumina sequencing instruments," *NAR Genomics and Bioinformatics*, vol. 3, no. 1, 03 2021.
[32] A. Mikheenko, V. Saveliev, and A. Gurevich, "MetaQUAST: evaluation of metagenome assemblies," *Bioinformatics*, vol. 32, no. 7, pp. 1088–1090, 2016.
[33] A. Eliseev, K. M. Gibson, P. Avdeyev, D. Novik, M. L. Bendall, M. Pérez-Losada, N. Alexeev, and K. A. Crandall, "Evaluation of haplotype callers for next-generation sequencing of viruses," *Infection, genetics and evolution: journal of molecular epidemiology and evolutionary genetics in infectious diseases*, vol. 82, p. 104277, August 2020.
[34] H. Li and R. Durbin, "Fast and accurate long-read alignment with Burrows-Wheeler transform." *Bioinformatics*, vol. 26, no. 5, pp. 589–595, 2010.
[35] N. Moshiri, M. Ragonnet-Cronin, J. O. Wertheim, and S. Mirarab, "FAVITES: simultaneous simulation of transmission networks, phylogenetic trees and sequences," *Bioinformatics*, vol. 35, no. 11, pp. 1852–1861, 11 2018.

Borja Freire received his bachelor degree in Computer Science at the University of A Coruña in 2016 and master degree in Bioinformatics at the same university in 2018. He is now a PhD student of the Doctorate Program in Computer Science at University of A Coruña, and he has been awarded a FPU fellowship to complete his doctorate.

Susana Ladra received the bachelor's degree in mathematics from the National Distance Education University (UNED), in 2014, and the master's in computer science engineering and the Ph.D. degree in computer science from the University of A Coruña, in 2007 and 2011, respectively. She is currently an Associate Professor with the Universidade da Coruña. She is the Principal Investigator of several national and international research projects. She has published more than 40 articles in various international journals and conferences. Her research interests include design and analysis of algorithms and data structures, and data compression and data mining in the fields of information retrieval and bioinformatics.

José R. Paramá has PhD in computer science from the University of A Coruña. His PhD Thesis was presented in 2001 and had research results also developed during stays at New Mexico State University and Texas Tech University. Since 1997 he is a professor at the University of A Coruña, and since 2008, Associate Professor. He has participated in more than twenty research projects funded by European, regional and national administrations, and in more than thirty R&D contracts.

He is the author of more than thirty scientific publications and more than sixty scientific conferences.

Leena Salmela received her M.Sc.(Tech) and D.Sc.(Tech) degrees in computer science from Helsinki University of Technology, Finland, in 2005 and 2009, respectively. Since 2009 she has worked in the University of Helsinki, Finland, where she currently is University Lecturer and Academy of Finland Research Fellow. She has published over 40 scientific articles in international journals and conferences. Her research field is algorithmic bioinformatics with a focus on applications in biological sequence analysis.

Supplementary Material

CONTENTS

1 BACKGROUND

In this section, we introduce some basic concepts used by our method, such as the network flow problem and kernel density estimation.

1.1 Min-cost network flow problem

A flow network is a tuple $N = (G, b, q)$, where $G = (V, E)$ is a directed graph, b is a function assigning a *capacity* b_{uv} to every arc $(u, v) \in E$, and q is a function assigning an exogenous flow $q_v \in \mathbb{Z}$ to every node $v \in V$, such that $\sum_{v \in V} q_v = 0$. Then, we have a flow over the network N, if for every arc $(u, v) \in E$ the flow $x_{uv} \in \mathbb{N}$ satisfies two conditions:

1) $0 \leq x_{uv} \leq b_{uv}$ for every $(u, v) \in E$
2) $\sum_{u \in V} x_{vu} - \sum_{u \in V} x_{uv} = q_v$, for every $v \in V$

In the min-cost flow problems, like in many other cost-flow optimization problems, a cost function $c_{uv}(x)$ is given for every $(u, v) \in E$, and the objective is to find the flow which minimizes:

$$\sum_{(u,v) \in E} c_{uv}(x_{uv}).$$

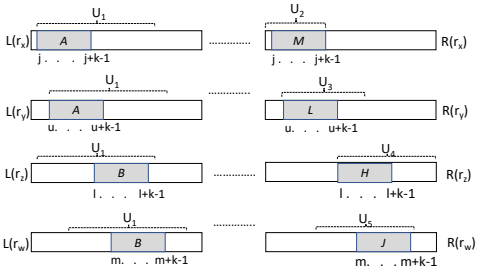


Fig. 1. Extracting paired unitigs from paired-end reads $P(U_1) = \{U_2, U_3, U_4, U_5\}$.

This problem can be solved in polynomial time when $c_{uv}(x)$ is a convex cost function. One of the most common cost functions is $c_{uv}(x) = x^2$.

[1] proposed a min-cost flow method for estimating transcript expression with RNA-Seq. We will use a similar approach as part of our method.

1.2 Kernel density estimation

Kernel density estimation is a very powerful mathematical tool for estimating the probability density function of a random variable. In practice, it creates a smooth curve given a set of data. More concretely, it can be expressed as:

$$\hat{f}_h(x) = \frac{1}{n} \sum_{i=1}^{n} K_h(x - x_i)$$

$$K_h(y) = \frac{K(y)}{h}$$

where x_i are the sample data points, n is the sample size, h is the window size or the bandwidth, and $K(\cdot)$ is a kernel function, such as Gaussian, uniform, or Epanechnikov, among others [2]. Once $\hat{f}_h$ is estimated, the minima and maxima of $\hat{f}_h$ are obtained by computing the zero crosses of its derivative.

2 EXAMPLE OF PAIRED UNITIGS

In Figure 1, the unitig U_1 contains (among others) the k-mers A and B. Observe that $L(r_x)$ contains the k-mer A at positions $j \ldots j + k - 1$, and in those positions of $R(r_x)$ we find the k-mer M. In r_y, A is at positions $u \ldots u + k - 1$ of $L(r_y)$, whereas positions $u \ldots u + k - 1$ of $R(r_y)$ contain the k-mer L. Therefore, $P(A) = \{M, L\}$. From the reads r_z and r_w, we obtain that $P(B) = \{H, J\}$. Since M is located

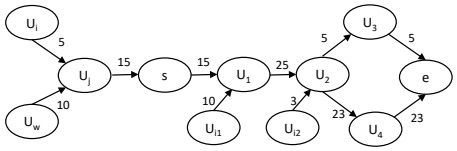


Fig. 2. Extract of an AG used to compute the coverage of the edge between s and e in a DAG DAG_{ij}.

inside U_2, that is $\mathcal{U}(M) = U_2$, and $\mathcal{U}(L) = U_3$, $\mathcal{U}(H) = U_4$, and $\mathcal{U}(J) = U_5$, then finally $P(U_1) = \{U_2, U_3, U_4, U_5\}$.

3 Coverage of an edge of the DAG

To split the nodes of the assembly graph into haplotypes, we build DAG_{ij} for each pair of adjacent nodes U_i and U_j in the assembly graph. The nodes of DAG_{ij} are the pair of adjacent nodes and the nodes pointed by their paired-end information. We add an edge between two nodes if there exists a path between the corresponding nodes in the assembly graph. Here, we present the method to compute the coverage of an edge (s, e) of a DAG_{ij}.

3.1 Initial coverage estimation

DAG_{ij} is constructed to analyze haplotypes that pass through U_i and U_j. Therefore when computing the coverage of an edge (s, e) in DAG_{ij}, we should only consider haplotypes that pass through all nodes U_i, U_j, s, and e. *The coverage of an edge* (s, e) *in* DAG_{ij} is thus defined as the total coverage of all haplotypes that pass through the nodes U_i, U_j, s, and e. For this, we analyze the coverages of the paths connecting U_i, U_j, s, and e *in the AG*. The coverage of an edge of the AG is the number of reads supporting that edge.

To illustrate the process, in Figure 2, we can see an extract of an AG. Observe in the figure that the coverage of the edge (s, U_1) is 15, but that coverage cannot be used as that of $cov(s, e)$ in the DAG, since it includes 10 reads that are actually reads corresponding to the haplotype that passes through U_w and U_j but not through U_i, and therefore the coverage corresponding to that haplotype must not be included in the coverage of (s, e) in DAG_{ij}, which is computed for detecting the haplotypes that pass through U_i and U_j. The mixture of coverages coming from different haplotypes in the paths from s to e makes it impossible to use simple estimators such as maximum, minimum, or mean of the coverages of edges on the path from s to e.

Given a node U_k of the AG, let $N^+_{U_k}$ be the list of out-neighbors of U_k, $N^-_{U_k}$ be the list of in-neighbors, and $cN^+_{U_k}$ and $cN^-_{U_k}$ the lists of the coverages of the corresponding edges. For example, in our example of Figure 2, $N^+_{U_2} = \{U_3, U_4\}$ and $N^-_{U_2} = \{U_1, U_{i2}\}$, and $cN^+_{U_2} = \{23, 5\}$ and $cN^-_{U_2} = \{25, 3\}$.

We are going to traverse the AG through all paths connecting U_i, U_j, s, and e, analyzing cN^+ and cN^- in each node, in order to derive a reliable coverage. During a traversal of the AG that has already processed the path $p = \{U_i, U_j, \ldots, s, U_1, \ldots, U_k\}$, roughly, we maintain a bag B_k containing the coverages of edges reaching nodes of p and coming from nodes that are not in p, thus they correspond to haplotypes different from the one we are following. Therefore, when e is reached, the coverage of that path is the coverage of the edge reaching e minus the coverages in the bag. However, on our way towards e, we will find outgoing edges leading to paths that do not reach e, so the coverage of those edges should be removed from the bag.

Therefore, when we process a new node with one or more outgoing edges leading to a node which is not on our way to e, we need a method to distribute the incoming coverages stored in the bag among those outgoing nodes. A first naive idea is to search for coverages in the bag matching the coverages of outgoing edges corresponding to paths that do not reach e. We remove those values from the bag, assuming that those values correspond to haplotypes that entered in p in a previous node and now, we found a node where those haplotypes are leaving the path we are following towards e. The problem with this approach is that each incoming edge may correspond to several haplotypes, and therefore they might leave our path p in different points, and thus this approach is not capable of dealing with this situation.

Therefore, we need a more complex method. As explained, the method is based on traversing the AG from U_i until e through all possible paths, carrying a bag with the incoming coverages. When a traversal finds a fork, it distributes those coverages among the nodes in the fork, by using the following iterative minimization problem.

- Let $p_k = \{U_i, \ldots, U_k\}$ be path of the AG followed so far.
- Let $cov(p_k)$ the coverage estimation of the path p_k.
- Let $iC = \{ic_1, ..., ic_n, cov(p_k)\}$ be the coverages of n incoming edges to p_k with the coverage of p_k included as well.
- Let $oC = \{oc_1, ..., oc_b\}$ be the coverages of the b outgoing edges from U_k.

The task is then to assign the incoming coverages to the outgoing coverages so that they correspond to each other as well as possible while simultaneously avoiding to split the incoming coverages, i.e. assigning them partially to several outgoing coverages.

For each $v \in \{1...n\}$ and $u \in \{1...b\}$, we define a variable $x_{vu} \in \{0, 1\}$ which indicates whether the incoming coverage ic_v is assigned to the outgoing coverage oc_u. In the first iteration, $iC = B_k \cup cov(p_k)$, where B_k is the bag of incoming coverages on path p_k and $cov(p_k)$ the coverage estimate of the traversed path, and $oC = cN^+_{U_k}$. In a single iteration, we assign one incoming coverage completely to an outgoing coverage, whereas several incoming coverages can be assigned to the same outgoing coverage. Because $cov(p_k)$ has been added to the set of incoming coverages, in each iteration it can only be assigned to one outgoing edge. However, we allow the coverage of the path $cov(p_k)$ to be split by including the remaining of it in successive iterations. Furthermore, we want to minimize the difference between the assigned incoming coverages and the outgoing coverage

of each edge. More formally, the minimization problem is defined as follows:

$$\begin{aligned} \min \quad & \sum_{u=1}^{b} \left| oc_u - \sum_{v=1}^{n} (ic_v x_{vu}) \right| \\ \text{subject to:} \quad & \sum_{u=1}^{b} x_{vu} = 1, \forall v \in \{1...n\} \\ & x_{vu} \in \{0,1\}, \forall vu \in \{1...n\} \times \{1...b\} \end{aligned} \tag{1}$$

The constraints make sure that one incoming coverage is assigned completely to a single outgoing coverage.

Let iteration i be defined by oC_i, the list of outgoing coverages still not completely used by the assigned incoming coverages, and iC_i, the remaining incoming coverages available. Then iteration $i+1$ is defined as follows. We subtract from each outgoing coverage the coverage of the incoming edges assigned to that outgoing edge:

$$oC_{i+1} = \{\max(oc_u - \sum_{v=1}^{n} ic_v x_{vu}, 0) \quad \forall u \in \{1...b\}\} \tag{2}$$

Note that if the sum of assigned incoming coverages exceeds the outgoing coverage of the edge, we set the outgoing coverage to zero for the next iteration. Similarly, we will subtract from the incoming coverages the coverage of the edge they have been assigned to. However, more than one incoming coverage can be assigned to the same outgoing coverage. If the sum of the assigned incoming coverages exceeds the outgoing coverage, we will decrease the incoming coverages starting from the largest incoming coverage. Assuming that ic_v are sorted in ascending order, we thus get:

$$\begin{aligned} iC_{i+1} = \{\max(ic_v - \max(oc_u - \sum_{v'=v+1}^{n} x_{v'u} ic_{v'}, 0), 0)\} \\ \text{where } u \text{ is such that } x_{vu} = 1 \quad \forall v \in \{1...n\}\} \end{aligned} \tag{3}$$

The iterative process ends when either all incoming coverages have been assigned to outgoing coverages or all outgoing coverages have been totally used. When the iterative process ends, the new estimation of the coverage of the new extended path ($cov(p_{k+1})$) is obtained by subtracting from the coverage of the assembly graph edge (U_k, U_{k+1}) the sum of incoming coverages in B_k assigned to the branch of U_{k+1}. These assigned coverages form the new bag B_{k+1}. As a reminder, B_{k+1} does not contain the fraction of $cov(p_k)$ assigned to the branch; notice that the inclusion of $cov(p_k)$ is only for minimization problem purposes.

This approach not only allows the incoming coverage to be distributed among several outgoing edges, but also allows estimating the coverage of the haplotype. In this way it is possible to avoid the traversal of paths without assigned coverage, speeding up the build process of the DAGs and leading to less tangled DAGs.

There can be several paths passing through U_i, U_j, s, and e and often these paths share a prefix. Next we will explain how to traverse all these paths to compute the coverages. The process is as follows: let $\mathcal{T}$ be a list of traversals of the AG. Each traversal is a path of the AG that starts at U_i, passes through U_j, and contains the nodes processed so far on its way through s towards e.

Initially $\mathcal{T}$ contains one traversal t_j, its path only contains U_iU_j, and its coverage ($cov(t_j)$) is the coverage of the edge (U_i, U_j). Each traversal has a bag, initially, the bag of t_j is $B_j = cN^-_{U_j}$.

Then we will repeatedly remove one traversal from $\mathcal{T}$ until it is empty, compute the extensions of the traversal, and insert them into $\mathcal{T}$ unless they have reached e, their coverage has dropped to zero, or they are not supported by the paired-end information[1]. The extensions of a traversal t_λ are computed as follows. Suppose $p_\lambda = \{U_i, ...U_k\}$ is the path followed by traversal t_λ so far and B_k is the bag of incoming coverages associated with the traversal. We create a new traversal for each node $U_{k+1} \in N^+_k$ by concatenating p_λ with U_{k+1}. If U_k has more than one outgoing edge, we use the minimization problem above to distribute the incoming coverages B_k appropriately and to compute the incoming coverages for the extended traversal and the coverage of the extended traversal. If U_{k+1} has more than one incoming edge, the coverages in $cN^-_{U_{k+1}}$ except for the coverage of the edge (U_k, U_{k+1}) are added to B_k to form B_{k+1}. If the coverage of the extended traversal drops to zero or it contradicts the paired-end information of p_λ, we stop the traversal and do not add it to $\mathcal{T}$. Finally, the coverage of the edge (s, e) in DAG_{ij} is the sum of coverages of all traversals that reached e.

To illustrate the process, we are going to compute the coverage of the edge connecting s and e in the DAG DAG_{ij} for the AG shown in Figure 2.

Table 1 shows in row number 1 the initial state of $\mathcal{T}$ with only one traversal t_{ij}, its bag $B_j = cN^-_{U_j} = \{10\}$, and the initial coverage estimation ($cov(t_{ij})$) is set to $cov(U_i, U_j)$.

Row 2 processes s, which since it only has one incoming and one outgoing edge does not produce any change. Row 3 processes U_1, that, in addition to the edge coming from s, has also an incoming edge coming from U_{i1}, and then, the coverage (10) of that edge is added to the bag B_1.

Row 4 processes the out neighbors of U_1, this adds only U_2. Then, the coverage of the edge coming from U_{i2} is added to the bag, that is $B_2 = \{10, 10, 3\}$.

TABLE 1
Trace of the computation of the coverage of the edge (s, e) of DAG_{ij}.

#	$\mathcal{T}$	U_k	B_{k+1}	cov
1	t_{ij}	U_i	$B_j = \{10\}$	5
2	t_{ijs}	U_j	$B_s = \{10\}$	5
3	t_{ijs1}	s	$B_1 = \{10, 10\}$	5
4	t_{ijs12}	U_1	$B_2 = \{10, 10, 3\}$	5
5	t_{ijs123}	U_2	$B_3 = \{\}$ $B_4 = \{10, 10, 3\}$	$cov(t_{s123}) = 5$ $cov(t_{s124}) = 0$
6	$t_{ijs123e}$	U_3	$B_e = \{\}$	$\text{cov}(t_{ijs123e}) = 5$

Row 5 shows the processing of the outgoing neighbors of U_2. This extracts t_{ijs12} from $\mathcal{T}$, and since U_2 has two outgoing neighbors (U_3 and U_4), two traversals (t_{ijs123} and t_{ijs124}) are created.

1. That is, the new node to be added should be in the paired-end information of at least one node in the path traversed so far.

- We formulate the minimization problem of Equation (1) taking $iC = B_2 \cup \{cov(t_{ijs12}]\} = \{10, 10, 3, 5\}$ and $oC = cN^+_{U_2} = \{23, 5\}$

$$\begin{aligned} \min \quad & |23 - 10x_{11} - 3x_{21} - 10x_{31} - 5x_{41}| \\ & + |5 - 10x_{12} - 3x_{22} - 10x_{32} - 5x_{42}| \\ \text{subject to:} \quad & x_{11} + x_{12} = 1, \\ & x_{21} + x_{22} = 1, \\ & x_{31} + x_{32} = 1, \\ & x_{41} + x_{42} = 1, \\ x_{11}, x_{12}, x_{21}, x_{22}, x_{31}, x_{32}, x_{41}, x_{42} & \in \{0, 1\} \end{aligned} \tag{4}$$

- An **optimal possible** solution is $x_{11} = 1$, $x_{21} = 1$,$x_{31} = 1$, $x_{12} = 0$, $x_{22} = 0$, $x_{32} = 0$, $x_{41} = 0$ and $x_{42} = 1$. With those values, expression (4) returns zero.
- That solution assigns all the excess in the bag to the outgoing edge that goes to U_4 and thus: $B_3 = \{10{\cdot}0, 3{\cdot}0, 10{\cdot}0\} = \{\emptyset\}$ and $B_4 = \{10{\cdot}1, 3{\cdot}1, 10{\cdot}1\} = \{10, 10, 3\}$.
  Both the set of incoming and outgoing coverages become empty sets for the next iteration and thus the minimization ends after only one iteration.
- Therefore, the coverages of the two traversals are: leftmargin=+.2cm
  - $cov(t_{s123}) = cov(U_2, U_3) - \sum_{v=1}^{3} ic_v x_{v2} = 5 - (0 + 0 + 0) = 5$. Thus t_{s123} is added to $\mathcal{T}$.
  - $cov(t_{s124}) = cov(U_2, U_4) - \sum_{v=1}^{3} ic_v x_{v1} = 23 - (10 + 3 + 10) = 0$. Therefore, t_{s124} is not added to $\mathcal{T}$.

Therefore, in this example, it is clear that all the input excesses found in the path through $U_i, U_j, s, U_1, U_2, U_3$ can be assigned to the path that goes through U_4, and thus the path that goes through U_3 continues without any excess in its bag, whereas that passing through U_4 is discarded.

Row 6 processes t_{ijs123} appending the target e. Therefore, the coverage of the edge (U_3, e) (5) is the coverage of the traversal passing through U_3 ($cov(U_3, e) = 5$), since the bag B_3 is empty.

Once all traversals reached e, we obtain the coverage of the edge (s, e) in DAG_{ij} as the sum of the coverages of the traversals that reached e, in our example, it is only $cov(t_{ijs123e}) = 5$.

3.2 Coverage readjustment

The previous section explains how to assign coverages to the edges of DAG_{ij}. However, there are some situations where this procedure generates unreliable variations of the coverages, building peaks and falls across the graph . These nodes will create an incomprehensible exogenous flow that leads to misleading flows. Therefore, they have to be adjusted whenever possible.

We say that a node U_r of DAG_{ij} is *out of coverage* when $\sum cN^-_{U_r} \leq (1 + RATIO) \sum cN^+_{U_r}$ or vice versa, where $RATIO$'s default value is 0.1 namely a node is *out of coverage* when the incoming or outgoing coverages is 10% higher than the other.

We are going to correct these *out of coverage* nodes in the DAG_{ij}. For this, the DAG is traversed in breadth-first manner making sure that all nodes in $N^-_{U_r}$ have been adjusted before adjusting U_r. When a node U_r is classified as *out of coverage*, then a solution is found based on its *in-degree* and *out-degree*. Three situations arise:

- *in-degree* ≥ 1 and *out-degree* = 1, these cases can be easily solved by setting $cov(U_r, U_{out}) = \sum cN^-_{U_r}$, where U_{out} is the out neighbor.
- *in-degree* = 1 and *out-degree* > 1, in these cases, the coverage provided by the incoming edge must be distributed between the nodes $N^+_{U_r}$ based on how close their coverage is to the $cov(U_i, U_j)$, where U_i and U_j are the nodes of the AG under study, that is, the nodes for which we are computing their DAG_{ij}. The basic idea is that the closer the coverage of an edge $cov(U_r, U_k)$ $U_k \in N^+_{U_r}$ is to $cov(U_i, U_j)$, the higher coverage will be assigned. Closer coverages are expected to be more likely to be the correct ones. The coverage translation is then:
  - For every node $U_l \in N^+_{U_r}$, we define $\mathit{diff}(U_l) = |cov(U_r, U_l) - cov(U_i, U_j)|$.
  - The new coverage is

$$cov(U_r, U_l) = cN^-_{U_r} \frac{|\mathit{diff}(U_l) - \sum_{\forall U_v \in N^+_{U_r}} \mathit{diff}(U_v)|}{\sum_{l \in cN^-_{U_r}} \left| \mathit{diff}(U_l) - \sum_{\forall U_v \in N^+_{U_r}} \mathit{diff}(U_v) \right|}$$

- *in-degree* > 1 and *out-degree* > 1, in these cases it is necessary to assign the nodes in $N^+_{U_r}$ to each node in $N^-_{U_r}$. To do so, we follow the exact same methodology as we followed to assign the excesses found so far in a traversal of the AG to the outgoing coverages of a given node, when computing the coverage of an edge of the DAG.

4 Experimental evaluation

4.1 Threshold computation

Both viaDBG and ViQUF are based on a de Bruijn graph. This requires to determine the set of k-mers used to build that graph. The traditional way to obtain this set is to calculate a threshold such that the k-mers with a higher frequency are selected. The value of the threshold might have a high impact on the results. A method based on the k-mer frequency histogram is used by viaDBG, whereas, as explained, ViQUF uses Kernel Density Estimation (KDE). Therefore, our first experiment is designed to check the impact of the methods used by ViQUF and viaDBG to determine the threshold.

Table 2 gathers the thresholds computed for all the datasets. Those results suggest that the new methodology is more aggressive than that of viaDBG, obtaining in general higher thresholds. To perform a fair comparison, viaDBG and ViQUF were ran with their own threshold and with the threshold obtained by the other tool. The overall results, included in Table 3, show that the viaDBG threshold is quite conservative because the higher threshold of ViQUF does not affect the genome fraction, although an almost negligible grow in mismatches appears. However, the contiguity of the assembly is in general higher with the higher threshold of ViQUF as seen by the N50 values.

TABLE 2
Threshold values to filter solid k-mers obtained for different datasets using an approach based on frequency histogram, as used by viaDBG, or the proposed approach using kernel density estimation (KDE), used by ViQUF.

dataset	Freq. histogram	KDE
HCV-10	16	103
HIV-5	21	146
POLIO-6	12	51
ZIKV-15	5	26
HIV-real	128	172

4.2 Haplotype abundance estimation

In the experimental evaluation of the haplotype relative frequency estimation error, we use the following two measures:

- Mean Estimation Error - which is the average error per haplotype

$$MEE = \frac{\sum_{c \in C} |\hat{freq_c} - freq_c|}{\#C},$$

  where $\hat{freq_c}$ is the estimated relative abundance for the haplotype c and $freq_c$ is the known haplotype relative abundance, finally $\#C$ is the number of haplotypes under evaluation.
- Estimation Error Standard Quasideviation - which measures the amount of dispersion in the frequency errors estimation

$$\hat{S}_{EE} = \sqrt{\frac{\sum_{c \in C}(\epsilon_c - MEE)^2}{\#C - 1}},$$

  where ϵ_c is the error in the frequency estimated for the haplotype c.

Next, we show the full results for frequency estimation. Tables 4–7 show the frequency estimations and estimation errors of each haplotype of the different datasets obtained by each tool. Similar to the average results, these tables show that both Virus-VG and VG-Flow have a slightly better performance on frequency estimation than ViQUF. The reason for this difference is more likely to be related with the raw data than the frequency estimation step. While VG-Flow and Virus-VG use preassembled contigs as input and the number of reads mapping to a contig as abundance estimation, ViQUF uses unitigs, which are faster to compute but shorter than contigs, and the average of the k-mer counting in each unitig as abundance estimation, once again faster to compute but in this case more error prone than mapping the reads. Therefore, because of the input data, the estimation of frequencies by ViQUF are less reliable than the estimation by VG-Flow and Virus-Vg. According to our results, all the tools have good performance. There is a remarkable exception in the case of HIV-5 dataset, with a significantly high error for one of its haplotypes, thus, impacting the overall result. Notice that the three methods make the same mistake overestimating the real frequency of the same haplotype, *NL_43*. This is probably because the *NL_43* haplotype shares most of its information with the rest of the haplotypes.Apart from that, it is important to compare VG-Flow results on ZIKV-15 haplotype by haplotype because according to Table 6 there are large differences in the error of frequency estimation from one haplotype to another.

4.3 Comparison using precision and recall

In the main paper, the metrics we employed are generalizations of common assembly quality metrics used for *de novo* genome assemblers. Recently, new approaches have been suggested to give a new point of view on the quality of the assembly. Table 8 shows the results of an adaption of the well-known computational metrics *precision* and *recall*, suggested by [3].

Before explaining how the metrics are adapted, we have to introduce the term *proper contigs*. The set of proper contigs Q' is a subset of the original set of contigs Q, namely $Q' \subseteq Q$, where $Q' = \{\forall q \in Q / mismatches(q) \leq 1\%\}$. Then, precision is defined as $\frac{TP}{TP+FP}$, where TP (true positives) is the total frequency of the haplotypes correctly predicted by the set of proper contigs and FP (false positives) is the total frequency of contigs $q \in Q'$ which do not match with any haplotype. On the other hand, recall is $\frac{TP}{TP+FN}$ where FN (false negatives) are $1 - TP$, thus $recall = TP$.

Since the results in Table 8 come from the same contigs than the ones in the main paper, similarities are expected. However, we can see new insights that might be interesting to point out. First, Virus-VG and VG-Flow exhibit a slightly better behavior since the haplotypes that they lose are the least abundant; thus, precision and recall suffer less than the metrics in the main paper, where all haplotypes count the same. Furthermore, the precision and recall of ViQUF and PEHaplo for HIV-5 dataset are worse since both retrieve some contigs with high level of mismatches (1% to 2.7%), and these contigs align best against 5_strain_HIV_JRCSF, which is a high abundance haplotype. The results for reference-based method are stable and have no remarkable changes from the previously exposed.

These metrics are more geared towards reference-based methods, albeit [3] extend them to *de novo* constructions. Among other problems, they fail to measure how fragmented the assemblies are, as this is not an issue for reference-based methods, which always retrieve full haplotypes. They prioritize the contigs of the most abundant haplotypes, a characteristic inherit from their original definition, but in viral quasispecies reconstruction, it is very important to also retrieve the low abundance haplotypes correctly. However, we included them to give a broader picture of our method.

4.4 Experimental analysis for different sequencing error rates and number of strains

The next section shows how PeHaplo and ViQUF perform on datasets with different sequencing error rates and number of strains. These data sets have been simulated with Simseq [4] using 2, 5, and 10 strains from the HCV sample. The sequencing error rate goes from 0.00% to 1.00% with step 0.25%. Higher error reads is not a realistic scenario since errors in NGS reads typically range from 0.25% to 0.75%. We ran SAVAGE as well, but unfortunately we were not able to set the correct parameters to get a comparable results, thus we decided to not report the inaccurate results.

TABLE 3
Results for the four viral-quasispecies assembly tools. The table also includes the results for the threshold comparison between viaDBG and ViQUF. For Virus-VG and VG-Flow, we show the elapsed time and memory usage separated into contig assembly by SAVAGE (first value) and the full haplotype reconstruction (second value).

dataset	method	% Genome	N50	misass-emblies	% mis-matches	elap time (min)	memory (GB)
HCV-10	Virus-VG	99.30%	9231	0	0.002	913.48 + 1009.08	26.13 — 8.35
	VG-Flow	99.79%	9293	0	0.001	913.48 + 559.56	26.13 — 8.29
	PEHaplo	94.78%	8661	0	0.013	68.45	8.94
	PredictHaplo	89.79%	9273	0	0.044	4.11 + 175.73	1.14
	CliqueSNV	9.97%	9273	0	2.10	4.11 + 3494.09	17.24
	viaDBG ($t = 16$)	97.72%	8934	0	0.005	69.10	2.81
	viaDBG ($t = 103$)	97.18%	8936	0	0.010	68.12	2.81
	ViQUF ($t = 16$)	97.55%	8944	0	0.046	3.43	1.09
	ViQUF ($t = 103$)	97.37%	8911	0	0.008	3.51	1.09
HIV-5	Virus-VG	96.85%	9632	2	0.332	1619.34 + 312.68	26.83 — 0.64
	VG-Flow	96.87%	9625	2	0.331	1619.34 + 312.20	26.83 — 0.65
	PEHaplo	78.59%	9328	2	0.690	73.33	4.84
	PredictHaplo	99.90%	9663	0	0.591	4.00 + 120.13	1.05
	CliqueSNV	99.86%	9649	0	1.15	4.00 + 93.67	8.51
	viaDBG ($t = 21$)	97.50%	8046	2	0.151	62.34	2.89
	viaDBG ($t = 146$)	95.27%	6237	3	0.161	61.23	2.87
	ViQUF ($t = 21$)	95.58%	9617	2	0.222	3.32	1.07
	ViQUF ($t = 146$)	99.71%	9237	2	0.321	3.26	1.07
POLIO-6	Virus-VG	89.96%	7436	0	0.141	3455.00 + 201.23	17.30 — 0.73
	VG-Flow	99.49%	7388	2	0.137	3455 + 532.33	17.30 — 0.30
	PEHaplo	98.15%	7428	0	0.125	107.96	3.63
	PredictHaplo	49.81%	7428	0	0.646	82.35	0.92
	CliqueSNV	83.07%	7428	0	1.84	27.95	8.45
	viaDBG ($t = 12$)	73.81%	1760	0	0.018	49.21	2.52
	viaDBG ($t = 51$)	80.20%	2290	0	0.016	47.90	2.52
	ViQUF ($t = 12$)	86.90%	4540	0	0.105	3.21	1.07
	ViQUF ($t = 51$)	97.40%	7428	0	0.247	2.61	1.06
ZIKV-15	Virus-VG	99.56%	10212	0	0.077	706 + 407.51	13.45 — 1.37
	VG-Flow	83.05%	10210	0	0.144	706.00 + 406.22	13.45 — 0.62
	PEHaplo	98.32%	10247	0	2.05	321.53	0.08 — 8.80
	PredictHaplo	46.65%	10251	0	0.133	4.06 + 149.68	1.11
	CliqueSNV	66.66%	10251	0	0.036	4.06 + 126.28	8.38
	viaDBG ($t = 5$)	89.85%	1398	0	0.110	65.48	3.25
	viaDBG ($t = 26$)	92.61%	2107	0	0.109	66.03	3.25
	ViQUF ($t = 5$)	80.92%	3042	0	0.111	4.80	1.12
	ViQUF ($t = 26$)	99.08%	10140	0	0.142	4.05	1.11
HIV-real	Virus-VG	83.36%	8637	0	3.384	3550.00 + 440.71	26.85 — 0.80
	VG-Flow	89.99%	5950	0	1.100	3550.00 + 1499.61	26.85 — 1.47
	PEHaplo	91.43%	1262	0	0.074	68.34	3.48
	PredictHaplo	90.21%	8702	0	0.287	4.71 + 100.75	0.87
	CliqueSNV	72.17%	8676	0	1.125	4.71 + 136.68	9.03
	viaDBG ($t = 128$)	87.25%	1813	0	0.197	17.24	3.74
	viaDBG ($t = 172$)	89.36%	1670	0	0.215	17.24	3.75
	ViQUF ($t = 128$)	90.27%	2302	1	0.349	3.78	1.07
	ViQUF ($t = 172$)	90.85%	2267	0	0.292	3.73	1.07

TABLE 4
Frequency estimations and estimation errors for the HIV-5 dataset.

Genome	Real Frequency	ViQUF estimation	ViQUF error	VG-Flow estimation	VG-Flow error	Virus-VG estimation	Virus-VG error	PredictHaplo estimation	PredictHaplo error	CliqueSNV estimation	CliqueSNV error
5_strain_HIV_NL43	11.20	15.71	4.51	24.24	13.04	26.66	15.46	27.00	15.80	12.97	1.77
5_strain_HIV_JRCSF	28.00	30.47	2.47	24.24	3.76	23.47	4.53	24.00	4.00	22.99	5.01
5_strain_HIV_YU2	11.10	5.56	5.53	5.24	5.86	5.08	6.02	5.00	6.10	4.90	6.20
5_strain_HIV_89.6	22.10	20.04	2.05	18.74	3.36	18.13	3.97	18.00	4.10	10.39	11.71
5_strain_HIV_HXB2	27.30	28.20	0.91	27.54	0.24	26.66	0.64	26.00	1.30	16.86	10.44

TABLE 5
Frequency estimations and estimation errors for the HCV-10 dataset.

Genome	Real Frequency	ViQUF estimation	ViQUF error	VG-Flow estimation	VG-Flow error	Virus-VG estimation	Virus-VG error	PredictHaplo estimation	PredictHaplo error	CliqueSNV estimation	CliqueSNV error
HCV_EU155339.2	12.00	12.06	0.06	12.11	0.11	12.10	0.10	13.10	1.10	0.00	12.00
HCV_EU255981.1	13.00	12.69	0.31	12.96	0.04	12.96	0.04	5.00	8.00	0.00	13.00
HCV_EU255973.1	10.00	10.41	0.41	9.98	0.02	9.98	0.02	12.20	2.20	0.00	10.00
HCV_EU255980.1	5.00	5.13	0.13	5.03	0.03	5.03	0.03	6.10	1.10	0.00	5.00
HCV_EU155344.2	5.00	5.01	0.01	5.05	0.05	5.05	0.05	27.80	22.80	0.00	5.00
HCV_EU255989.1	19.00	18.98	0.02	18.89	0.11	18.88	0.12	10.10	8.90	0.00	19.00
HCV_EU255983.1	6.00	6.10	0.10	6.02	0.02	6.02	0.02	12.40	6.40	0.00	6.00
HCV_EU234065.2	8.00	7.97	0.03	8.00	0.00	8.00	0.00	0.00	8.00	0.00	8.00
HCV_EU255982.1	10.00	9.79	0.21	10.02	0.02	10.02	0.02	8.10	1.90	0.00	10.00
HCV_EU255965.1	12.00	11.81	0.19	11.95	0.05	11.95	0.05	5.00	7.00	100.00	87.00

TABLE 6
Frequency estimations and estimation errors for the ZIKV-15 dataset.

Genome	Real Frequency	ViQUF estimation	ViQUF error	VG-Flow estimation	VG-Flow error	Virus-VG estimation	Virus-VG error	PredictHaplo estimation	PredictHaplo error	CliqueSNV estimation	CliqueSNV error
Strain 0	2.00	2.01	0.01	0.00	2.00	2.05	0.05	12.40	10.40	0.00	2.00
Strain 1	2.00	2.01	0.01	2.10	0.10	2.05	0.05	0.00	2.00	0.00	2.00
Strain 2	2.00	2.01	0.01	8.31	6.31	1.98	0.02	32.40	30.40	0.00	2.00
Strain 3	4.00	4.03	0.03	2.19	1.81	2.30	1.70	0.00	4.00	0.00	4.00
Strain 4	4.00	4.27	0.27	4.42	0.42	4.35	0.35	0.00	4.00	0.00	4.00
Strain 5	4.00	4.09	0.09	4.83	0.83	4.29	0.29	0.00	4.00	3.20	0.80
Strain 6	6.00	6.02	0.02	2.77	3.23	4.87	1.13	0.00	6.00	6.50	0.50
Strain 7	6.00	6.21	0.21	4.54	1.46	4.58	1.42	12.10	6.10	6.60	0.60
Strain 8	6.00	6.26	0.26	0.00	6.00	3.66	2.34	0.00	6.00	6.50	0.50
Strain 9	8.00	8.22	0.22	8.88	0.88	8.76	0.76	8.10	0.10	8.10	0.10
Strain 10	8.00	8.39	0.39	8.95	0.95	8.84	0.84	8.00	0.00	8.00	0.00
Strain 11	8.00	8.00	0.00	8.93	0.93	8.77	0.77	0.00	8.00	7.90	0.10
Strain 12	13.00	13.81	0.81	14.68	1.68	14.54	1.54	13.60	0.60	13.38	0.38
Strain 13	13.00	13.70	0.70	14.72	1.72	14.47	1.47	13.40	0.40	13.36	0.36
Strain 14	13.00	13.79	0.79	14.69	1.69	14.49	1.49	0.00	13.00	13.36	0.36

TABLE 7
Frequency estimations and estimation errors for the POLIO-6 dataset.

Genome	Real Frequency	ViQUF estimation	ViQUF error	VG-Flow estimation	VG-Flow error	Virus-VG estimation	Virus-VG error	PredictHaplo estimation	PredictHaplo error	CliqueSNV estimation	CliqueSNV error
seq_1	50.80	46.98	3.81	50.86	0.06	49.32	1.48	77.20	26.40	54.26	3.46
seq_2	25.40	21.81	3.58	18.84	6.56	22.55	2.85	0.00	25.40	10.00	15.40
seq_3	12.70	10.97	1.72	15.91	3.21	15.33	2.63	14.00	1.30	9.62	3.08
seq_4	6.30	9.00	2.70	8.66	2.36	8.35	2.05	0.00	6.30	4.38	1.92
seq_5	3.20	8.99	5.79	3.45	0.25	3.21	0.01	8.70	5.50	6.20	3.00
seq_6	1.60	2.23	0.69	2.27	0.67	1.24	0.36	0.00	1.60	0.00	1.60

TABLE 8
Precision and recall measures for the viral quasispecies tools, de novo and reference-based

	Precision				Recall			
	HIV-5	HCV-10	POLIO-6	ZIKV-15	HIV-5	HCV-10	POLIO-6	ZIKV-15
Virus-VG	100.00%	100.00%	100.00%	100.00%	95.68%	99.79%	98.31%	98.62%
VG-Flow	100.00%	100.00%	100.00%	100.00%	95.60%	99.81%	99.70%	88.82%
PEHaplo	87.18%	100.00%	100.00%	70.50%	75.46%	99.66%	99.42%	97.60%
PredictHaplo	49.26%	100.00%	85.71%	100.00%	48.16%	91.99%	76.20%	55.99%
CliqueSNV	55.20%	0.00%	51.29%	100.00%	88.47%	0.00%	63.20%	84.99%
ViQUF	71.99%	100.00%	100.00%	94.26%	70.51%	98.53%	99.69%	97.95%

The Table 9 gathers the achieved results for PeHaplo and ViQUF. These results show that both tools get accurate results even for the highest error level. However, for 10 strains, both tools have worse performance for intermediate error rates than for high error rates. The reason is likely to be in the simulation more than in the tools since both tools have a correlated performance, namely where PeHaplo has a worse performance in genome fraction or the number of mismatches, the performance is worse also for ViQUF. However, it looks like ViQUF is slightly more stable in terms of genome fraction. In terms of mismatches ViQUF looks more precise giving almost always lower levels of errors. Nevertheless, apart from the case with 5 strains and 1.0% error ratio, where PeHaplo has an unexpected level of mismatches, both tools work well. PeHaplo has a clearly better N50 but this is easily understandable since *de Bruijn* graph methods tend to lose the beginning and the end of the genomes.

5 Plots with the results of the experimental evaluation of the main paper

Here, we present the values shown in the Table 2 of the main paper, using plots for a better view. Figure 3 shows genome

TABLE 9
Results for the HCV data simulation with 2, 5 and 10 strains and with errors from 0.0 to 1.0% with 0.25% step.

		Error	0.00%		0.25%		0.5%		0.75%		1.0%	
			ViQUF	PeHaplo	ViQUF	PeHaplo	ViQUF	PeHaplo	ViQUF	PeHaplo	ViQUF	PeHaplo
Number of strains	2 Strains	% Genome Fraction	99.71%	99.91%	99.97%	99.91%	99.96%	99.91%	99.94%	99.91%	99.81%	99.91%
		Mismatches (kbps)	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
		N50	9252	9284	9281	9284	9279	9284	9276	9284	9263	9284
	5 Strains	% Genome Fraction	99.36%	99.34%	99.66%	99.99%	99.74%	99.33%	99.30%	99.31%	99.17%	99.96%
		Mismatches (kbps)	0.03	0.02	0.09	0.09	0.17	0.02	0.04	0.02	0.04	116.31
		N50	9221	9273	9266	9283	9262	9273	9218	9273	9215	9283
	10 Strains	% Genome Fraction	98.64%	99.98%	96.79%	89.64%	88.75%	89.98%	92.12%	90.20%	96.56%	99.96%
		Mismatches (kbps)	0.12	0.99	0.08	0.26	0.13	0.34	0.08	0.99	0.25	0.86
		N50	9100	9297	8928	9297	9036	9297	8899	9297	9044	9296

fraction, that is, the fraction of all haplotypes retrieved by each method (% Genome). Figure 4 shows the value of N50, that is, the length of the shortest contig needed to be included to cover at least half of the total assembly. Figure 5 shows the error rate of the assembly, that is, the sum of mismatch rate, indel rate, and N-rate (% error rate). Figure 6 shows the elapsed time. Figure 7 shows the peak memory consumption. Finally, Figure 8 shows the mean estimated error of haplotype frequencies (% MEE), and Figure 9 plots the standard quasideviation of the estimated error of haplotype frequencies ($\hat{S}_{EE}$). We do not include misassemblies, since this is a simple integer.

6 Commands to create the datasets of Table 3 of the main paper

Favites, the software used for the simulation, runs with a configuration file where you have to define the software used for the simulation and the evolution rate of the strains. In our case we made the simulation by using DWGSIM, here we show the configuration file:

```
"Sequencing": "DWGSIM",
\dwgsim_path":"dwgsim"

(40000x depth)
"dwgsim_options":"-C 8000 -1 250 -e 0.002 -2 250 -r 0.0 -d 250 -F 0.0 -R 0.0 -X 0.0 -y 0.0 -c 0"
and (20000x depth)
"dwgsim_options":"-C 4000 -1 250 -e 0.002 -2 250 -r 0.0 -d 250 -F 0.0 -R 0.0 -X 0.0 -y 0.0 -c 0"
```

And the evolutionary rate:

```
"seed_sequence_length": 10000,
# Params normal
\tree_rate_loc": 0.0008,
\tree_rate_max": float('inf'),
\tree_rate_min": 0,
"tree_rate_scale": 0.02,
```

And the phylogenetic tree that we achieve in Newick format is:

```
 (N10|72|2.0:0.004224808705067463,(N14|5|2.0:0.019158707294626317,
((N12|87|2.0:0.006835150300967025, N11|89|2.0:0.010340673182958728):0.011236682911299836,
N13|21|2.0:0.024531817257230353):0.015398627402850992):0.004694301281588564):0.0012078787901201905;
```

The average divergence between the different simulations is around 1.08%.

References

[1] A. I. Tomescu, A. Kuosmanen, R. Rizzi, and V. Mäkinen, "A novel min-cost flow method for estimating transcript expression with RNA-Seq," *BMC Bioinformatics*, vol. 14, p. S15, 2013.

[2] M. Jones, "The performance of kernel density functions in kernel distribution function estimation," *Statistics & Probability Letters*, vol. 9, no. 2, pp. 129 – 132, 1990.

[3] A. Eliseev, K. M. Gibson, P. Avdeyev, D. Novik, M. L. Bendall, M. Pérez-Losada, N. Alexeev, and K. A. Crandall, "Evaluation of haplotype callers for next-generation sequencing of viruses," *Infection, genetics and evolution: journal of molecular epidemiology and evolutionary genetics in infectious diseases*, vol. 82, p. 104277, August 2020.

[4] D. N. Sam Benidt, "SimSeq: a nonparametric approach to simulation of RNA-sequence datasets," *Bioinformatics*, vol. 31, no. 13, pp. 2131–2140, 2015.

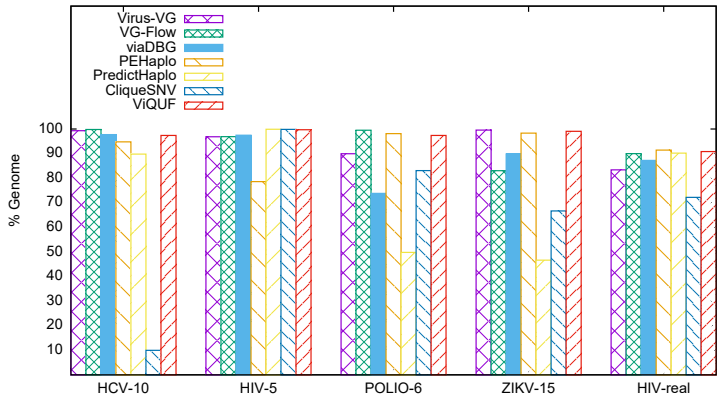


Fig. 3. Genome fraction (% Genome) results for the different viral quasispecies assembly tools.

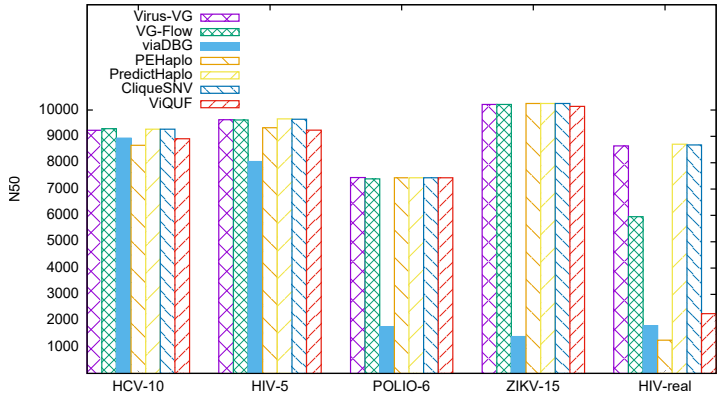


Fig. 4. N50 results for the different viral quasispecies assembly tools.

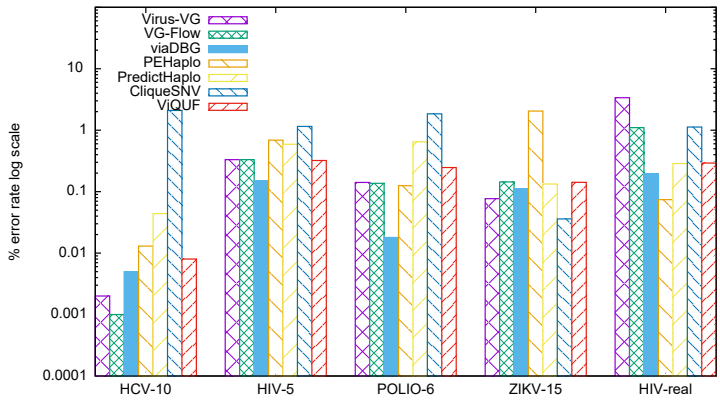


Fig. 5. Error rate results for the different viral quasispecies assembly tools (Y scale logarithmic).

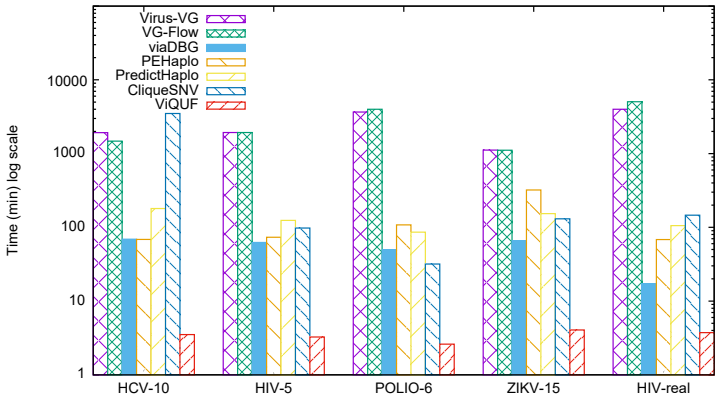


Fig. 6. Elapsed time results for the different viral quasispecies assembly tools (Y scale logarithmic).

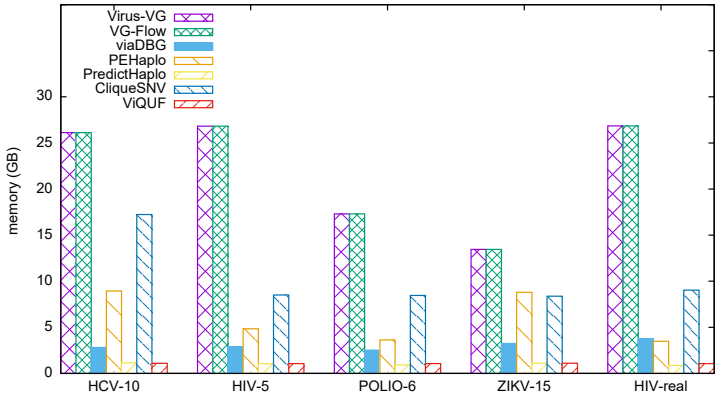


Fig. 7. Peak memory consumption results for the different viral quasispecies assembly tools.

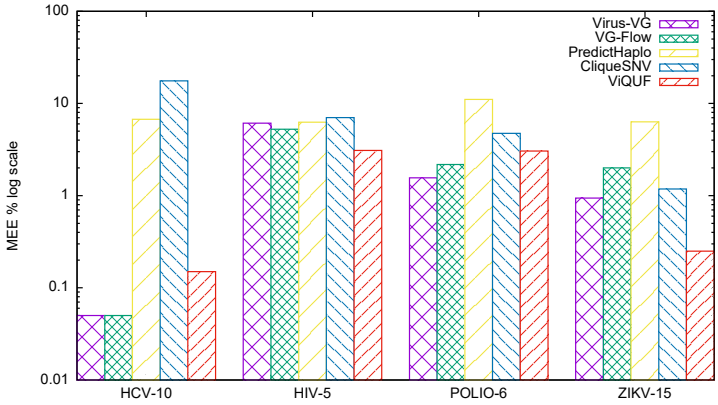


Fig. 8. Mean estimated error of haplotype frequencies (%MEE) results for the different viral quasispecies assembly tools (Y scale logarithmic).

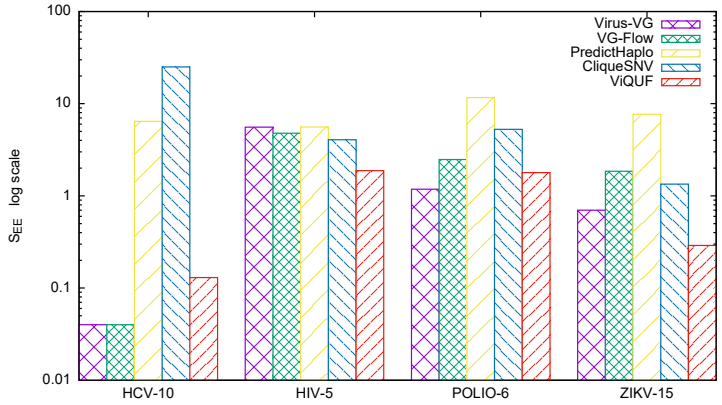


Fig. 9. Standard quasideviation of the estimated error of haplotype frequencies ($\hat{S}_{EE}$) results for the different viral quasispecies assembly tools (Y scale logarithmic).

Memory-Efficient Assembly using Flye

Borja Freire, Susana Ladra, and José R. Paramá

Abstract—In the past decade, next-generation sequencing (NGS) enabled the generation of genomic data in a cost-effective, high-throughput manner. The most recent third-generation sequencing technologies produce longer reads; however, their error rates are much higher, which complicates the assembly process. This generates time- and space- demanding long-read assemblers. Moreover, the advances in these technologies have allowed portable and real-time DNA sequencing, enabling in-field analysis. In these scenarios, it becomes crucial to have more efficient solutions that can be executed in computers or mobile devices with minimum hardware requirements. We re-implemented an existing assembler devoted for long reads, more concretely Flye, using compressed data structures. We then compare our version with the original software using real datasets, and evaluate their performance in terms of memory requirements, execution speed, and energy consumption. The assembly results are not affected, as the core of the algorithm is maintained, but the usage of advanced compact data structures leads to improvements in memory consumption that range from 22% to 47% less space, and in the processing time, which range from being on a par up to decreases of 25%. These improvements also cause reductions in energy consumption of around 3–8%, with some datasets obtaining decreases up to 26%.

Index Terms—compact data structures, genome assembly, long-reads assembly, memory efficiency, third-generation DNA sequencing

1 Introduction

Since the 50s, we are attending to a rapid increase in the scale of the treated bioinformatics datasets [1]–[3]. Current databases, such as the Sequence Read Archive [4], contain a large number of datasets; moreover, they are also growing in size, making old techniques unable to process them. Genomics poses unique challenges in terms of data acquisition, storage, distribution, and analysis [5], which require new innovative approaches.

Nowadays, the most common approach for facing computationally very expensive processes is to use some sort of parallel computing [6]–[10]. This is due to the availability of techniques, useful tools, and cheap hardware. However, another, less frequent, way to obtain scalable systems is to use more efficient methods or data structures in order to reduce the memory consumption and/or time [11]–[13]. Despite now being an unusual approach, in the early days of computer science, it was common to spend considerable effort to obtain more efficient software, as the hardware was expensive and had low computational power. In addition to their intrinsic benefits, these techniques are completely compatible with parallel computing strategies, thus, we can join the advantages of both approaches.

In-memory databases [14] constitute an example of this. Instead of using the traditional setup, where data reside in disk and portions are translated to main memory when needed, these database management systems keep all data in main memory all the time. Obviously, this is a challenge that requires complex procedures and data structures, including compression techniques. Another example is the so-called compact data structures [15], [16]. The idea is basically the same as in in-memory databases, data are stored in the upper levels of the memory hierarchy by using compression. The difference is that this field faces all types of data.

Many compact data structures use bitmaps as the main basic block to build complex data structures. Given a bitmap $B[1 \dots n]$ storing a sequence of n bits, there are three basic operations: *access(B,i)*, which obtains the bit at position i of B; $rank_a(B, i)$, which counts the occurrences of bit $a \in \{0, 1\}$ in $B[1 \dots i]$; and $select_a(B, i)$ locates the position for the i^{th} occurrence of $a \in \{0, 1\}$ in B.

Several data structures (see [17] for example) allow solving these operations in constant time and using $n+o(n)$ bits of total space. There exist implementations that enable fast $rank$ and $access$ operations in only 5% extra space over the original bit array.

By using bitmaps, we present a modification of the well-known assembler Flye [18], aimed at decreasing the amount of main memory used when creating the draft genome assembly and the subsequent assemblies during the rest of phases of the process. Furthermore, the execution time is not affected, but improved in most cases.

Flye was recently compared with five state-of-the-art assemblers, obtaining better or comparable assemblies, while it is an order of magnitude faster [19]. Moreover, Flye obtains longer contigs as it doubles the NGA50 metric. Therefore, taking as a starting point such an efficient assembler in terms of speed as Flye, we decided to face the other relevant parameter for an efficient implementation, its memory consumption.

Decreasing the memory footprint of assembly processes is crucial for new DNA sequencing technologies that aim at offering portable and real-time genome sequencing [20]. In-field analyses are now possible thanks to the development of mobile and affordable devices, such as the pocket-sized Oxford Nanopore Technologies' (ONT) MinION. The main advantage of these mobile genomic labs, which can be deployed for in situ DNA extraction and sequencing, is the possibility of shortening the time from the collection of the

• *B. Freire, S. Ladra, and J.R. Paramá are with Universidade da Coruña, Centro de investigación CITIC, 15071, A Coruña, Spain. E-mail: {borja.freire1,susana.ladra,jose.parama}@udc.es*

Manuscript received xxxxx xx, 2020; revised xxxxx xx, xxxx.

sample to the data analysis. This approach has been recently used, for instance, during the Nigeria 2018 Lassa fever outbreak, where real-time analysis allowed a better understanding of its molecular epidemiology [21]. Currently, most bioinformatics pipelines are executed on high-performance computing cluster or powerful computers, most of the time on the cloud. Thus, data must be transferred and queued into those systems, which slows down the whole process. Moreover, it is possible that these analyses involve personal and sensitive data, such as genetic data; therefore transferring the data to external facilities can be problematic. Thus, due to privacy issues or immediateness, there exist scenarios where the analysis of genomic data in the same place where the data is extracted, is of vital importance. In these scenarios, it is necessary that these real-time analyses can be done not only using portable DNA sequencers but also portable computing devices, such as laptops or smartphones, which have memory limitations.

1.1 Long error-prone reads assemblers

The expected outcome of an assembly process is a set of safe contigs that belong to the genome of interest, hopefully covering as much of it as possible. The traditional way of doing this is to build a de Bruijn graph [22] from k-mers[1] or an Overlap-Layout-Consensus (OLC) graph [23] and then look for safe paths within these graphs [24]. Nowadays, there are several algorithmic ways to discover these paths, and even graph transformations that lead to longer paths like Y-to-V transformation [25], EULER [26], or using omnitigs [27].

Although all these techniques are well-known and they are widely used in practice when working with short reads obtained from second-generation sequencing[2] platforms, they are not as useful with third-generation sequencing reads, i.e. PacBio SMRT, or Oxford NanoPore sequencing. The reason is that, due to the high error rate of the reads, the de Bruijn graphs built with a standard k-mer size (between 25–30 bp) are extremely tangled and the OLC graphs need an extremely large coverage to work properly.

The first attempts of long error-prone assemblers were based on Overlap-Layout-Consensus or on similar string graph approaches [28], but these methods have quadratic complexity. Another way to face the problem is to return to the de Bruijn graph, or more precisely, to a variation called the A-Bruijn graph, which was originally designed to assemble a rather long Sanger reads [29].

Based on the A-Bruijn graph, Lin et al. presented Flye [18], which is able to obtain good results with a rather efficient process. Our work is based on this approach, which is better explained at Section 2.2.

2 Background

2.1 Compact data structures

Compact data structures have been extensively used in bioinformatics. The best example is the FM-Index [30], which is able to store a text using roughly the space required for representing that text in compressed form and, at the same time, is able to locate any substring in sublinear time. It is the main data structure of the majority of short-read aligners including Bowtie [31], BWA [32], and SOAP2 [33]. More specific tasks, such as k-mer counting and k-mer indexation, have also been addressed by using compact data structures. Välimäki and Rivals [34] used a FM-Index-like structure, called compressed suffix array [35] for this. Claude et al. [36] used techniques coming from the field inverted indexes. There are many succinct versions for de Bruijn graphs [37]–[40] that use different compact data structures techniques, among others, the FM-index. An important recent research line in compact data structures is to represent and index genomes of different individuals in very little space [41]–[43]. There is a large list of works in this field, just to cite a few [44]–[48].

As explained, many compact data structures use bitmaps combined with fast rank and select operations. Jacobson [49] proposed a solution able to compute rank in constant time. Given a bitmap B of size n, it uses a two-level directory structure. The first-level directory stores $rank_1(B, p)$ for every p multiple of $s = \lfloor \log n \rfloor \lfloor (\log n)/2 \rfloor$. For every p multiple of $b = \lfloor (\log n)/2 \rfloor$, the second-level directory keeps the relative rank value from the previous multiple of s. By using these data structures, $rank_1(B, i)$ can be computed in constant time by accumulating the values from both directories. The first-level directory returns the rank value until the previous multiple of s. The second-level directory gives the value of rank from that position until the previous multiple of p. Finally, the number of 1s from that position until position i is computed using a precomputed table that stores the rank values for all possible byte values. The sizes s and p are carefully chosen so that the auxiliary dictionary structures use $o(n)$ additional space.

Although $n + o(n)$ representations are asymptotically optimal for incompressible binary sequences, it is possible to obtain better space when the binary sequence is compressible, for example when the number of 1s (alternatively the 0s) is small. In that case, the bitmaps are usually called *sparse bitmaps*. For instance, Raman et al. [50] presented two representations for sparse bitmaps with $nH_0(B) + o(\ell) + O(\log \log n)$ and $nH_0(B) + O(n \log \log n / \log n)$ bits, where $H_0(B)$ is the zeroth-order entropy of B and ℓ is the number of 1-bits in B.

The constant time solution for select is significantly more complex than that of rank. Clark [51] presented a solution based on a three-level directory that requires $3n/\lceil \log \log n \rceil + O(\sqrt{n} \log n \log \log n)$ bits of extra space. For example, in case $n = 2^{30}$, the additional data structures occupy 60% of the original bitmap. Practical implementations of select [17] reuse the same directories used for rank, although this yields $O(\log n)$ time. The simple solution is to binary search in B a position i such that $rank_1(B, i) = j$ and $rank_1(B, i - 1) = j - 1$. Real implementations, instead of using the rank operation as a black box, binary search the directories D_s and D_b to speed up the query, yielding a $O(\log \log n)$ cost.

1. k-mers are subsequences of length k contained within a biological sequence. In our work, they are sequences of k nucleotides (i.e. A, T, G, and C).

2. Second-generation sequencing is also known as next-generation sequencing (NGS).

2.2 Flye assembler

Most single-molecule or third-generation sequencing assemblers spend much time making sure that the created contigs are correctly assembled. Flye, in contrast, does not waste time on making sure that the created contigs are correct. Actually, Flye intentionally builds misassembled contigs and, from them, it builds accurate and longer contigs. Briefly, Flye constructs an assembly graph and, starting from a given read, it creates random walks on that graph. Once a random walk is built, all reads that map with that walk are found, and a consensus contig is built from the whole set of reads. Obviously, this new consensus contig is legit because it is supported by several reads. Once all contigs have been created, they are glued into an accurate assembly graph which is untangled, and unbridged repetitions are solved by using the number of reads that traversed consecutive edges in the accurate assembly graph. More precisely:

1) ***Building a draft genome assembly and generating consensus contigs***. To deal with the drawbacks of long reads, Flye corrects them before building the assembly. This stage computes an A-Bruijn graph that uses only solid k-mers[3] instead of all the k-mers from the reads. Furthermore, the value stored in an edge between nodes X and Y of the graph corresponds to the distance between them in a given read Z, unlike traditional de Bruijn graphs, which store the char following node X.
   Once the graph is built, Flye generates arbitrary paths in the graph[4] creating inaccurate contigs. Then, a consensus process is carried out using all the reads that contribute to the contig. Finally, these new "accurate" contigs are used to build an accurate assembly graph, which can be traversed like a regular de Bruijn graph.
2) ***Treating repetitive regions***. One of the biggest challenges when assembling NGS data is to deal with genome repetitive regions, which are the result of recombination, transposons and/or mini/microsatellites. The success of the de Bruijn graphs in the genome assembly field is mainly due to their ability to represent repeat families as mosaics of ideally error-free[5], sub-repeats [29].
   Furthermore, overlap graphs scale quadratically with the number of reads, thus, they become unfeasible for NGS. The direct application of the classic de Bruijn graph does not obtain accurate assembles from long reads, and thus OLC graphs with high-coverage reads are the usual choice to obtain an accurate assembly. However, with OLC, it is not possible to define repeat families, thus much research efforts have been devoted to adapt de Bruijn graphs to be useful for long reads.
   Flye is able to treat repetitive regions using a de Bruijn graph because the reads have been corrected in the previous step. Therefore, using previous consensual reads, an unravelled *repeat graph*[6] is built. Since the idea behind repeat graphs is actually the same as that of the de Bruijn graph, Flye is able to transfer the power of de Bruijn graphs to long reads without losing accuracy or demanding extremely high levels of coverage throughout *repeat graphs*.
3) ***Polishing the final genome***. Once the contigs are obtained, most NGS assemblers are capable of increasing their length by using paired-end information to add topological knowledge. These extended contigs are then referred to as scaffolds. Furthermore, this step also allows polishing the obtained contigs removing those which are misassemblies, namely chimaeras, and thus increasing the veracity of the assembly.

All of these three steps are critical and need to be treated carefully. Even though all the steps are equally relevant, the most memory demanding phase is the first one, since it has to quickly process all k-mers in reads and build consensus contigs, which requires having all overlaps between reads. Therefore, since our goal is to reduce memory consumption, we have focused our improvements on this module.

Next, we will introduce how Flye builds the assembly draft. It is important to remark that this work does not introduce any changes to the Flye's assembly algorithm. Our goal is to improve Flye's memory efficiency, and therefore its scalability, by using new data structures to store and manipulate data, but always using Flye's assembly procedure.

The traditional genome assembly process is based on creating a graph representation from the reads using k-mers and then looking for safe solutions/paths inside the graph. Afterwards, multiple heuristic steps are applied to extend these solutions, and once the contigs have been stretched, the final stages of the process are scaffolding and gap filling.

However, this process suffers from problems when dealing with long reads due to, as explained, their high error ratio, which is around 15% for long reads compared to 0.1–1% for NGS. To overcome this, the Flye's assembly process consists of three different phases: (i) approximate k-mer counting, (ii) selection and indexing of solid k-mers, and (iii) draft genome assembly. The first two steps focus on filtering the k-mers in reads and selecting those k-mers that can be considered genomic. Once Flye has selected the legit genomic k-mers, it assembles the reads looking for those that overlap each other and also fulfil a set of restrictions. As a final step, Flye generates consensus contigs to build *long error-free reads* and, from them, it obtains an accurate genome assembly.

3. A solid k-mer is a k-mer that appears at least t times in the reads, being t a threshold.

4. Instead of looking for the best one, Flye extends the paths selecting an arbitrary read. Therefore, it does not lose time assaying every single overlapping read and selecting the most promising one.

5. Note that de Bruijn graphs work for "perfect" repetitions but unfortunately DNA repeat families are usually full of gaps and mismatches. Therefore, even with de Bruijn graphs, treating repeat regions remains messy.

6. A repeat graph is an alternative graph representation that compactly represents all repeats in a genome and reveals their mosaic structure [18], [29]. Furthermore, repeat graphs can deal with mismatches and gaps inside repetitions, offering a better way of treating and detecting repetitive regions.

2.2.1 Counting k-mers

The high error rate of the reads produces a large number of false or non-genomic k-mers.[7] In order to overcome this problem, Flye separates the genomic and non-genomic k-mers by counting the appearances of the k-mers in the reads and selecting only those whose frequency is above a threshold t. While this might sound simple, it actually becomes a hard problem when working with long reads, due to the huge amount of k-mers present in the reads, which makes their indexation extremely difficult in a reasonable time and space. In fact, this problem is the origin of a research line by itself [52]–[54].

The threshold used by Flye in this phase is automatically selected as a value between 2 and 5, depending on the coverage of the input dataset, which is estimated by Flye from the genome length (provided as a parameter) and the sum of the read lengths.

The solution of Flye requires two counts: (i) approximate counting, and (ii) exact counting. The approximate counting is carried out by using a Bloom filter [55], a hash table that does not solve collisions. Therefore, one hash entry can accumulate appearances of different k-mers. This step removes k-mers with a very low frequency.

We illustrate an example of the use of the hash table for the approximate counting in the left part of Figure 1. In the upper part, we include two reads (read_1 and read_2), and, for each of their positions, the k-mer that starts at that position. As explained, in this approximate counting, collisions are not solved. Observe that entry 4 accumulates six appearances, which correspond to four appearances of k_{13} and two appearances of k_{14}, since the hash function $hash_1$ maps both k-mers to the same entry.

Being L_r the average length read, N_r the number of reads, and $GenomeSizeFactor$ a value between 1 and 16 that Flye calculates based on the estimated size of the genome provided by the user, this phase costs $\theta(L_r * N_r)$ time and uses $GenomeSizeFactor * 4^{15} * 8$ bits.

In a second pass of the reads, Flye counts the exact occurrences of the k-mers whose entries reached the threshold t in the first pass. For this exact counting, Flye uses the well-known data structure Cuckoo hash [56]. It is a variation of hashing that assures a small and constant number of iterations for indexing and accessing operations. This is achieved by using several hash functions, usually two is enough, and a collision handler. When a collision is produced, the corresponding key already stored at the entry is moved to one of its other possible positions, defined by an alternative hash function. This can cause another collision, which is solved in the same way.

We illustrate the exact counting in the right part of Figure 1, including different states of the hash table for different instants of the process. Now, the entries of the hash table contain the key (a k-mer), in addition to the counter. In reality, a Cuckoo hash usually uses two tables of the same size, and, at each entry, several keys can be stored (collisions). For a given key, if $hash(key) = i$, that key can be in the entry i of either table. However, in order to simplify the figure, we use just one table with just one key per entry.

In the hash table under the (a) label, we show the state of the table after traversing read_1 until position 6. As seen, k_{13} and k_{54} appeared twice and k_{32} once. These keys are placed in the position given by the hash function $hash_1$. Under the label (b), we show the state of the hash table after processing position 7 of read_1. The k-mer in that position (k_{44}) is mapped to the entry 5 according to $hash_1$, which is already occupied by k_{32}. Therefore, k_{32} is moved to its alternative position given by the hash function $hash_2$, that is, to position 15. Once position 5 is released, k_{44} is placed there.

Under label (c), we show the state of the hash table when processing position 4 of read_2. In that position, the k-mer k_{14} begins, and $hash_1$ maps it to entry 4, which is already occupied by k_{13}. Then, k_{13} should be moved to its alternative position given by $hash_2$, which is position 5, thus releasing position 4 for k_{14}. However, position 5 is also occupied by k_{44}. Then, k_{44} is also moved to its alternative position defined by $hash_2$, which is position 11. Again, position 11 is occupied, in this case by k_{54}, but now, we enter in a cycle, because the alternative position of k_{54} is 4. To avoid this situation, there is a limit in the number of handled collisions (logarithmic in the table size). If that value is reached, this requires the change of the hash functions $hash_1$ and $hash_2$, and all the k-mers should be reallocated accordingly.

This method guarantees constant time access as long as the table is not occupied more than 50% of its capacity. To avoid this, when the number of collisions triggers the change of the hash functions, the table size is doubled as well.

The worst-case time complexity of the exact count is $O(L_r * N_r + S^2)$ time, being S the number of k-mers that passed the approximate counting. Recall that all k-mers must be processed, checking if they have passed the first threshold, and then inserted in the Cuckoo table. These insertions may cause duplication of the hash table and the reallocation of most entries. Observe that each duplication implies $O(S)$ reallocations, and, in the worst-case scenario, this can happen for $O(S)$ keys.

Regarding space, the memory required at the beginning is $O(S)$ bits, but this is doubled when the number of collisions triggers a reallocation, therefore it can reach $O(S^2)$ bits in the worst-case scenario. In practice, this easily reaches 20–30 GB.

Here we can see one of the weaknesses of Flye that our work addresses. In order to ensure an efficient use of the Cuckoo hash, and therefore to allow fast access to the information of any k-mer, Flye needs big amounts of memory. We aim at replacing the Cuckoo hash table by a more compact data structure without losing speed.

2.2.2 Selecting/Indexing k-mers

Although the previous phase significantly reduces the amount of non-genomic k-mers, some non-genomic or low frequent k-mers remain. Therefore, a second filtering step is necessary to minimize those non-genomic k-mers. Furthermore, more information about k-mers is needed for the next steps, such as the reads in which each k-mer appears,

7. The maximum number of non-genomic k-mers is $(L_r * Fr) * Cov * N_r$, where: L_r is the average length read, Fr is the failure ratio, Cov is the coverage, and N_r is the number of reads.

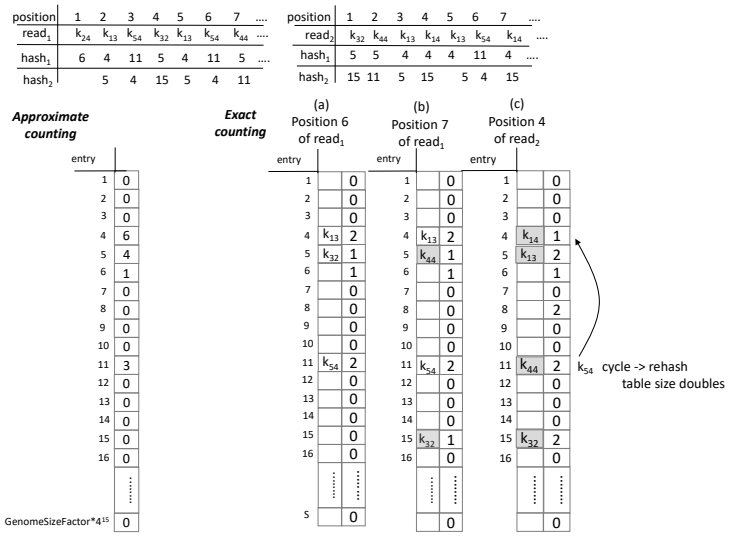


Fig. 1: Counting k-mers in Flye.

and the position inside these reads. This second filter uses a different threshold, which depends on the results of the first filtering step. Actually, the second threshold is equal to the highest frequency that ensures that the number of selected k-mers is greater than the length of the genome.

This step requires another traversal of the reads, and the k-mers that pass this second threshold are indexed into a Cuckoo hash again, but now, they are stored with the positions within the reads where they appear,[8] that is, a list of pairs $(read, positions\_inside\_read)$, thus creating an index of the appearances of the solid k-mers.

This index is the key element for the subsequent step of Flye, that is, the assembly phase. To perform the assembly, Flye uses a variation of the de Bruijn graph called *A-Bruijn graph*. However, the graph is never created and Flye implements it "virtually" by means of this index. The nodes are the reads and the edges are simulated by searching "on the fly" overlaps between reads. Therefore, since that operation is very frequent, this operation must be fast. The overlaps are computed by taking all the k-mers of a read, and, for each one, this index gives all the reads where that k-mer is also found. With this information, Flye computes the overlapping reads.

The creation of this index consumes $O(L_R * N_r)$ time. In space, we need $O(S^2)$ bits for the hash table, plus $O(S * Cov)$ bits for the appearances of the k-mers[9], being Cov the average coverage of the dataset. Thus, the worst-case space complexity is $O(S^2 + S * Cov)$.

8. Each k-mer can occur an arbitrary number of times within a read.

9. There is another filter in this phase, and the number of k-mers considered decreases, but we keep S as the number of solid k-mers.

Algorithm 1: FlyeWalk (*AllReads*, *MinOvelap*, *HashTable*)

```
1  Contigs ← empty set of contigs
2  UnprocessedReads ← AllReads
3  IndexOfReads ← BuildIndex(AllReads, HashTable)
4  for each Read in UnprocessedReads do
5      ChainOfReads ← ExtendRead (UnprocessedReads,
         Read, IndexOfReads, MinOverlap)
6      ContigSequence ← Consensus(ChainOfReads,
         AllReads, MinOverlap, IndexOfReads)
7      add ContigSequence to Contigs
8      remove Overlap(ChainOfReads) from
         UnprocessedReads
9  end
10 return Contigs
```

2.2.3 Assembling contigs

As previously explained, Flye does not use a real de Bruijn graph that is traversed seeking for contigs, but simulates this graph navigation by obtaining all the possible overlaps between reads. More precisely, the construction of contigs is done by checking the overlaps between a processed read and others, while considering several restrictions.

Algorithm 1 shows this process. The algorithm starts by obtaining for each read all its overlapping reads. The process uses the index described in Section 2.2.2. The procedure *BuildIndex* processes all reads, each one is traversed position by position. The k-mer starting at the processed position is used to query the index and, if it is solid, it gives all the overlapping reads.

Once the overlaps have been computed, the process continues with the *for* in Line 4, which processes the reads in

Algorithm 2: ExtendRead(*UnprocessedReads*, *Read*, *MinOverlap*, *IndexOfReads*)

```
1  ChainOfReads ← empty sequence of reads
2  while true do
3      NextRead ← FindNextRead(UnprocessedReads,
         Read, MinOverlap,IndexOfReads)
4      if NextRead = empty string then
5          return ChainOfReads
6      else
7          add NextRead to ChainOfReads
8          Read ← NextRead
9          remove Read from UnprocessedReads
10     end
11 end
```

random order. Then, in Line 5, each read is extended using the function *ExtendRead*, which is shown in Algorithm 2. Given a read *Read*, *ExtendRead* finds an unprocessed read that overlaps with *Read* by at least *MinOverlap* base pairs. This is done during the call of the function *FindNextRead*. Flye does not waste much time checking if the next read fits well in the current path; it just chooses one that overlaps *MinOverlap* base pairs with the current read and fulfils other simple conditions. This process continues until it cannot find another read overlapping the last processed read.

Finally, *Consensus* constructs the consensus of all reads that contribute to the processed *ContigReads*. The process is as follows. Let $Read_1$, $Read_2$, ..., $Read_n$ be the reads in *ContigReads*. Let $prefix(Read_i)$ be the overlapping region between consecutive reads $Read_{i-1}$ and $Read_i$, let $suffix(Read_i)$ be the suffix of $Read_i$ after the removal of $prefix(Read_i)$, and let $concatenate(ChainOfReads)$ be the concatenation $suffix(Read_1)||suffix(Read_2)||\dots||suffix(Read_n)$.
Then, all reads from the dataset are aligned to $concatenate(ChainOfReads)$ using the method minimap2 [57]. The consensus is taking by the majority vote. Finally, the reads considered in the consensus step are removed from the *UnprocessedReads* (Line 8), and therefore they are not considered anymore.

In this process, with $O(N_r^2)$ time, the *Consensus* is the dominant cost.

3 Our proposal: Compact Flye

In this section, we describe our memory-efficient variant of Flye assembler, where we use compact data structures. We detail how our proposal addresses each of the phases of the method.

3.1 Counting k-mers

3.1.1 Approximate counting

Our aim is to perform the same filtering, but without using such a large amount of memory. Moreover, we do not want to penalise the temporal efficiency; thus, our target is also to maintain the execution times in the same order of magnitude or even improve them.

The original method uses 8-bit counters for this phase, allocating a hash table of size $GenomeSizeFactor * 4^{15} * 8$ bits, where $GenomeSizeFactor$ is set to 1 or 16 depending on the size of the genome. This large space hardly provokes collisions. For example, in the case of k-mers of length 15, those 8-bit counters for all the 4^{15} possible combinations would require 8 GB.

Instead, we propose an adaptive solution that is able to store the same number of entries as the hash tables of Flye (both, of 1 GB and 16 GB), but using much less space. For the approximate counting, instead of allocating a complete byte for the counters of k-mers, we use just t bits, being t the given threshold, which is a number between 2 and 5. When a new appearance occurs, the first bit set to 0 is changed from 0 to 1. For instance, if the threshold is 5, then each entry initially has the value 00000. The first appearance changes the entry to 10000, the second to 11000, the third one to 11100, and so on. With this approach, to perform the update, it is only necessary to access the memory location and flip just one bit, rather than adding 1 to the memory location, which requires moving the data to the Arithmetic Logical Unit of the processor and executing an operation of addition.

With this approach, we do not have a real count, that is, we do not know how many times an entry has been processed. This is not a problem, as at this step of the process, we are only interested in those entries that have been found t times or more, and thus, their corresponding k-mer passes this filtering step. Recall that in Flye, this is an approximate counting, as each entry can accumulate appearances of different k-mers.

Our approach is able to filter as many k-mers as Flye does, but only requiring $GenomeSizeFactor * 4^{15} * t$ bits. Of course, the key of a low space requirement is that we are assuming that the threshold value is small, between 2 and 5, otherwise valuable k-mers would be removed. Therefore, in the worst case, our structure is always cheaper than 1 GB or 16 GB (the value of the hash table of Flye), independently of the genome size.

Figure 2 shows an example of this process under the brace labelled "approximate counting". More concretely, we have two reads composed of different k-mers, and we will filter those that have more than 3 appearances ($t = 3$). Thus, our data structure contains 3 bits per entry, and only those entries having their third bit set to 1 pass this filter. We denote b the bitmap composed of the last bit of all entries, as indicated in the figure. Observe that entry 5 has reached the threshold, due to the occurrences of two different k-mers that have the same hash value, namely two appearances of k_{32} and two appearances of k_{14}. After the third appearance, additional occurrences are no longer recorded.

Although the conceptual description is that shown in Figure 2, we implemented it in a faster way, to avoid the sequential search of the first bit set to 0. Specifically, the first $t-1$ bits of each entry are joined in a unique bitmap. In addition, the bits are completely flipped so the first 1 is set in the least significant position. Then, given the $t-1$ bits of an entry, when a new occurrence should be registered, we simply shift one bit to the left and we introduce a 1-bit on the right. The leftmost bit, which is lost, is used to update the corresponding entry of the t^{th} bit, which is stored separately from the others in bitmap b. For example, if the entry is 001, first it is shifted one bit to left, and the leftmost bit (0), is used

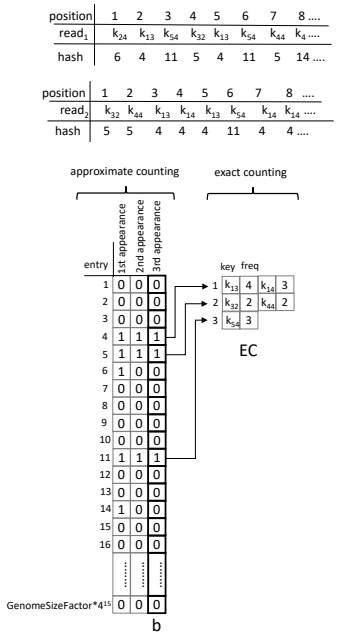


Fig. 2: Example of the k-mer counting.

to update the corresponding entry at b, and we introduce a 1 on the right, obtaining 011. Therefore this method has a cost of $O(1)$, regardless of t.

An alternative implementation is using a counter of $\log\lceil(t)\rceil$ bits, which would save some space. In this case, the bitmap corresponding to the first $t-1$ bits is replaced by an array of counters of $\lceil\log(t-1)\rceil$ bits, while bitmap b is kept. When a new occurrence of a k-mer appears and the corresponding value at the counter is already $t-1$, we set the 1-bit of the corresponding entry of bitmap b. In practice, we will use the method shown in Figure 2 when $k \leq 15$, and the counter of $\lceil\log(t-1)\rceil$ bits when $k > 15$.

The time cost for this step is the same of Flye, that is, $\theta(L_r * N_r)$. The space consumption is $GenomeSizeFactor * 4^{15} * t$ bits if $k \leq 15$ and $GenomeSizeFactor * 4^{15} * (\log\lceil(t-1)\rceil + 1)$ when $k > 15$.

3.1.2 Exact counting

Once the approximate counting has been done, we create a new data structure, denoted EC, to support the exact counting. This data structure has as many entries as positions set to 1 in the k-mer bitmap b, that is, $rank_1(b, size(b))$ entries. At each entry, we store a pointer to a list of pairs, each pair containing a k-mer that has passed the first filtering and an integer counter for storing the exact number of occurrences. Then, in a second traversal of the reads, for each read k-mer k_r that satisfies $b[hash(k_r)] = 1$, we compute its position $pos = rank_1(b, hash(k_r))$. In case k_r is already stored at $EC[pos]$, we increase its counter by one. Otherwise, a new pair $(k_r, 1)$ is added to the list pointed to by the pointer in $EC[pos]$.

In our example of Figure 2, when the traversal reaches position 4 of $read_1$, which is the first appearance of k_{32}, $EC[2]$ is empty[10], therefore we write $(k_{32}, 1)$ in the list pointed to by $EC[2]$. When the traversal of $read_1$ reaches position 7, k-mer k_{44} is also hashed to position 5, but the list pointed to by $EC[2]$ has only one pair, with key k_{32}, therefore a new pair $(k_{44}, 1)$ is added to the list pointed to by the pointer in $EC[2]$. It is important to remark that the use of a list of pairs at each entry does not damage the execution times significantly, as these lists are generally short.[11]

Here we can see one of the main differences between our method and Flye. If we compare Figures 1 and 2, we can see that compact Flye uses a simple bitmap of $GenomeSizeFactor * 4^{15}$ bits to index the k-mers, and the collisions, such as in the case of k_{13} and k_{14}, are stored in a list pointed to by an entry of EC. However, as explained, the lists of collisions are short. Instead, in Figure 1, Flye uses the Cuckoo hash table, which gives constant time access but requiring a big amount of memory and also time, due to reallocations. We can see how the entries of k_{13} and k_{14} are separated. Thus, compact Flye reduces the memory usage in exchange of probably increasing time processing when accessing to those counters in next steps of the process.

The worst-case time complexity of our approach is $O(L_r * N_r + S^2)$, since this phase processes the $L_r * N_r$ input k-mers, and, in the worst-case scenario, all the solid k-mers are mapped to the same entry, and thus, the S k-mers are added to just one list pointed to by one entry of EC. The data structure requires $GenomeSizeFactor * 4^{15}$ bits of space for bitmap b, which is constant, plus $O(S)$ counters, therefore the worst-case space complexity is $O(S)$.

3.2 Selecting/Indexing k-mers

Analogously to original Flye, reads are processed again to index only the k-mers with a number of occurrences higher than the second threshold.

Figure 3 shows the result of this step with our running example, assuming that the new threshold is 3 again. As in the case of the exact count, compact Flye relies on bitmap b, of $GenomeSizeFactor * 4^{15}$ bits, to index the k-mers, and now the collisions, such as in the case of k_{13} and k_{14}, are stored in an array, ordered by frequency of appearance. In our example, k_{13} is the first entry of the list corresponding to the hash entry 4, since it is more frequent than k_{14}.

Therefore, the worst-case time complexity of our method is $O(L_r * N_r + S^2)$, which includes both indexing and sorting the arrays. In space, we need the $GenomeSizeFactor * 4^{15}$ bits of bitmap b (constant), and $O(S * Cov)$ for storing the appearances of the solid k-mers.

3.3 Contigs Assembly

As the main goal of this research is to prove the good properties of compact data structures for the implementation of

10. Since $hash(k_{32})$=5, and $rank_1(b, 5) = 2$, then k_{32} must be included in the *second* entry of EC.

11. For instance, in our experiments for the *E. coli* datasets, the average number of elements at each list of EC was lower than 2, and the maximum number of elements was not higher than 5.

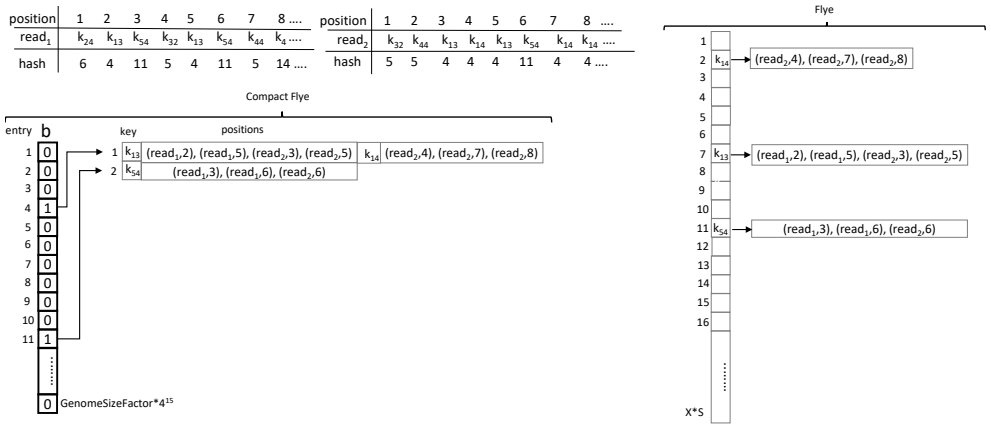


Fig. 3: Example of the k-mer indexing.

TABLE 1: Time complexities. L_r average length of reads, N_r number of reads, S the number of solid k-mers.

	Compact Flye	Original Flye
Approx Count	$\theta(L_r * N_r)$	$\theta(L_r * N_r)$
Exact Count	$O(L_r * N_r + S^2)$	$O(L_r * N_r + S^2)$
Indexing	$O(L_r * N_r + S^2)$	$O(L_r * N_r)$
Assembly	$O(L_r * N_r + S^2 + N_r^2)$	$O(L_r * N_r + N_r^2)$

bioinformatics tools, we wanted to compare time and space results of our proposal with those obtained by the original software, but without significantly changing the underlying algorithm. Thus, we maintain the original A-Bruijn algorithm for genome assembly, as changes on the algorithm would lead to different results with a higher/lower number of contigs and/or longer/shorter contigs.

3.4 Comparison of time and space

We include a summary of the time complexities for our proposal and the original method in Table 1. Worst-case time complexities of the approximate and the exact counting phases are the same for both approaches. However, these worst-case scenarios are too pessimistic. For instance, in the case of compact Flye, the worst-case scenario occurs when all the keys are mapped to the same entry of the hash table, and thus it degenerates to a linked list. This is extremely rare in practice when using well-designed hash functions. In the case of the original Flye, collisions may cause resizing and rehashing the table. This may not cause a quadratic time in the number of solid k-mers, but it actually has a great impact, and it will be reflected in the experimental section.

In the indexing phase, original Flye has a cost of $O(1)$ per processed k-mer, whereas the compact version has to deal with collisions plus the sorting of the arrays. However, collisions only occur when $k > 15$, and arrays are short, therefore, as we will see in the experimental section, only when $k > 15$, compact Flye pays a price in time. However, the use of the index during the assembly phase (employed to search for overlaps between reads) does not introduce

TABLE 2: Space complexities. GSF denotes $GenomeSizeFactor$, S the number of solid k-mers, Cov the coverage, t the threshold used in the counting, and $outp$ the assembly output.

	Compact Flye	Original Flye
Appr. Count	$GSF * 4^{15} * t$	$GSF * 4^{15} * 8$
Exact Count	$O(S)$	$O(S^2)$
Indexing	$O(S * Cov)$	$O(S^2 + S * Cov)$
Assembly	$O(S * Cov + outp)$	$O(S^2 + S * Cov + outp)$

significant changes in times in any case, since the dominant cost is the quadratic cost in the number of reads, which is the same for both approaches.

Table 2 shows the space complexity. In the approximate counting, we use t bits (or $\log\lceil(t-1)\rceil + 1$, if $k > 15$) per entry instead of 8 bits of the original Flye. The second and more important difference is that in the exact count and the index, the size of the structure used by the original approach is tied to the number of collisions, doubling the data structure when the number of collisions surpasses a threshold. However, our data structure is only tied to the simple bitmap b, of constant size, and only increases the size of the entries linearly with the number of solid k-mers that passed the first filter (S). On the contrary, in the case of the original Flye, collisions and therefore duplications pose a big price in space. These duplications of the hash table are inherited during the indexing phase, therefore, in the space complexity the original Flye has an additional S^2 term.

As summary, we trade off time for the sake of reducing space consumption. In the time complexity, the only significant difference is the presence of an additional S^2 term caused by the fact of having linked lists instead of a constant-time Cuckoo hashing, which is a very pessimistic scenario that does not occur in practice, as lists are generally short. The counterpart is that the original Flye has in its space complexity that additional S^2 term in the exact counting, indexing, and assembly phases, precisely due to the use of that very efficient Cuckoo hashing strategy. However, in

practice, the price in space that Flye has to pay is much bigger in comparison with the worsening in times of the compact version. Compact data structures are more memory friendly, and this yields even slight improvements in time in some cases, as we will see in the experimental section.

4 Experimental evaluation

4.1 Experimental framework

All experiments were run on an Intel® Core™ Xeon es2470 CPU @ 2.3 GHz (32 cores), 64 GB of RAM, over Debian GNU/Linux 10 (buster). All programs were coded in C++. We use version 2.4 of Flye. Our code is available at https://bitbucket.org/bfreirec1/compactflye. To ensure the availability of all datasets, we have collected them and uploaded them into the following repository: https://bitbucket.org/bfreirec1/datasets-compactflye.

We have run the tests with two different k-mer sizes: the k value recommended by Flye for each dataset ($k = 15$ for the smallest datasets and $k = 17$ for the largest datasets), and a higher k value, more concretely, $k = 31$, for all datasets. All experiments were run five times for the smallest datasets and three times for the largest datasets, and we report the average results.

We have used five datasets from Oxford Nanopore Technology (ONT) and PacBio (PB) sequencers, some of which were used also by the authors of Flye [18]. We describe now the datasets, and include some properties in Table 3.

- `BACTERIA-PB` dataset[12] contains data gathered with a PacBio RS II System and P4-C2 chemistry on a size selected 20kb library of *E. coli* K12 substr. MG1655.
- `BACTERIA-ONT` dataset[13] also contains reads from whole-genome shotgun sequencing of the model organism *E. coli* K-12 substr. MG1655, but generated on a MinION device.
- `WORM` dataset[14] contains Pacific Biosciences reads (coverage 40x) from a Bristol mutant strain of *C. elegans* genome of length 100 Mb (6 chr.).
- `DROSOPHILA-PB` dataset[15] contains Pacific Bioscience reads (coverage 120x) from a subline of the ISO1 strain of Drosophila melanogaster.
- `DROSOPHILA-ONT` dataset[16] contains reads from Oxford Nanopore technology (coverage 30x) from a subline of the ISO1 strain of Drosophila melanogaster.

4.2 Results

4.2.1 Memory consumption

Figure 4 shows the memory consumption of the original Flye and that of our improved version. In all cases we obtain important improvements, ranging from 22% to 47% less space. We require almost half the space for almost all experiments, except for `WORM` with $k = 31$, which uses 78% of the space needed by the original version. The most remarkable result is observed for `DROSHOPILA-ONT` with $k = 31$. In this case, the physical memory of the machine (64 GB) was not enough for the original Flye. On the contrary, our version finished successfully, which shows that our version is more scalable due to better memory usage.

4.2.2 Time performance

Figure 5 shows the processing times of the complete process. One of the most significant improvements is obtained for `DROSHOPILA-ONT` with $k = 17$ (the recommended setup), where our method is 13% faster, whereas, as explained, with $k = 31$, original Flye did not run in our machine. In `WORM` dataset, the results are on a par, whereas in the smallest datasets, our method is between 5% and 11% faster with $k = 15$ and between 8% and 18% with $k = 31$. For `DROSHOPILA-PB`, results are on a par when using $k = 17$ and 25% faster for $k = 31$. These results show that improved memory usage and, therefore better scalability, can even lead to better runtimes with large datasets when physical memory is nearly exhausted, or when the datasets are smaller, but significant parts of the data structures can be kept in higher levels of the memory hierarchy, like in our `BACTERIA` datasets.

We have measured the time spent at each of the four phases of the assembly process. We show the results in Figure 6. In the small datasets, the pre-assembly phases are faster with our bitmap-based data structures compared to the Cuckoo based hash table of Flye, whereas for the assembly phase, both methods are on a par.

The time complexity of the approximate count is the same in both approaches, $O(1)$ per processed k-mer, still compact Flye obtains an improvement of 33% in the `WORM` dataset, but it is on a par in the case of `DROSOPHILA-ONT`. However, in the exact count, here we can see clear differences. Our method has a worst-case cost of $O(S)$ per processed k-mer, while the original Flye has $O(1)$ access time if we exclude the duplications of the Cuckoo hash table needed to keep that $O(1)$ access time. However, as seen, each duplication requires a considerable waste of time. This implies that compact Flye is between 1.20 times and 4.24 times faster than the original version. The counterpart is in the indexing phase, where compact Flye has $O(L_r * N_r + S^2)$ time due to the sorting of the arrays and the mixture of data from several k-mers in the lists of the index, which is worse than the $O(L_r * N_r)$ of the original FLye, and this can be seen in the largest datasets, where our method is between 37% and 48% slower. Nevertheless, our index based on a bitmap does not slow down the assembly. As it can be seen from the figures, both approaches are on a par in all cases.

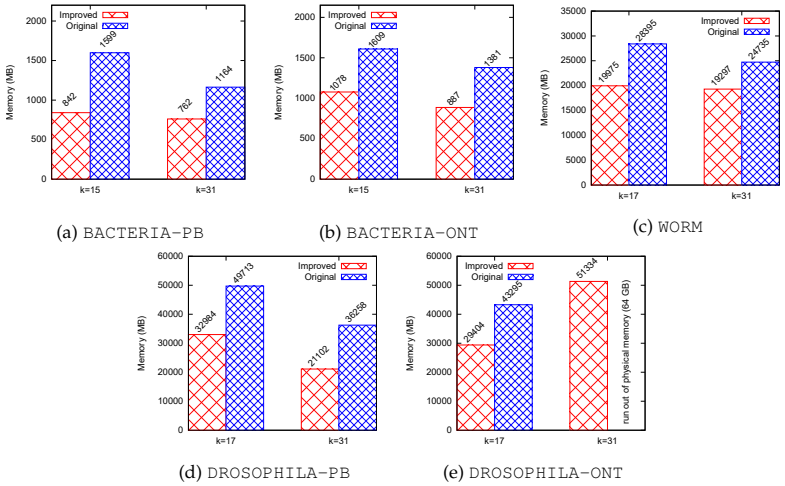
4.2.3 Analysis of underlying data structures

One question that may arise is that, during the approximate counting, if instead of using t bits (or $\log\lceil(t-1)\rceil + 1$ bits) for counters, a 1-byte counter would be faster. Moreover, is that solution scalable when the threshold increases? For answering these questions, we present an experiment varying the value of threshold t, and show the results in Figures 7–8.

12. https://github.com/PacificBiosciences/DevNet/wiki/E.-coli-20kb-Size-Selected-Library-with-P4-C2
13. http://lab.loman.net/2015/09/24/first-sqk-map-006-experiment/
14. https://github.com/PacificBiosciences/DevNet/wiki/C.-elegans-data-set
15. https://github.com/PacificBiosciences/DevNet/wiki/Drosophila-sequence-and-assembly
16. https://www.ebi.ac.uk/ena/data/view/SRR6702603

TABLE 3: Datasets used in the experiments.

Dataset	Species	Est. Gen. Size	Techn.	Cover.	Number of reads	Aver. bp	Max bp
BACTERIA-PB	*E. coli*	5 Mbp	PB	50x	48,048	8,637	41,331
BACTERIA-ONT	*E. coli*	5 Mbp	ONT	50x	50,966	9,753	57,229
WORM	*C. elegans*	100 Mbp	PB	40x	1,481,552	10,958	55,460
DROSOPHILA-PB	*D. melanogaster*	175 Mbp	PB	120x	3,227,724	9,303	44,766
DROSOPHILA-ONT	*D. melanogaster*	175 Mbp	ONT	30x	663,784	6,956	446,050



(a) BACTERIA-PB (b) BACTERIA-ONT (c) WORM

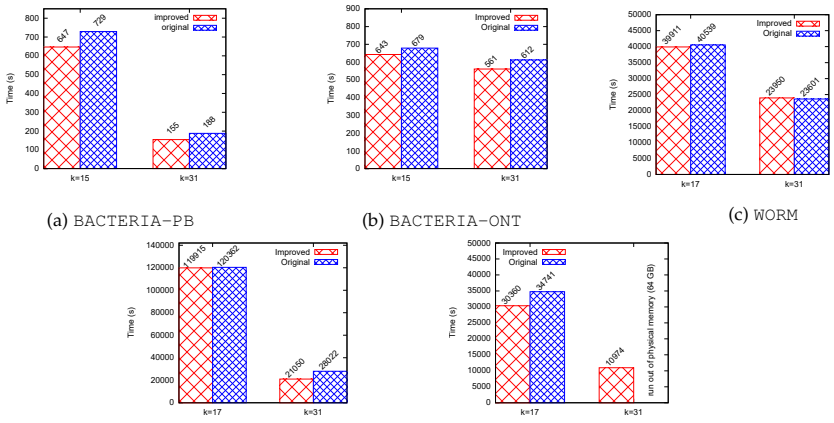
(d) DROSOPHILA-PB (e) DROSOPHILA-ONT

Fig. 4: Main memory peak (in Megabytes).

Recall that Flye chooses automatically the correct value for this threshold, ranging between 2 and 5. More specifically, Flye sets $t = 2$ for WORM and DROSOPHILA-ONT, $t = 4$ for BACTERIA-PB and BACTERIA-ONT, and $t = 5$ for DROSOPHILA-PB. For this experiment, we hard-coded the values between 2 and 5. However, we must note that, by doing this, the assembly obtained can be of a lower quality, or even Flye may not get an assembly at all.

As explained, our method for the approximate counting is based on identifying the first bitmap with a 0-bit in the corresponding entry and changing its value to 1. However, instead of a sequential scan, our implementation requires constant time by using bitwise shifts. We compare our approach with that of the original Flye and also with the use of a 1-byte counter. We can see the time results for BACTERIA-PB with $k = 15$ in Figure 7(a), where a counter using 1 byte is slightly faster for the approximate counting step, but requiring much more memory, as shown in Figure 8(a). We can also observe that times for our approach remain stable, regardless of the value of t, and always below the times of the original Flye. In addition, there are no significant differences between using t-bits counters and 1-byte counters when we take into account the subsequent phases (exact counting and indexing). In terms of space, we can see in Figure 8(a) that, during the approximate counting step, the solution using t bits requires more space when using a larger t, but not as much as the 1-byte counters. In fact, the space used for the 1-byte counters only for the approximate counting is even larger than the space required by the rest of the phases of our improved version. Moreover, the 1-byte counters also require more space than the original solution of Flye for the approximate counting. In any case, the original Flye requires much more memory for the following steps. In general, we can see that the global memory consumption tends to decrease when t grows. This is due to the fact that if the threshold augments, then fewer k-mers pass the first filter, and thus this may lead to a decrease in memory consumption.

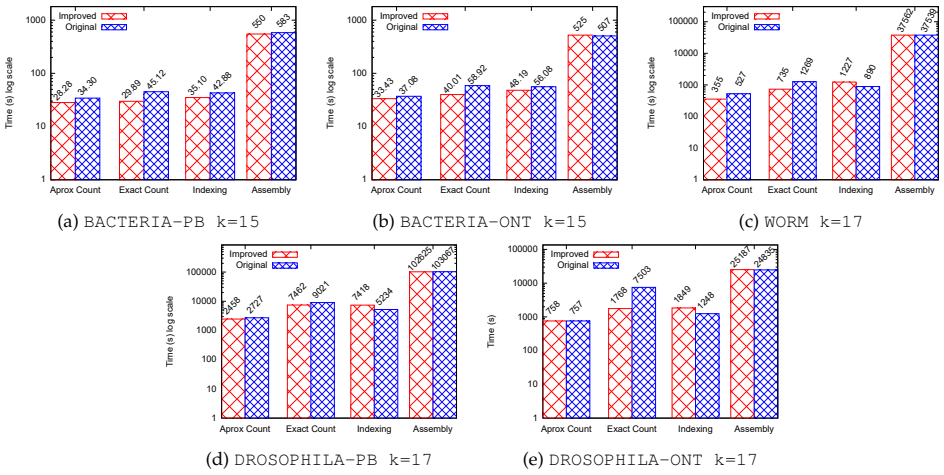
In Figures 7(b) and 8(b), we can see the same experiment for WORM dataset with $k = 17$. In this case, compact Flye uses a counter of $\log\lceil(t-1)\rceil$ bits. Times for the approximate counting remain in the same order as using 1-byte counters when t increases. This is expected, as the only difference is the use of counters using bits not aligned to bytes, and, as seen, this does not significantly affect the times. In any case, our approach is clearly faster than the original method, between 37–70% for this step. In terms of peak memory consumption, we observe the same pattern as the one described for the smallest dataset. However, for this dataset, there are no big differences when t decreases, as the number of k-mers that pass the first filter remains similar for the different values of t: there is a reduction only of 12% comparing those k-mers that pass the filter when $t = 2$ and $t = 5$ for WORM, but 83% reduction for BACTERIA-PB. Thus, we can see that our approach is not only more efficient in terms of space and time compared to the original Flye, but also more stable



(a) BACTERIA-PB (b) BACTERIA-ONT (c) WORM

(d) DROSOPHILA-PB (e) DROSOPHILA-ONT

Fig. 5: Processing time (in seconds).



(a) BACTERIA-PB k=15 (b) BACTERIA-ONT k=15 (c) WORM k=17

(d) DROSOPHILA-PB k=17 (e) DROSOPHILA-ONT k=17

Fig. 6: Processing time (in seconds with log scale) of the different phases.

when the value of t varies, as the space/time results of the original Flye are very dependant on the number of k-mers that pass the first filter.

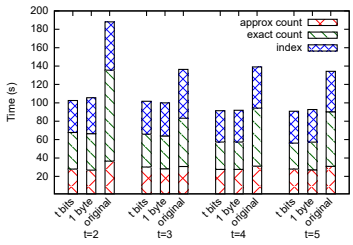
4.2.4 Energy study

Even though energy was not considered during the design of our proposal, we also measured the energy consumption of both tools. It was measured using Perf, which uses the Intel RAPL (Running Average Power Limit) energy estimates. As it can be seen in Figure 9, our proposal obtains reductions in energy consumption for all datasets, with improvements of 3–8% when $k = 15$ or $k = 17$, and reaching an improvement of 26% in the case of DROSOPHILA-PB when $k = 31$.

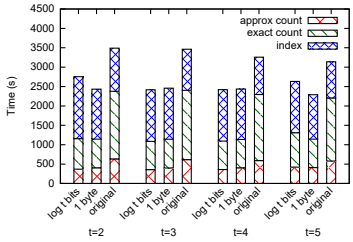
5 Discussion

As seen in the previous section, the improved version has better memory consumption. Flye allocates considerably large amounts of new memory when an insertion in the hash table, which uses a Cuckoo strategy, reaches a given number of collisions. Therefore, as the input size grows, the amount of memory needed by Flye grows much faster. With our approach, when the input datasets are huge, our growth speed is the same as for small cases. Thus, at the early stages of the process, the allocated memory grows fast, although not even close to the original Flye, but in the last stages our rhythm falls heavily.

In principle, better memory usage yields a more scalable system. A prove of this is that our improved version suc-

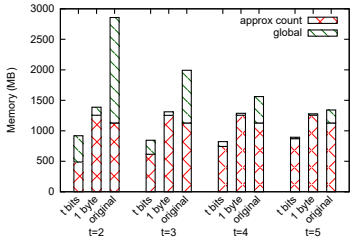


(a) BACTERIA-PB k=15

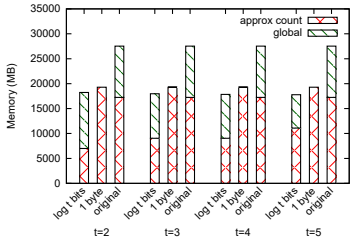


(b) WORM k=17

Fig. 7: Processing time (in seconds) of each of the steps (approximate counting, exact counting, and indexing) when using different data structures and varying the threshold value (t) for the approximate counting step.



(a) BACTERIA-PB k=15



(b) WORM k=17

Fig. 8: Peak memory consumption for the approximate counting and the complete process when using different data structures and varying the threshold value (t) for the approximate counting step.

cessfully assembled DROSOPHILA-ONT with $k = 31$ in our machine of 64 GB, whereas the original Flye was not able.

Moreover, as can be observed in Figure 6, the counting phases are faster in the new version. Observe that our enhancements are designed, in principle, to save space. Indeed, observe that in the approximate counting phase, the process is the same in both implementations, except that in our version, we do not use a hash table, but a bitmap. In the exact count and the indexing, we have an additional cost due to collisions that are kept in the same entry.

Therefore, why our version is faster? We measured the different procedures included in the exact count, and the gain is due to the insertion and update procedure of the hash table. In BACTERIA-PB, this procedure consumes 11.82 seconds in our version, whereas the original version requires 17.82; in WORM, the new version consumes 663.50 seconds versus the 1077.43 seconds of the original.

To better determine the origin of this improvement, we measured the cache references using *cachegrind*.[17] In BACTERIA-PB, our code made 157 billions references to the instructions cache, whereas the original one required 192 billions references; in WORM, the new version issued 1,840 billions references versus the 2,733 billions of the original. Moreover, our version required much fewer accesses to the data. In BACTERIA-PB, it performed 26 billions references versus the 44 billions of the original; and in WORM, 287 billions versus 717 billions. This shows that Cuckoo hash is penalized by the doubling procedures.

6 Conclusions

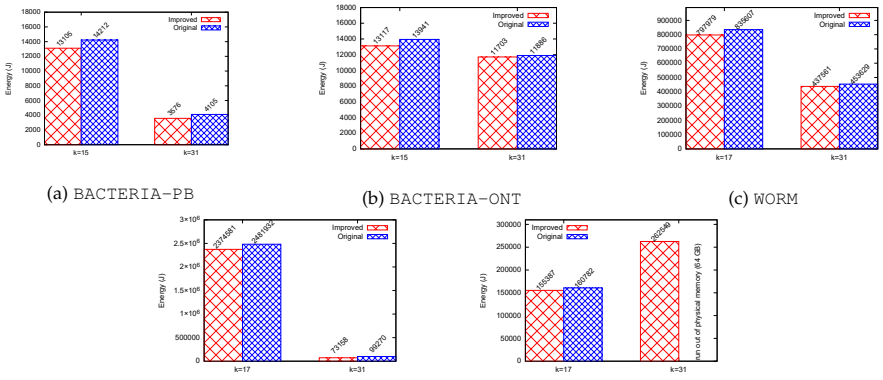
We have successfully modified the original Flye software in order to obtain a more efficient version of the same software, both in terms of space usage and execution time. The enhancements are mainly found in the k-mer counting phase, where we were able to obtain the exact same results with less memory consumption and even faster.

The improvements in memory consumption are considerable, halving the space required in most cases, and in the processing time from being on a par up to obtaining decreases of 25%. More importantly, we are able to assemble datasets that the original Flye is not able to process. In addition, as a side effect, our method saves between 3–8% of energy in general, and up to 26% for one of the experiments.

This implies a more scalable and faster software, which also requires less energy consumption. These memory-, time- and energy-efficient approaches will contribute to the advance of in-field analysis that are now becoming possible thanks to the advances on portable and real-time DNA sequencing and the appearance of affordable and portable handheld devices, such as the Oxford Nanopore's MinION and SmidgION.

As future work, we plan to further reduce the space consumption by counting, selecting and indexing the k-mers in compressed form in main memory. This is not an easy task, and traditional compressors cannot be used, as we must be able to decompress a given k-mer individually. Moreover, this problem becomes even harder, as it requires an on-line compression of the k-mers, that is, compressing

17. https://www.valgrind.org/docs/manual/cg-manual.html



(a) BACTERIA-PB (b) BACTERIA-ONT (c) WORM

(d) DROSOPHILA-PB (e) DROSOPHILA-ONT

Fig. 9: Energy consumption (in Joules).

them during the traversal of the reads, and without storing all the k-mers in main memory.

ACKNOWLEDGMENTS

This research has received funding from: the European Union's Horizon 2020 research and innovation programme under the Marie Sklodowska-Curie [grant agreement No 690941]; CITIC Research Center, funded by "Consellería de Cultura, Educación e Universidade from Xunta de Galicia", supported in an 80% through ERDF Funds, ERDF Operational Programme Galicia 2014-2020, and the remaining 20% by "Secretaría Xeral de Universidades" (Grant ED431G 2019/01); Xunta de Galicia/FEDER-UE under Grants [IG240.2020.1.185; IN852A 2018/14] and Ministerio de Ciencia e Innovación under Grants [TIN2016-78011-C4-1-R; PID2019-105221RB-C41; PID2020-114635RB-I00; FPU17/02742].

REFERENCES

[1] P. Muir, S. Li, S. Lou, D. Wang, D. J. Spakowicz, L. Salichos, J. Zhang, G. M. Weinstock, F. Isaacs, J. Rozowsky *et al.*, "The real cost of sequencing: scaling computation to keep pace with data generation," *Genome biology*, vol. 17, no. 1, p. 53, 2016.

[2] A. Sboner, X. J. Mu, D. Greenbaum, R. K. Auerbach, and M. B. Gerstein, "The real cost of sequencing: higher than you think!" *Genome biology*, vol. 12, no. 8, p. 125, 2011.

[3] H. Stevens, *Life out of sequence: a data-driven history of bioinformatics*. USA: University of Chicago Press, 2013.

[4] R. Leinonen, H. Sugawara, M. Shumway, and I. N. S. D. Collaboration, "The sequence read archive," *Nucleic acids research*, vol. 39, no. suppl_1, pp. D19–D21, 2010.

[5] Z. D. Stephens, S. Y. Lee, F. Faghri, R. H. Campbell, C. Zhai, M. J. Efron, R. Iyer, M. C. Schatz, S. Sinha, and G. E. Robinson, "Big data: Astronomical or genomical?" *PLOS Biology*, vol. 13, no. 7, pp. 1–11, 07 2015.

[6] Y. Liu, B. Schmidt, and D. L. Maskell, "Parallelized short read assembly of large genomes using de bruijn graphs," *BMC Bioinformatics*, vol. 12, no. 1, p. 354, 2011.

[7] E. Georganas, A. Buluç, J. Chapman, L. Oliker, D. Rokhsar, and K. Yelick, "Parallel de bruijn graph construction and traversal for de novo genome assembly," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC'14)*. IEEE, 2014, pp. 437–448.

[8] E. Georganas, A. Buluç, J. Chapman, S. Hofmeyr, C. Aluru, R. Egan, L. Oliker, D. Rokhsar, and K. Yelick, "Hipmer: An extreme-scale de novo genome assembler," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC '15)*. ACM, 2015, pp. 14:1–14:11.

[9] J. Meng, B. Wang, Y. Wei, S. Feng, and P. Balaji, "Swap-assembler: scalable and efficient genome assembly towards thousands of cores," *BMC Bioinformatics*, vol. 15, no. 9, p. S2, 2014.

[10] C. Gamboa-Venegas and E. Meneses, "Comparative analysis of de bruijn graph parallel genome assemblers," in *2018 IEEE International Work Conference on Bioinspired Intelligence (IWOBI)*. IEEE, 2018, pp. 1–8.

[11] D. Kleftogiannis, P. Kalnis, and V. B. Bajic, "Comparing memory-efficient genome assemblers on stand-alone and cloud infrastructures," *PloS one*, vol. 8, no. 9, p. e75505, 2013.

[12] R. Chikhi, A. Limasset, and P. Medvedev, "Compacting de bruijn graphs from sequencing data quickly and in low memory," *Bioinformatics*, vol. 32, no. 12, pp. i201–i208, 2016.

[13] N. R. Brisaboa, R. Cao, J. R. Paramá, and F. Silva-Coira, "Scalable processing and autocovariance computation of big functional data," *Software: Practice and Experience*, pp. 123–140, 2018.

[14] H. Plattner and A. Zeier, *In-memory data management: technology and applications*. Springer, 2012.

[15] G. Jacobson, "Succinct static data structures," Ph.D. dissertation, Carnegie-Mellon University, Pittsburgh, PA, Jan. 1989, tech Rep CMU-CS-89-112.

[16] G. Navarro, *Compact Data Structures – A practical approach*. New York, NY: Cambridge University Press, 2016.

[17] R. González, S. Grabowski, V. Mäkinen, and G. Navarro, "Practical implementation of rank and select queries," in *Poster Proc. Volume of 4th Workshop on Efficient and Experimental Algorithms (WEA)*, 2005, pp. 27–38.

[18] Y. Lin, J. Yuan, M. Kolmogorov, M. W. Shen, M. Chaisson, and P. A. Pevzner, "Assembly of long error-prone reads using de bruijn graphs," *Proceedings of the National Academy of Sciences*, vol. 113, no. 52, pp. E8396–E8405, 2016.

[19] M. Kolmogorov, J. Yuan, Y. Lin, and P. A. Pevzner, "Assembly of long, error-prone reads using repeat graphs," *Nature biotechnology*, vol. 37, no. 5, pp. 540–546, 2019.

[20] A. D. Tyler, L. Mataseje, C. J. Urfano, L. Schmidt, K. S. Antonation, M. R. Mulvey, and C. R. Corbett, "Evaluation of oxford nanopore's minion sequencing device for microbial whole genome sequencing applications," *Scientific Reports*, vol. 8, no. 1, 07 2018, 11907.

[21] L. E. Kafetzopoulou, S. T. Pullan, P. Lemey, M. A. Suchard, D. U. Ehichioya, M. Pahlmann, A. Thielebein, J. Hinzmann *et al.*, "Metagenomic sequencing at the epicenter of the Nigeria 2018 Lassa fever outbreak," *Science*, vol. 363, no. 6422, pp. 74–77, 2019.

[22] R. M. Idury and M. S. Waterman, "A new algorithm for dna sequence assembly," *Journal of Computational Biology*, vol. 2, no. 2, pp. 291–306, 1995, pMID: 7497130.

[23] J. D. Kececioglu and E. W. Myers, "Combinatorial algorithms for dna sequence assembly," *Algorithmica*, vol. 13, no. 1, p. 7, Feb 1995.
[24] Z. Li, Y. Chen, D. Mu, J. Yuan, Y. Shi, H. Zhang, J. Gan, N. Li, X. Hu, B. Liu, B. Yang, and W. Fan, "Comparison of the two major classes of assembly algorithms: overlap-layout-consensus and de-bruijn-graph," *Briefings in Functional Genomics*, vol. 11, no. 1, pp. 25–37, 2012.
[25] P. A. Pevzner, H. Tang, and M. S. Waterman, "An eulerian path approach to dna fragment assembly," *Proceedings of the National Academy of Sciences*, vol. 98, no. 17, pp. 9748–9753, 2001.
[26] B. G. Jackson, P. S. Schnable, and S. Aluru, "Parallel short sequence assembly of transcriptomes," *BMC Bioinformatics*, vol. 10, no. 1, p. S14, 2009.
[27] A. I. Tomescu and P. Medvedev, "Safe and complete contig assembly through omnitigs," *Journal of Computational Biology*, vol. 24, no. 6, pp. 590–602, 2017.
[28] E. W. Myers, "The fragment assembly string graph," *Bioinformatics*, vol. 21, no. suppl_2, pp. ii79–ii85, 2005.
[29] P. A. Pevzner, H. Tang, and G. Tesler, "De novo repeat classification and fragment assembly," *Genome research*, vol. 14, no. 9, pp. 1786–1796, 2004.
[30] P. Ferragina and G. Manzini, "Indexing compressed text," *J. ACM*, vol. 52, no. 4, p. 552–581, Jul. 2005.
[31] B. Langmead, C. Trapnell, M. Pop, and S. L. Salzberg, "Ultrafast and memory-efficient alignment of short dna sequences to the human genome," *Genome biology*, vol. 10, no. 3, p. R25, 2009.
[32] H. Li and R. Durbin, "Fast and accurate short read alignment with burrows–wheeler transform," *Bioinformatics*, vol. 25, no. 14, pp. 1754–1760, 2009.
[33] R. Li, C. Yu, Y. Li, T.-W. Lam, S.-M. Yiu, K. Kristiansen, and J. Wang, "Soap2: an improved ultrafast tool for short read alignment," *Bioinformatics*, vol. 25, no. 15, pp. 1966–1967, 2009.
[34] N. Välimäki and E. Rivals, "Scalable and versatile k-mer indexing for high-throughput sequencing data," in *Bioinformatics Research and Applications*. Springer Berlin Heidelberg, 2013, pp. 237–248.
[35] K. Sadakane, "New text indexing functionalities of the compressed suffix arrays," *Journal of Algorithms*, vol. 48, no. 2, pp. 294–313, 2003.
[36] F. Claude, A. Fariña, M. Martínez Prieto, and G. Navarro, "Compressed q-gram indexing for highly repetitive biological sequences," in *Proceedings of the 10th Int. Conf. on Bioinformatics and Bioengineering (BIBE)*, 2010, pp. 86–91.
[37] T. C. Conway and A. J. Bromage, "Succinct data structures for assembling large genomes," *Bioinformatics*, vol. 27, no. 4, pp. 479–486, 01 2011.
[38] A. Bowe, T. Onodera, K. Sadakane, and T. Shibuya, "Succinct de bruijn graphs," in *Algorithms in Bioinformatics*. Springer Berlin Heidelberg, 2012, pp. 225–235.
[39] R. Chikhi, A. Limasset, S. Jackman, J. T. Simpson, and P. Medvedev, "On the representation of de bruijn graphs," in *Research in Computational Molecular Biology*, R. Sharan, Ed. Cham: Springer International Publishing, 2014, pp. 35–55.
[40] E. A. Rødland, "Compact representation of k-mer de bruijn graphs for genome read assembly," *BMC Bioinformatics*, vol. 14, no. 1, p. 313, 2013.
[41] J. He, H. Yan, and T. Suel, "Compact full-text indexing of versioned document collections," in *Proceedings of the 18th ACM conference on Information and knowledge management*, 2009, pp. 415–424.
[42] J. He, J. Zeng, and T. Suel, "Improved index compression techniques for versioned document collections," in *Proceedings of the 19th ACM international conference on Information and knowledge management*, 2010, pp. 1239–1248.
[43] S. Kuruppu, S. J. Puglisi, and J. Zobel, "Relative lempel-ziv compression of genomes for large-scale storage and retrieval," in *Proceedings of the 17th International Symposium on String Processing and Information Retrieval (SPIRE)*, 2010, pp. 201–206.
[44] G. Navarro and V. Sepúlveda, "Practical indexing of repetitive collections using relative lempel-ziv," in *2019 Data Compression Conference (DCC)*, 2019, pp. 201–210.
[45] S. Deorowicz and S. Grabowski, "Robust relative compression of genomes with random access," *Bioinformatics*, vol. 27, no. 21, pp. 2979–2986, 2011.
[46] S. Kreft and G. Navarro, "On compressing and indexing repetitive sequences," *Theoretical Computer Science*, vol. 483, pp. 115–133, 2013.
[47] S. Kuruppu, S. J. Puglisi, and J. Zobel, "Optimized relative lempel-ziv compression of genomes," in *Proceedings of the 34th Australasian Computer Science Conference*, 2011, pp. 91–98.
[48] T. Gagie, P. Gawrychowski, J. Kärkkäinen, Y. Nekrich, and S. J. Puglisi, "A faster grammar-based self-index," in *International Conference on Language and Automata Theory and Applications*. Springer, 2012, pp. 240–251.
[49] G. Jacobson, "Space-efficient static trees and graphs," in *Proceedings of the IEEE Symposium on Foundations of Computer Science (FOCS)*, 1989, pp. 549–554.
[50] Raman, Raman, and Rao, "Succinct indexable dictionaries with applications to encoding k-ary trees and multisets," in *Proceedings Symposium on Discrete Algorithms (SODA)*, 2002.
[51] D. Clark, "Compact pat trees," Ph.D. dissertation, University of Waterloo, 1997.
[52] M. Kokot, M. Długosz, and S. Deorowicz, "Kmc 3: counting and manipulating k-mer statistics," *Bioinformatics*, vol. 33, no. 17, pp. 2759–2761, 2017.
[53] G. Rizk, D. Lavenier, and R. Chikhi, "Dsk: k-mer counting with very low memory usage," *Bioinformatics*, vol. 29, no. 5, pp. 652–653, 2013.
[54] P. Melsted and J. K. Pritchard, "Efficient counting of k-mers in dna sequences using a bloom filter," *BMC Bioinformatics*, vol. 12, no. 1, p. 333, 2011.
[55] B. H. Bloom, "Space/time trade-offs in hash coding with allowable errors," *Commun. ACM*, vol. 13, no. 7, p. 422–426, Jul. 1970.
[56] R. Pagh and F. F. Rodler, "Cuckoo hashing," *Journal of Algorithms*, vol. 51, no. 2, pp. 122 – 144, 2004.
[57] H. Li, "Minimap2: pairwise alignment for nucleotide sequences," *Bioinformatics*, vol. 34, no. 18, pp. 3094–3100, 05 2018.

Borja Freire received his bachelor degree in Computer Science at the University of A Coruña in 2016 and master degree in Bioinformatics at the same university in 2018. He is now a PhD student of the Doctorate Program in Computer Science at University of A Coruña, and he has been awarded a FPU fellowship to complete his doctorate.

Susana Ladra received the bachelor's degree in mathematics from the National Distance Education University (UNED), in 2014, and the master's in computer science engineering and the Ph.D. degree in computer science from the University of A Coruña, in 2007 and 2011, respectively. She is currently an Associate Professor with the Universidade da Coruña. She is the Principal Investigator of several national and international research projects. She has published more than 40 articles in various international journals and conferences. Her research interests include design and analysis of algorithms and data structures, and data compression and data mining in the fields of information retrieval and bioinformatics.

José R. Paramá has PhD in computer science from the University of A Coruña. Since 1997 he is a professor at the University of A Coruña, and since 2008, Associate Professor. He has participated in more than twenty research projects funded by European, regional and national administrations, and in more than thirty R&D contracts. He is the author of more than thirty scientific publications and more than sixty scientific conferences.

Appendix A

Resumen del trabajo realizado

En este capítulo se presenta un resumen del trabajo realizado durante la tesis. En la sección A.1 se presenta una breve introducción y la motivación para la realización de esta tesis. Además, se introducen brevemente los temas de trabajo que se han desarrollado durante la tesis. En la sección A.2 se introducen cada uno de los algoritmos desarrollados para los distintos problemas tratados. En la sección B.3 se presentan las conclusiones a las que se llegaron tras el desarrollo de la tesis. Finalmente, este capítulo se cierra con la sección B.4, donde se abordan diferentes líneas de investigación para mejorar y ampliar en un futuro nuestras contribuciones aquí expuestas.

A.1 Introducción

Durante los últimos veinte años la biología ha vivido una de sus etapas de máximo esplendor gracias a la creación de las máquinas de secuenciación. Dichas máquinas permitieron paralelizar el proceso de obtención de secuencias biológicas produciendo las conocidas como lecturas de nueva generación, NGS del inglés Next Generation Sequencing, o lecturas de segunda generación. Como es evidente esto permitiendo que empezarán a producirse de manera masiva lecturas puesto que era más barato y rápido que los antiguos métodos de Sanger o de descomposición química. A partir de disciplinas como las ciencias de computación entraron al procesamiento de estas secuencias desde múltiples vertientes distintas: algoritmia, inteligencia artificial, matemáticas, etc. La que inicialmente tuvo mayor aceptación y relevancia puesto que permitía resolver problemas hasta el momento prácticamente impensables fue la algoritmia y el reto principal eran los algoritmos de ensamblaje. Dentro de este campo surgió el interés por ensamblar genomas de organismos procariotas para

posteriormente saltar a grandes organismos eucariotas. Hoy día tenemos genomas ensamblados de prácticamente cualquier especie conocida. Sin embargo, desde el momento de su instauración ciertos temas se han resistido y no tiene aún hoy una solución y son problemas abiertos. Algunos de los retos abiertos más populares en este campo son aquellos que implican algoritmos de ensamblaje múltiple como: reconstrucción de haplotipos, transcriptómica o meta transcriptómica, entre otros. Dado que es imposible abarcarlos todos en esta tesis nos hemos centrado en la reconstrucción de haplotipos, y de manera más precisa en la reconstrucción de haplotipos víricos. El interés del estudio de haplotipos, y particularmente los víricos, viene de que los virus están formados íntegramente por información codificante y de que estos presentan un ratio de mutación superior al resto de organismos. La primera de las propiedades supone un problema puesto que cualquier mutación afectará a una proteína funcional bien dejándola igual, mutación silenciosa, o cambiándola, y por tanto afectando a su función. En el segundo caso puede la mutación puede acabar con la función de la proteína y por tanto afectar a la supervivencia del virus, cambio positivo en caso de ser un virus dañino, o por el contrario puede dotar a dicho virus de mayores niveles de: agresividad, resistencia, replicación, contagio, etc. Una vez aclarado esto es evidente que si el ratio de mutación es elevado solo agrava la situación haciendo que la probabilidad de que se incremente alguna propiedad perniciosa se incremente. Es por esto que es de vital importancia desarrollar métodos para la reconstrucción lo más precisa posible de los haplotipos contenidos en una muestra dada. Además, estos métodos deben ser eficientes en tiempo y espacio aprovechando las características propias de los virus, como es su pequeño tamaño. Todos los problemas de reconstrucción de haplotipos suelen presentan un escenario similar y un objetivo parejo. Sin embargo, suelen diferenciarse tanto en las hipótesis a priori y en las restricciones finales del problema. De manera general el problema se podría enunciar como: “Dada una muestra o muestras S, que contienen un número de genomas G desconocidos cada uno de estos con una proporción relativa P_i. Obtener dichos G y dichas P_i.”. En nuestro caso, como ya se comentó previamente, nos centramos en la reconstrucción de haplotipos víricos. En este caso las muestras suelen ser más profundas de lo habituales, es decir se hacen más pasadas por cada sección del genoma del virus, estas solo contienen el virus y sus variantes, y el genoma de todos los virus está contenido en dicha muestra. A mayores los niveles de exigencia suelen ser altos puesto que se pide llevar a niveles de similitud inferiores al 1% e incluso recuperar variantes con proporciones inferiores nuevamente al 1%.

Al margen de las lecturas NGS durante la última década surgió un nuevo tipo de lecturas de secuenciación conocidas como lecturas de tercera generación, o TGS por sus siglas en inglés Third Generation Sequencing. A diferencia de las NGS estas eran mucho más largos y erróneas. Inicialmente supuso un problema puesto que su alto nivel de errores hacía impracticables la mayoría de soluciones ideadas para NGS. Sin embargo, su longitud las hacía ideales para el tratamiento de repeticiones en tándem puesto que son capaces de cubrirlas íntegramente. Estas propiedades

han promovido que durante los últimos años mucha investigación se haya centrado en la definición de técnicas de corrección de estas lecturas, así como en adaptar soluciones previas de NGS. En la actualidad, ya contamos con ensambladores de TGS como Flye, análogo al clásico ensamblador SPAdes de NGS, e incluso están surgiendo aproximaciones de reconstrucción de haplotipos basados íntegramente en lecturas de tercera generación.

A.2 Objetivos

Durante esta tesis nos hemos centrado en dos temas fundamentalmente: la reconstrucción de haplotipos víricos y el ensamblaje de lecturas de tercera generación. En el primer caso, el objetivo era desarrollar una solución íntegramente basada en grafos de Bruijn que fuese competitiva en resultados con las aproximaciones existentes y que conservase las propiedades de estos grafos. En el segundo, nuestro objetivo era la optimización en tiempo o/y en espacio de los procesos de ensamblaje de tercera generación. El objetivo último era hacer viable la ejecución de Flye, ensamblador de tercera generación, en dispositivos de uso personal. En esta sección se describen brevemente las aportaciones que se hizo en cada uno de estos campos.

A.2.1 Reconstrucción de haplotipos víricos

Al comienzo de la tesis la reconstrucción de haplotipos víricos era un tema cuyas soluciones estaban todas basadas en el uso de referencias o en el uso de grafos de solapes. Si bien el uso de referencia está totalmente justificado y es viable en ciertas ocasiones, no lo es siempre. De hecho, para casos de infecciones largas o donde la cepa del virus es desconocida las técnicas basadas en referencia sueles producir sesgos graves en los ensamblajes. Por otro lado, las técnicas basadas en grafos de solapes si que presentaban buenos resultados. Pero como toda técnica basada en este tipo de grafos necesitan realizar el alineamiento por pares de todas las lecturas de secuenciación, y esto en muchos casos es impracticable. Por lo tanto, los resultados en ese punto o eran imprecisos o eran demasiado lentos. Por lo que una solución con resultados competitivos en precisión pero que ofreciese un beneficio en tiempo podría ser útil. A la hora de realizar ensamblajes generales suelen usarse dos tipos de aproximaciones basadas en grafos: las overlap layout consesus, OLC, o las de de Bruijn. Las segundas a diferencia de las primeras no escalan con el número de lecturas sino que típicamente lo hacen con el tamaño del genoma. De modo que son, casi siempre, varios órdenes de magnitud más rápidos. Sin embargo, a costa de esta velocidad suelen perder algo de precisión puesto que pasan de trabajar con las lecturas completas a trabajar con ngrams, conocidos como k-mers. Debido a que sabíamos de la existencia de este trade-off eficiencia-tiempo decidimos enfocar el problema de la reconstrucción de haplotipos a través de grafos de de Bruijn. Si bien existía alguna opción la mayoría no ofrecían resultados comparables a los grafos

de solape y además perdían las propiedades de eficiencia por usar procesamientos demasiado complejos. Como resultado conseguimos desarrollar dos herramientas diferentes:

- La primera de las herramientas desarrolladas ha sido viaDBG. viaDBG es la primera herramienta basada en grafos de De Bruijn competitiva en resultados con sus análogas basadas en grafos de solape. Sin embargo, frente a estas presenta que mantiene las propiedades de los grafos de de Bruijn y se presenta varios órdenes de magnitud más rápida que estas. La idea detrás de viaDBG sigue la misma línea que las herramientas desarrolladas hasta aquel momento y era la enumeración de cliques. En la mayoría de aproximaciones esta enumeración se hacía o sobre el propio grafo de solapes (HaploClique) o sobre versiones reducidas de este, PeHaplo o SAVAGE, lo que hacía que dicha enumeración implicase nuevamente un coste alto. Sin embargo, viaDBG usaba la enumeración de cliques sobre grafos construidos para cada par de nodos adyacentes a partir de su información de emparejados. Por lo tanto, se beneficiaba de la bondad de los cliques para la identificación de haplotipos. Pero no tenía penalización de tamaño pues los grafos que construía eran típicamente pequeños. A mayores de esto, viaDBG era una herramienta íntegramente basada en grafos de De Bruijn por lo que su construcción, pulido y compactación eran muy eficientes. Pese a que viaDBG en líneas generales funcionaba bien tenía tres problemas importantes:

  - Presentaba una fragmentación del ensamblaje en ocasiones mayor que el resto de herramientas. Esto se puede observar en valores lígeramente inferiores de la métrica N50. Esto a nivel biológico se traduce en unas reconstrucciones parciales de los genomas y no en un full-haplotype reconstruction.
  - Era incapaz de realizar estimaciones de las frecuencias relativas de cada una de las hebras reconstruídas.
  - Presentaba un ratio de duplicación más elevado de 2 en alguna ocasión, lo que sugería que en ocasiones sobreestimaba el número de haplotipos en la muestra.

- La segunda herramienta desarrollada ha sido ViQUF. Esta originalmente se pensó como una versión 2.0 de viaDBG puesto que ambas estaban basadas en grafos de De Bruijn y buscaban resolver el mismo problema. Sin embargo, ViQUF presentaba finalmente tantas mejoras y cambios respecto a viaDBG que finalmente decidimos tratarla como una herramienta nueva. Si bien los objetivos, inputs y grafo inicial son los mismos, la lógica cambia totalmente con respecto a viaDBG. ViQUF está basado en el grafo de ensamblaje, o versión compacta del grafo de De Bruijn, por lo que sus nodos no son k-mers sino unitigs. Esto representó otra complejidad añadida puesto que la

información de emparejados ya no iba asociada a un k-mer sino a un unitig y casuísticas como que un unitig se emparejase consigo mismo podían ocurrir. Sin embargo, el mayor cambio vino a la hora de realizar la inferencia de los haplotipos y estimar las frecuencias relativas de estos. Como recordatorio, viaDBG se basaba en construir un grafo no dirigido, un grafo de alcance, y calcular cliques en este para inferir el número de haplotipos. En lugar de esto, ViQUF construye un DAG, grafo acíclico dirigido, preservando los valores de abundancias de los nodos y aristas, que previamente ha calculado, traduce este grafo en una red de flujo (asociando demandas a los nodos, capacidades y costes a las aristas) calculando apropiadamente las capacidades en función del grafo original y asociando funciones de coste convexas para asegurar la resolución lineal. Finalmente, resuelve sobre cada uno de estos DAGs un problema conocido como el máximo flujo con el mínimo coste y descompone heurísticamente el flujo calculado en caminos. A diferencia de viaDBG esta metodología es más formal (mayor inteligibilidad), produce menos caminos espurios (menor fragmentación), el flujo asignado a un camino sirve de proxy de su abundancia real y es sustancialmente más rápido. Como resultado final obtuvimos una herramienta que nuevamente superaba en tiempos a su predecesora y mejoraba los resultados de viaDBG y competía con las versiones más modernas de SAVAGE, VG-Flow y Virus-VG.

Es importante remarcar que ambas herramientas viaDBG y ViQUF se basan en grafos de De Bruijn que previamente en este contexto solo había sido usado de manera satisfactoria por MLEHaplo. Sin embargo, esta usa el grafo de De Bruijn como base para generar un conjunto de caminos candidatos. Posteriormente estima la verosimilitud de este conjunto de caminos candidatos y determina la siguiente población. Además, MLEHaplo se basa en una búsqueda aleatoria de soluciones a través de la enumeración de caminos en un grafo. Por tanto, sus tiempos son en la mayoría de casos superiores a las aproximaciones basadas en grafos de solape. De este modo podemos decir que viaDBG y ViQUF representan las primeras aproximaciones competitivas para la reconstrucción de haplotipos víricos basadas íntegramente en grafos de De Bruijn.

A.2.2 Ensamblaje de tercera generación compacto

Las dos últimas décadas, han supuesto avances increíbles en el campo de la biología computacional o biotecnología. La aparición de las NGS y posteriormente las TGS supuso un cambio mayúsculo en como conocer la información genética de los distintos organismos. De hecho, las técnicas desarrolladas basadas en NGS han permitido ensamblar cientos de miles de genomas de diferentes virus, bacterias, animales y humanos a lo largo de estos veinte años. Sin embargo, siempre con las NGS no eran capaces de resolver ciertas secciones del genoma como son las secciones repetitivas. Dichas secciones pueden componerse de pequeñas secuencias que se

repiten en tándem, las NGS no pueden capturar secuencias repetitivas muy largas, o simplemente secuencias repetitivas largas. Como solución a este problema surgieron las lecturas de tercera generación caracterizadas por una gran longitud y tasa de errores elevadas. Su longitud las hace ideales para el tratamiento de estas secciones puesto que una sola lectura puede cubrirlas íntegramente. Desafortunadamente, su gran tasa de error hace que su manejo sea complejo, y que procesos de corrección previos sean necesarios.

Originalmente, las técnicas de manejo corrección o guiado de los ensamblajes basadas en lecturas TGS eran de carácter híbrido mezclando tanto lecturas NGS y TGS. Sin embargo, esto implicaba que para una muestra dada se tenían que realizar procesos de secuenciación separados con ambas tecnologías. Esto no era económicamente rentable puesto que la secuenciación pese a ser más barata que antaño sigue siendo cara. Además, las lecturas TGS eran inservibles para el resto de herramientas y técnicas disponibles puesto que estas no contemplaban las lecturas de tercera generación. Esta entre otras razones ha motivado la aparición de aproximaciones totalmente basadas en TGS.

En el ensamblaje NGS uno de los ensambladores más populares es SPAdes. SPAdes es un ensamblador desarrollado en la Universidad de St Petesburgo por el grupo de algoritmia biotecnológica. SPAdes está basado en el concepto de assembly graph, basado a su vez en el grafo de de Bruijn, y este presenta múltiples versiones dependiendo del problema a resolver: metaSPAdes para metagenómica, rnaSPAdes para transcriptómica, hybridSPAdes para ensamblaje híbrido entre secuencias NGS y TGS, plasmidSPAdes para el ensamblaje de plásmidos, o por último biosyntheticSPAdes para la reconstrucción de clusters de genes. Del mismo laboratorio y bajo la misma supervisión se desarrolló Flye la que hoy es y pretende ser en el futuro la herramienta principal de procesamiento de lecturas de tercera generación a nivel mundial. Flye como en el caso de SPAdes está basado en el grafo de ensamblaje y usa reducciones del grafo similares para a través de un recorrido finalmente producir el conjunto de contigs óptimo. Sin embargo, SPAdes usa, cuando es necesario, herramientas de corrección de errores como BayesHammer, IonHammer para la eliminación de información no genómica. En que en el caso Flye la corrección de errores no se puede hacer con estas herramientas puesto que no están adaptadas a TGS. Por tanto, en Flye para la clasificación de información como genómica o no genómica se han de usar mecanismos ad-hoc. Para resolver esto Flye maneja dos estructuras de datos conocidas como filtros bloom y cuckoo que hacen un conteo inexacto y exacto de los k-mers respectivamente. Estas dos estructuras pese a estar optimizadas requerir de:

- Grandes usos de memoria, por ejemplo la estructura Bloom implementada en Flye usa siempre 1 o 16Gb de memoria, pero no valores intermedios.
- Escalado no continuo presentando saltos o incrementos, por ejemplo la estructura cuckoo dobla su tamaño cada vez que un número N de colisiones ocurren.

Nuestro trabajo en Compact-Flye presentamos una solución a ambos problemas. Presentamos una estructura de datos tipo hash con vectores de bits que permite reducir el tamaño, hacer que los incrementos de tamaño sean lineales y además competir en tiempo, llegando incluso a ser sustancialmente más rápidos en casos de k-mers de tamaños reducidos. Además, evitamos tener que reservar grandes cantidades de memoria cuando no es necesario. Como resultado final obtuvimos una herramienta capaz de resolver el ensamblaje de un genoma basándose en lecturas de tercera generación y que podía ser ejecutado en dispositivos de memoria reducida.

A.3 Discusión y conclusiones

Como se ha podido observar el tema principal de la tesis es el desarrollo de soluciones a problemas biológicos basándonos en grafos de de Bruijn. Por un lado, se han desarrollado viaDBG y ViQUF que son las primeras soluciones competitivas a la reconstrucción de haplotipos víricos basadas íntegramente en grafos de de Bruijn. Por otro lado, se ha desarrollado Compact-Flye una versión compacta del conocido ensamblador de secuencias de tercera generación Flye, nuevamente basado en grafos de de Bruijn.

En el caso de la reconstrucción de haplotipos el objetivo principal era obtener una versión eficiente en espacio y tiempo que ofreciese unos resultados comparables a las técnicas que representaban el estado del arte en aquel momento. Para ello nos basamos en un grafo teórico presentado en 2012 conocido como el grafo de Bruijn aproximado emparejado. Con estos dos trabajos, viaDBG y ViQUF, no solo hemos dado una nueva solución al problema de la reconstrucción de haplotipos. Sino que también hemos demostrado que la inclusión explícita de información de emparejados en el proceso de ensamblaje, y no como post procesado, es beneficioso. Además, hemos propuesto e implementado dos opciones de cómo realizar dicha inclusión. Esto supone un avance importante puesto que se abre la puerta a poder realizar estos procesos en ensambladores genéricos y apoyar a procesos de resolución de repeticiones.

En el caso del ensamblaje de tercera generación no hemos propuesto un nuevo método en ensamblaje o hemos mejorado la precisión de la metodología actual. Sin embargo, a través del uso ingenioso de estructuras de datos sencillas como hash y bitvectors ofrecemos una alternativa más rápida y barata al proceso de conteo de k-mers basado en cuenta inexacta, filtrado, y cuenta exacta. Todo esto, como se puede observar en los resultados obtenidos, supone que ciertos ensamblajes solo viables en computadores con cientos de gigas de memoria puedan realizarse en dispositivos de uso más cotidiano.

A.4 Trabajo futuro

En esta sección proponemos varias consideraciones que pueden ser interesantes para un futuro de cara a mejorar la precisión y el rendimiento de nuestras contribuciones. Entre ellas podemos destacar las siguientes líneas de investigación:

- Realizar un benchmarking con datos reales de las aproximaciones publicadas en la para medir el funcionamiento real de las técnicas en la actualidad. A día de hoy la mayoría de pruebas se realizar sobre un data set real pero que es antiguo y en el último año ha empezado a comentarse que es poco fiable dado que contiene más cosas de las que inicialmente se dijo que tenían.

- A nivel del estudio en grafos de De Bruijn sería necesario estudiar la posibilidad de reducir los ratios de duplicidad y el número de contigs reportados. Para eso sería necesario perfeccionar los sistemas de ajuste de flujo entre pares de nodos adyacentes y que el número de splits reales se ajustase mejor al número de hebras reales.

- Otra línea de investigación que ha surgido recientemente es la reconstrucción de haplotipos a través de lecturas de tercera generación. Si bien es un tema que en la actualidad solamente tiene una aproximación y ha sido publicada en 2022, ya hemos planteado una nueva aproximación basada en ViQUF y programación lineal y entera.

- Otro problema identificado surge de que las muestras se toman en instante $t_0 = 0$ y sin embargo las secuencias se obtienen en un instante $t_1 = t$. Una posible nueva línea de investigación es estudiar como son los procesos de difusión y transporte de los virus a lo largo del cuerpo en una infección. Una posible vía de estudio es adecuar procesos de difusión de contaminantes en un fluido, incluyendo información estocástica para simular procesos de mutación y transporte.

Bibliography

[AMO91] Ravindra K. Ahuja, Thomas L. Magnanti, and James B. Orlin. Some recent advances in network flows. *SIAM Review*, 33(2):175–219, 1991.

[BARS17] Jasmijn A. Baaijens, Amal Zine El Aabidine, Eric Rivals, and Alexander Schönhuth. De novo assembly of viral quasispecies using overlap graphs. *Genome Research*, 27:835–848, 2017.

[BAT12] Sadakane Kunihiko Bowe Alexander, Onodera Taku and Shibuya Tetsuo. Succinct de bruijn graphs. *WABI 2012*, pages 225–235, 2012.

[BCPSC18] Nieves R. Brisaboa, Ricardo Cao, José. R. Paramá, and Fernando Silva-Coira. Scalable processing and autocovariance computation of big functional data. *Software: Practice and Experience*, pages 123–140, 2018.

[Blo70] Burton H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM*, 13(7):422–426, July 1970.

[BNA+12] A. Bankevich, S. Nurk, D. Antipov, A.A. Gurevich, M. Dvorkin, A.S. Kulikov, V.M. Lesin, S.I. Nikolenko, S. Pham, A.D Prjibelski, A.V. Pyshkin, A.V. Sirotkin, N. Vyahhi, G. Tesler, M.A. Alekseyev, and P.A. Pevzner. Spades: a new genome assembly algorithm and its applications to single-cell sequencing. *Journal of Computational Biology*, 19(5):455–477, 2012.

[BSS20] Jasmijn A. Baaijens, Leen Stougie, and Alexander Schönhuth. Strain-aware assembly of genomes from mixed samples using flow variation graphs. *bioRxiv*, 2020.

[BVdRK+19] Jasmijn A Baaijens, Bastiaan Van der Roest, Johannes Köster, Leen Stougie, and Alexander Schönhuth. Full-length de novo viral quasispecies assembly through variation graph construction. *Bioinformatics*, 05 2019. btz443.

[CB73] Joep Kerboscht Coen Bron. Finding all cliques of an undirected graph. *Communications of the ACM*, 16(9), 1973.

[CB11] Thomas C. Conway and Andrew J. Bromage. Succinct data structures for assembling large genomes. *Bioinformatics*, 27(4):479–486, 01 2011.

[CCF+21] Sara Castellano, Federica Cestari, Giovanni Faglioni, Elena Tenedini, Marco Marino, Lucia Artuso, Rossella Manfredini, Mario Luppi, Tommaso Trenti, and Enrico Tagliafico. ivar, an interpretation-oriented tool to manage the update and revision of variant annotation and classification. *Genes*, 12(3), 2021.

[CFMPN10] F. Claude, A. Fariña, M.A. Martínez Prieto, and G. Navarro. Compressed q-gram indexing for highly repetitive biological sequences. In *Proceedings of the 10th Int. Conf. on Bioinformatics and Bioengineering (BIBE)*, pages 86–91, 2010.

[CK10] Mihai Pop Carl Kingsford, Michael C. Schatz. Assembly complexity of prokaryotic genomes using short reads. *BMC Bioinformatics*, 11, 2010.

[Cla97] David Clark. *Compact pat trees*. PhD thesis, University of Waterloo, 1997.

[CLJ+14] Rayan Chikhi, Antoine Limasset, Shaun Jackman, Jared T. Simpson, and Paul Medvedev. On the representation of de bruijn graphs. In Roded Sharan, editor, *Research in Computational Molecular Biology*, pages 35–55, Cham, 2014. Springer International Publishing.

[CLM16a] Rayan Chikhi, Antoine Limasset, and Paul Medvedev. Compacting de Bruijn graphs from sequencing data quickly and in low memory. *Bioinformatics*, 32(12):i201–i208, 06 2016.

[CLM16b] 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.

[CZS18] Jiao Chen, Yingchao Zhao, and Yanni Sun. *De novo* haplotype reconstruction in viral quasispecies using paired-end read guided path finding. *Bioinformatics*, 34(17):2927–2935, 2018.

[DF18] Ruofei Du and Zhide Fang. Statistical correction for functional metagenomic profiling of a microbial community with short ngs reads. *Journal of Applied Statistics*, 45(14):2521–2535, 2018. PMID: 30505061.

[DG11] Sebastian Deorowicz and Szymon Grabowski. Robust relative compression of genomes with random access. *Bioinformatics*, 27(21):2979–2986, 2011.

[DKO+84] David J DeWitt, Randy H Katz, Frank Olken, Leonard D Shapiro, Michael R Stonebraker, and David A Wood. Implementation techniques for main memory database systems. In *Proceedings of the 1984 ACM SIGMOD international conference on management of data*, pages 1–8, 1984.

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

[DSH08] Siobain Duffy, Laura A. Shackelton, and Edward C. Holmes. Rates of evolutionary change in viruses: patterns and determinants. *Nature Reviews Genetics*, 9:267–276, 2008.

[DSP12] Esteban Domingo, Julie Sheldon, and Celia Perales. Viral quasispecies evolution. *Microbiology and Molecular Biology Reviews*, 76(2):159–216, 2012.

[EGA+20] Anton Eliseev, Keylie M. Gibson, Pavel Avdeyev, Dmitry Novik, Matthew L. Bendall, Marcos Pérez-Losada, Nikita Alexeev, and Keith A. Crandall. Evaluation of haplotype callers for next-generation sequencing of viruses. *Infection, Genetics and Evolution*, 82:104277, 2020.

[EK72] Jack Edmonds and Richard M. Karp. Theoretical improvements in algorithmic efficiency for network flow problems. *J. ACM*, 19(2):248–264, apr 1972.

[FAKM14] Bin Fan, Dave G Andersen, Michael Kaminsky, and Michael D Mitzenmacher. Cuckoo filter: Practically better than bloom. In *Proceedings of the 10th ACM International on Conference on emerging Networking Experiments and Technologies*, pages 75–88. ACM, 2014.

[FF56] L. R. Ford and D. R. Fulkerson. Maximal flow through a network. *Canadian Journal of Mathematics*, 8:399–404, 1956.

[FLP21] Borja Freire, Susana Ladra, and Jose R. Parama. Memory-efficient assembly using flye. *IEEE/ACM Transactions on Computational Biology and Bioinformatics*, pages 1–1, 2021.

[FLPS21] Borja Freire, Susana Ladra, Jose Paramá, and Leena Salmela. Inference of viral quasispecies with a paired de Bruijn graph. *Bioinformatics*, 37(4):473–481, 2021.

[FLPS22] Borja Freire, Susana Ladra, José R. Paramá, and Leena Salmela. Viquf: De novo viral quasispecies reconstruction using unitig-based flow networks. *IEEE/ACM Transactions on Computational Biology and Bioinformatics*, pages 1–13, 2022.

[FM05] Paolo Ferragina and Giovanni Manzini. Indexing compressed text. *J. ACM*, 52(4):552–581, July 2005.

[GBC+14] Evangelos Georganas, Aydin Buluç, Jarrod Chapman, Leonid Oliker, Daniel Rokhsar, and Katherine Yelick. Parallel de bruijn graph construction and traversal for de novo genome assembly. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC'14)*, pages 437–448. IEEE, 2014.

[GBC+15] Evangelos Georganas, Aydin 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 (SC '15)*, pages 14:1–14:11. ACM, 2015.

[GGK+12] Travis Gagie, Paweł Gawrychowski, Juha Kärkkäinen, Yakov Nekrich, and Simon J Puglisi. A faster grammar-based self-index. In *International Conference on Language and Automata Theory and Applications*, pages 240–251. Springer, 2012.

[GGMN05] Rodrigo González, Szymon Grabowski, Veli Mäkinen, and Gonzalo Navarro. Practical implementation of rank and select queries. In *Poster Proc. Volume of 4th Workshop on Efficient and Experimental Algorithms (WEA)*, pages 27–38, 2005.

[GTR+14] F. Di Giallonardo, A. Töpfer, M. Rey, S. Prabhakaran, Y. Duport, C. Leemann C, S. Schmutz, N. K. Campbell, B. Joos, M. R. Lecca, A. Patrignani, M. Däumler, C. Beisel, P. Rusert, A. Trkola, H. F. Günthard, V. Roth, N. Beerenwinkel, and K. J. Metzner. Full-length haplotype reconstruction to infer the structure of heterogeneous virus populations. *Nucleic Acids Res*, 42(14):e115, 2014.

[GVM18] Carlos Gamboa-Venegas and Esteban Meneses. Comparative analysis of de bruijn graph parallel genome assemblers. In *2018 IEEE International Work Conference on Bioinspired Intelligence (IWOBI)*, pages 1–8. IEEE, 2018.

[Heo21] Yun Heo. Comprehensive evaluation of error-correction methodologies for genome sequencing data. *Bioinformatics*, 2021.

[HYS09] Jinru He, Hao Yan, and Torsten Suel. Compact full-text indexing of versioned document collections. In *Proceedings of the 18th ACM conference on Information and knowledge management*, pages 415–424, 2009.

[HZS10] Jinru He, Junyuan Zeng, and Torsten Suel. Improved index compression techniques for versioned document collections. In *Proceedings of the 19th ACM international conference on Information and knowledge management*, pages 1239–1248, 2010.

[IW95] Ramana M. Idury and Michael S. Waterman. A new algorithm for dna sequence assembly. *Journal of Computational Biology*, 2(2):291–306, 1995. PMID: 7497130.

[Jac89a] G. Jacobson. Space-efficient static trees and graphs. In *Proceedings of the IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 549–554, 1989.

[Jac89b] Guy Jacobson. *Succinct Static Data Structures*. PhD thesis, Carnegie-Mellon University, Pittsburgh, PA, January 1989. Tech Rep CMU-CS-89-112.

[Jon90] M.C. Jones. The performance of kernel density functions in kernel distribution function estimation. *Statistics & Probability Letters*, 9(2):129 – 132, 1990.

[JSM+15] Duleepa Jayasundara, I. Saeed, Suhinthan Maheswararajah, B.C. Chang, S.-L. Tang, and Saman K. Halgamuge. Viquas: an improved reconstruction pipeline for viral quasispecies spectra generated by next-generation sequencing. *Bioinformatics*, 31(6):886–896, 2015.

[JYP88] David S. Johnson, Mihalis Yannakakis, and Christos H. Papadimitriou. On generating all maximal independent sets. *Information Processing Letters*, 27(3):119 – 123, 1988.

[KDD17] Marek Kokot, Maciej Dlugosz, and Sebastian Deorowicz. Kmc 3: counting and manipulating k-mer statistics. *Bioinformatics*, 33(17):2759–2761, 2017.

[KKB13] Dimitrios Kleftogiannis, Panos Kalnis, and Vladimir B Bajic. Comparing memory-efficient genome assemblers on stand-alone and cloud infrastructures. *PloS one*, 8(9):e75505, 2013.

[KM16] Anna Kuosmanen and Veli Mäkinen. Evaluating approaches to find exon chains based on long reads. *bioRxiv*, 2016.

[KN13] Sebastian Kreft and Gonzalo Navarro. On compressing and indexing repetitive sequences. *Theoretical Computer Science*, 483:115–133, 2013.

[KPL+19] L. E. Kafetzopoulou, S. T. Pullan, P. Lemey, M. A. Suchard, D. U. Ehichioya, M. Pahlmann, A. Thielebein, J. Hinzmann, et al. Metagenomic sequencing at the epicenter of the Nigeria 2018 Lassa fever outbreak. *Science*, 363(6422):74–77, 2019.

[KPZ10] Shanika Kuruppu, Simon J Puglisi, and Justin Zobel. Relative lempel-ziv compression of genomes for large-scale storage and retrieval. In *Proceedings of the 17th International Symposium on String Processing and Information Retrieval (SPIRE)*, pages 201–206, 2010.

[KPZ11] Shanika Kuruppu, Simon J Puglisi, and Justin Zobel. Optimized relative lempel-ziv compression of genomes. In *Proceedings of the 34th Australasian Computer Science Conference*, pages 91–98, 2011.

[KTM+19] Sergey Knyazev, Viachaslau Tsyvina, Andrew Melnyk, Alexander Artyomenko, Tatiana Malygina, Yuri B. Porozov, Ellsworth Campbell, William M. Switzer, Pavel Skums, and Alex Zelikovsky. CliqueSNV: scalable reconstruction of intra-host viral populations from ngs reads. *bioRxiv*, 2019.

[KYLP19] Mikhail Kolmogorov, Jeffrey Yuan, Yu Lin, and Pavel A Pevzner. Assembly of long, error-prone reads using repeat graphs. *Nature biotechnology*, 37(5):540–546, 2019.

[LD09] Heng Li and Richard Durbin. Fast and accurate short read alignment with burrows–wheeler transform. *Bioinformatics*, 25(14):1754–1760, 2009.

[LD10] H. Li and R. Durbin. Fast and accurate long-read alignment with burrows-wheeler transform. *Bioinformatics*, 26(5):589–595, 2010.

[Li18] Heng Li. Minimap2: pairwise alignment for nucleotide sequences. *Bioinformatics*, 34(18):3094–3100, 05 2018.

[LSM11] Yongchao Liu, Bertil Schmidt, and Douglas L Maskell. Parallelized short read assembly of large genomes using de bruijn graphs. *BMC Bioinformatics*, 12(1):354, 2011.

[LSSC10] Rasko Leinonen, Hideaki Sugawara, Martin Shumway, and International Nucleotide Sequence Database Collaboration. The sequence read archive. *Nucleic acids research*, 39(suppl_1):D19–D21, 2010.

[LTPS09] 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):R25, 2009.

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

[LYL+09] Ruiqiang Li, Chang Yu, Yingrui Li, Tak-Wah Lam, Siu-Ming Yiu, Karsten Kristiansen, and Jun Wang. Soap2: an improved ultrafast tool for short read alignment. *Bioinformatics*, 25(15):1966–1967, 2009.

[Man99] Giovanni Manzini. *The Burrows-Wheeler Transform: Theory and Practice*. Springer Berlin Heidelberg, Berlin, Heidelberg, 1999.

[MGMB07] Paul Medvedev, Konstantinos Georgiou, Gene Myers, and Michael Brudno. *Computability of Models for Sequence Assembly*. Springer Berlin Heidelberg, Berlin, Heidelberg, 2007.

[MLL+16] Paul Muir, Shantao Li, Shaoke Lou, Daifeng Wang, Daniel J Spakowicz, Leonidas Salichos, Jing Zhang, George M Weinstock, Farren Isaacs, Joel Rozowsky, et al. The real cost of sequencing: scaling computation to keep pace with data generation. *Genome biology*, 17(1):53, 2016.

[MP11] Pall Melsted and Jonathan K Pritchard. Efficient counting of k-mers in dna sequences using a bloom filter. *BMC Bioinformatics*, 12(1):333, 2011.

[MPC+11] Paul Medvedev, Son Pham, Mark Chaisson, Glenn Tesler, and Pavel Pevzner. Paired de Bruijn graphs: a novel approach for incorporating mate pair information into genome assemblers. *Journal of Computational Biology*, 18(11):1625–1634, 2011.

[MRCWM18] Niema Moshiri, Manon Ragonnet-Cronin, Joel O Wertheim, and Siavash Mirarab. FAVITES: simultaneous simulation of transmission networks, phylogenetic trees and sequences. *Bioinformatics*, 35(11):1852–1861, 11 2018.

[MS18] Swati C Manekar and Shailesh R Sathe. A benchmark study of k-mer counting methods for high-throughput sequencing. *GigaScience*, 7(12), 10 2018. giy125.

[MSG16] A. Mikheenko, V. Saveliev, and A. Gurevich. MetaQUAST: evaluation of metagenome assemblies. *Bioinformatics*, 32(7):1088–1090, 2016.

[MWR+15] Raunaq Malhotra, Manjarl Mukhopadhyay Steven Wu, Allen Rodrigo, Mary Poss, and Raj Acharya. Maximum likelihood de novo reconstruction of viral populations using paired end sequencing data. *arXiv e-prints*, page arXiv:1502.04239, 2015.

[MWW+14] 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(9):S2, 2014.

[Mye05] Eugene W. Myers. The fragment assembly string graph. *Bioinformatics*, 21(suppl_2):ii79–ii85, 2005.

[Nav16] Gonzalo Navarro. *Compact Data Structures – A practical approach*. Cambridge University Press, New York, NY, 2016.

[NB15] Moritz Gerstung Niko Beerenwinkel, Roland F Schwarz. Cancer evolution: mathematical models and computational inference. *System biology*, 2015.

[Niu04] Tianhua Niu. Algorithms for inferring haplotypes. *Genetic Epidemiology*, 2004.

[NMKP17] S. Nurk, D. Meleshko, A. Korobeynikov, and P.A. Pevzner. metaspades: a new versatile metagenomic assembler. *Genome Research*, 27:824–834, 2017.

[Now06] Martin A. Nowak. Five rules for the evolution of cooperation. *Science*, 314(5805):1560–1563, 2006.

[NP16] Nelson Vera Nelson Pérez, Miguel Gutierrez. Computational performance assessment of k-mer counting algorithms. *Journal of Computational Biology*, 34(4), 2016.

[NS19] G. Navarro and V. Sepúlveda. Practical indexing of repetitive collections using relative lempel-ziv. In *2019 Data Compression Conference (DCC)*, pages 201–210, 2019.

[OAM93] James B. Orlin, Ravindra K. Ahuja, and Thomas L. Magnanti. *Network Flows: Theory, Algorithms, and Applications*. Prestice Hall, 1993.

[PCSB17] Susana Posada-Cespedes, David Seifert, and Niko Beerenwinkel. Recent advances in inferring viral diversity from high-throughput sequencing data. *Virus Research*, 239:17–32, 2017.

[Pla09] Hasso Plattner. A common database approach for oltp and olap using an in-memory column database. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of data*, pages 1–2, 2009.

[PM11] Jonathan K. Pritchard Pall Melsted. Inference of viral quasispecies with a paired de Bruijn graph. *BMC Bioinformatics*, 12(-):473–481, 2011.

[PPG+15] Bharath Pattabiraman, Md. Mostofa Ali Patwary, Assefaw H. Gebremedhin, Wei keng Liao, and Alok Choudhary. Fast algorithms for the maximum clique problem on massive graphs with applications to overlapping community detection. *Internet Mathematics*, 11(4–5):421–448, 2015.

[PR04] Rasmus Pagh and Flemming Friche Rodler. Cuckoo hashing. *Journal of Algorithms*, 51(2):122 – 144, 2004.

[PRZ+10] S. Prabhakaran, M. Rey, O. Zagordi, N. Beerenwinkel, and V. Roth. Hiv haplotype inference using a constraint-based dirichlet process mixture model. In *NIPS Workshop on Machine Learning in Computational Biology*, 2010.

[PTT04] Paul A Pevzner, Haixu Tang, and Glenn Tesler. De novo repeat classification and fragment assembly. *Genome research*, 14(9):1786–1796, 2004.

[PTW01] 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.

[PZ12] Hasso Plattner and Alexander Zeier. *In-memory data management: technology and applications*. Springer, 2012.

[RLC13a] Guillaume Rizk, Dominique Lavenier, and Rayan Chikhi. DSK: k-mer counting with very low memory usage. *Bioinformatics*, 29(5):652–653, 2013.

[RLC13b] Guillaume Rizk, Dominique Lavenier, and Rayan Chikhi. Dsk: k-mer counting with very low memory usage. *Bioinformatics*, 29(5):652–653, 2013.

[Rød13] Einar Andreas Rødland. Compact representation of k-mer de bruijn graphs for genome read assembly. *BMC Bioinformatics*, 14(1):313, 2013.

[RRR02] Raman, Raman, and Rao. Succinct indexable dictionaries with applications to encoding k-ary trees and multisets. In *Proceedings Symposium on Discrete Algorithms (SODA)*, 2002.

[Sad03] Kunihiko Sadakane. New text indexing functionalities of the compressed suffix arrays. *Journal of Algorithms*, 48(2):294–313, 2003.

[SB15] Dan Nettleton Sam Benidt. SimSeq: a nonparametric approach to simulation of RNA-sequence datasets. *Bioinformatics*, 31(13):2131–2140, 2015.

[SLF+15] Zachary D. Stephens, Skylar Y. Lee, Faraz Faghri, Roy H. Campbell, Chengxiang Zhai, Miles J. Efron, Ravishankar Iyer, Michael C. Schatz, Saurabh Sinha, and Gene E. Robinson. Big data: Astronomical or genomical? *PLOS Biology*, 13(7):1–11, 07 2015.

[SMG+11] Andrea Sboner, Xinmeng Jasmine Mu, Dov Greenbaum, Raymond K Auerbach, and Mark B Gerstein. The real cost of sequencing: higher than you think! *Genome biology*, 12(8):125, 2011.

[SN21] Nicholas Stoler and Anton Nekrutenko. Sequencing error profiles of Illumina sequencing instruments. *NAR Genomics and Bioinformatics*, 3(1), 03 2021.

[SR14] L. Salmela and E. Rivals. LoRDEC: accurate and efficient long read error correction. *Bioinformatics*, 30(24):3506–3514, 2014.

[Ste13] Hallam Stevens. *Life out of sequence: a data-driven history of bioinformatics*. University of Chicago Press, USA, 2013.

[TKRM13] Alexandru I. Tomescu, Anna Kuosmanen, Romeo Rizzi, and Veli Mäkinen. A novel min-cost flow method for estimating transcript expression with RNA-Seq. *BMC Bioinformatics*, 14:S15, 2013.

[TM17] Alexandru I. Tomescu and Paul Medvedev. Safe and complete contig assembly through omnitigs. *Journal of Computational Biology*, 24(6):590–602, 2017. PMID: 27749096.

[TMB+14] Armin Töpfer, Tobias Marschall, Rowena A. Bull, Fabio Luciani, Alexander Schönhuth, and Niko Beerenwinkel. Viral quasispecies assembly via maximal clique enumeration. *PLOS Computational Biology*, 10(3):e1003515, 2014.

[TMCB15] Alexandru Tomescu, Veli Mäkinen, Fabio Cunial, and Djamal Belazzougui. *Genome-Scale Algorithm Design*. Cambridge University Press; Illustrated edition, 2015.

[TMU+18] Andrea D. Tyler, Laura Mataseje, Chantel J. Urfano, Lisa Schmidt, Kym S. Antonation, Michael R. Mulvey, and Cindi R. Corbett. Evaluation of oxford nanopore's minion sequencing device for microbial whole genome sequencing applications. *Scientific Reports*, 8(1), 07 2018. 11907.

[TTT06] Etsuji Tomita, Akira Tanaka, and Haruhisa Takahashi. The worst-case time complexity for generating all maximal cliques and computational experiments. *Theoretical Computer Science*, 363(1):28 – 42, 2006. Computing and Combinatorics.

[VR13] Niko Välimäki and Eric Rivals. Scalable and versatile k-mer indexing for high-throughput sequencing data. In *Bioinformatics Research and Applications*, pages 237–248. Springer Berlin Heidelberg, 2013.

[WAB+12] Andreas Wilm, Pauline Poh Kim Aw, Denis Bertrand, Grace Hui Ting Yeo, Swee Hoe Ong, Chang Hua Wong, Chiea Chuen Khor, Rosemary Petric, Martin Lloyd Hibberd, and Niranjan Nagarajan. LoFreq: a sequence-quality aware, ultra-sensitive variant caller for uncovering cell-population heterogeneity from high-throughput sequencing datasets. *Nucleic Acids Research*, 40(22):11189–11201, 10 2012.

[WH15] Qinghua Wu and Jin-Kao Hao. A review on algorithms for maximum clique problems. *European Journal of Operational Research*, 242(3):693–709, 2015.

[WL22] Mumey B. Williams L, Tomescu AII. Flow decomposition with subpath constraints. *IEEE/ACM Trans Comput Biol Bioinform*, 2022.

[ZBEB11] Osvaldo Zagordi, Arnab Bhattacharya, Nicholas Eriksson, and Niko Beerenwinkel. Shorah: estimating the genetic diversity of a mixed sample from next-generation sequencing data. *BMC Bioinformatics*, 12:119, 2011.

[ZO12] Beisel C Zagordi O, Däumer M. Read length versus depth of coverage for viral quasispecies reconstruction. *PLoS ONE*, 2012.