Pontifícia Universidade Católica
do Rio de Janeiro

Elvismary Molina de Armas

# A novel approach for de Bruijn Graph construction in de novo genome fragment assembly

Tese de Doutorado

Thesis presented to the Programa de Pós–graduação em Informática of PUC-Rio in partial fulfillment of the requirements for the degree of Doutor em Ciências - Informática.

Advisor: Prof. Sérgio Lifschitz

Rio de Janeiro
September 2019

Pontifícia Universidade Católica do Rio de Janeiro

**Elvismary Molina de Armas**

# A novel approach for de Bruijn Graph construction in de novo genome fragment assembly

Thesis presented to the Programa de Pós–graduação em Informática of PUC-Rio in partial fulfillment of the requirements for the degree of Doutor em Ciências - Informática. Approved by the Examination Committee.

Rio de Janeiro, September 23rd, 2019

**Elvismary Molina de Armas**

Graduation in Engineering in Computer Science in 2008 from the University of Informatics Sciences, Cuba. In 2011 the author received his degree of Master of Science in Applied Informatics from the University of Informatics Sciences. Receives the Award of the Academy of Sciences of Cuba 2011, titled New models for the prediction of mutations and drug resistance of viruses such as HIV and influenza, as a part group of other authors, Academy of Sciences of Cuba in 2011. In 2012 receives the Award of the Rector of the University of Informatics Sciences 2012, Category: Results that Reflect the Advancement of Science, Technology, and Innovative of Major Significance and Originality, Rector of the University of Informatics Sciences. Has experience in Computer Science, focusing on Bioinformatics and Data Management.

Bibliographic data

de Armas, Elvismary Molina

A novel approach for de Bruijn Graph construction in de novo genome fragment assembly / Elvismary Molina de Armas; advisor: Prof. Sérgio Lifschitz. – Rio de janeiro: PUC-Rio , Departamento de Informática, 2019.

v., 104 f: il. color. ; 30 cm

Tese (doutorado) - Pontifícia Universidade Católica do Rio de Janeiro, Departamento de Informática.

Inclui bibliografia

1. Informática – Teses. 2. grafo de Bruijn ;. 3. montagem de genoma;. 4. k-mer. I. Lifschitz, Sérgio. II. Pontifícia Universidade Católica do Rio de Janeiro. Departamento de Informática. III. Título.

CDD:004

# Acknowledgments

First of all, I am really grateful for my family and friends. I am especially grateful to my mother, my father and my husband, who have always been supporting all the decisions in my life, and gave me a lot of encouragement and emotional and practical support in these last years to be able to finish my PhD. An extraordinary thanks to my daughter, for having come to the world in this last period of my doctorate, making me grow more every day with her smile and love.

Personal and professional thanks to my advisor, Sergio Lifschitz, who guided my research for these almost 6 years and always believed in me.

I acknowledge the significant support I received from Clícia Gravitol, Paulo Cavalcanti Gomes and the other members of the Instituto de Bioquimica Medica da UFRJ that started me in the world of fragment genome assembly, and from the jury, Prof. Edward Hermann Haeusler, Prof. Marcus Vinicius Soledade Poggi de Aragao, Prof. Nalvo Franco de Almeida Junior and Prof. Daniel Cardoso Moraes de Oliveira. Your attention and interest were crucial to me.

Many thanks to BioBD students, and PhD student colleagues in particular to Julio, Alain, Rafael, Liester, Guilherme, Sonia, and Grettel, for listening to me, criticizing, and sharing hard and good times.

Thank you very much to the Tecgraf Institute and my colleagues there, for the support and the opportunity to work part-time during the last period of my PhD.

I am thankful for receiving this opportunity to study at PUC, and now, having the grace to become a PhD and going abroad as a scientist.

This study was financed in part by the Conselho Nacional de Desenvolvimento Científico e Tecnológico - Brasil (CNPq).

## Abstract

de Armas, Elvismary Molina; Lifschitz, Sérgio (Advisor). **A novel approach for de Bruijn Graph construction in de novo genome fragment assembly**. Rio de Janeiro, 2019. 104p. Tese de doutorado – Departamento de Informática, Pontifícia Universidade Católica do Rio de Janeiro.

Fragment assembly is a current fundamental problem in bioinformatics. In the absence of a reference genome sequence that could guide the whole process, a *de Bruijn* Graph data structure has been considered to improve the computational processing. Notably, we need to count on a broad set of $k$-mers, biological sequences substrings. However, the construction of *de Bruijn* Graphs has a high computational cost, primarily due to main memory consumption. Some approaches use external memory processing to achieve feasibility. These solutions generate all $k$-mers with high redundancy, increasing the number of managed data and, consequently, the number of I/O operations. This thesis proposes a new approach for *de Bruijn* Graph construction that does not need to generate all $k$-mers. The solution enables to reduce computational requirements and execution feasibility, which is confirmed with the experimental results.

## Keywords

de Bruijn Graph; genome assembling; k-mer

## Resumo

de Armas, Elvismary Molina; Lifschitz, Sérgio. **Uma nova abordagem para a construção do grafo de Bruijn na montagem de novo de fragmentos de genoma**. Rio de Janeiro, 2019. 104p. Tese de Doutorado – Departamento de Informática, Pontifícia Universidade Católica do Rio de Janeiro.

A montagem de fragmentos de sequências biológicas é um problema fundamental na bioinformática. Na montagem de tipo De Novo, onde não existe um genoma de referência, é usada a estrutura de dados do grafo *de Bruijn* para auxiliar com o processamento computacional. Em particular, é necessário considerar um conjunto grande de $k$-mers, substrings das sequências biológicas. No entanto, a construção deste grafo tem grande custo computacional, especialmente muito consumo de memoria principal, tornando-se inviável no caso da montagem de grandes conjuntos de $k$-mers. Há soluções na literatura que utilizam o modelo de memória externa para conseguir executar o procedimento. Porém, todas envolvem alta redundância nos cálculos envolvendo os $k$-mers, aumentando consideravelmente o número de operações de E/S. Esta tese propõe uma nova abordagem para a construção do grafo *de Bruijn* que torna desnecessária a geração de todos os $k$-mer. A solução permite uma redução dos requisitos computacionais e a viabilidade da execução, o que é confirmado com os resultados experimentais.

## Palavras-chave

grafo de Bruijn ;     montagem de genoma;     k-mer

# Table of contents

# List of figures

# List of tables

# List of Abreviations

DBG – de Bruijn Graph
GAGE – Genome Assembly Gold Standard Evaluations
NGS – Next-Generation Sequencing
OLC – Overlap Layout Consensus
GB – Gigabytes, equivalent to 1024 Megabytes
hrs – Hours, equivalent to 60 minutes
bp – base pair. Refers to a nitrogenous base that building blocks of DNA and RNA: adenine, guanine, cytosine, thymine and uracil. BF – Bloom Filter data structure.
FP – False positives.

# 1
# Introduction

The computational fragment assembly [El-Metwally et al. 2013] is a fundamental problem for bioinformatics. As DNA sequencing technologies cannot read whole genomes in a single run, one needs to reconstruct the original sequences considering the large volume of short *reads*. Indeed, Next-Generation Sequencing (NGS) projects break the genome randomly at several places and generate several small fragments, the so-called *reads* of the genome. NGS [Metzker 2010] machines commonly deliver a massive number of small reads - varying from 35 to 400 base pairs depending on the specific technology used-, with low cost comparing with previous Sanger generation.

Since fragments of DNA are broken in randomly positions, and sequencer machines do not have a 100% of accuracy, it is needed to increase the sequencing coverage. The coverage is measured in function of the average number of reads covering a position in the genome.

Given the pieces taking from non-exact positions, and the great coverage, a high level of redundancy is generated in the fragments. The number of reads could be hundreds of millions; thus, the total volume of data may reach tens or even hundreds of GB. For example, sequence data from Assemblathon2 competition includes BGI Illumina HiSeq 2000 reads for a parrot with 269 GB of sequences, 219x coverage, with read lengths 90 and 151bp [Bradnam et al. 2013].

There is an additional challenge when dealing with *de-novo* [Schatz et al. 2010] assembling methods since there are no reference genomes to guide the assembly procedure.

## 1.1
## *de Bruijn* graph and *k*-mers

Some genome assemblers have been implemented based on a *de Bruijn* graph (DBG) data structure, which helps to compute assembly overlaps [Bradnam et al. 2013, Salzberg et al. 2012].

In order to build a *de Bruijn* graph, the set of short reads $R = \{r\}$ are firstly decomposed into $k$-mers (substrings with specific $k$ length). A short read $r$ is a string over the alphabet $\Sigma = \{A, T, C, G\}$, with $|\Sigma| = 4$. Each character of the alphabet represents one of the four nitrogenous bases present in DNA: adenine ($A$), guanine ($G$), cytosine ($C$) and thymine ($T$).

The short reading length $m$ depends on the sequencing technology used, which varies from 35 to 150 bp (base pairs) with low error rates of up to 300 bp, using Illumina technology, for example. However, it can reach 10 to 15 kbp, but paying a higher error rate, when using Pacific Biosciences technology, for example.

In a DBG, unique $k$-mers constitute nodes, and an edge is set between two nodes when the $k$-mers of those nodes occur consecutively in at least one read. The total number of $k$-mers present in one read (not only distinct $k$-mers) is equal to $m - k + 1$, while the total number of $k$-mers present in $n$ reads is $(m - k + 1) * n$. Taking into account that the number of $n$ can be in the order of the millions, the number of $k$-mers can easily reach the billions.

The creation and manipulation of the *de Bruijn* graph has been identified as the step with most memory and runtime consumption for some assembling experiments [Li et al. 2009][Cook and Zilles 2009][Li et al. 2013]. In fact, the fundamental computational drawback of the *de Bruijn* graph approach is that it requires an enormous amount of memory for its construction. For example, on the supplementary results for Assemblathon 2 [Bradnam et al. 2013], in which software-based on *de Bruijn* graphs are included, there is a list of the computational requirements of different assembly pipelines. It is not surprising to see that they round over 14 hours run time with 512 GB RAM machine with 48 cores, depending on species and the software used.

## 1.2 Approaches for DBG construction

Several techniques have been proposed to reduce the memory and time footprint during the assembly process [Kleftogiannis et al. 2013]. They use a variety of data structures and processing approaches, focused directly or not in the construction process. Most of them fit into three categories: those that focus on reducing the amount of data, others that try to increase computational resources, and yet those based on the external memory processing model.

The first category tries to reduce the amount of data as much as possible by removing, sampling or using probabilistic data structures like Bloom Filters. They are committed to the quality of the assembly. These solutions are not exact representations of the DBG. The second group focuses on the allocation of more resources over the cloud or increasing the degree of parallelism until sufficiency to generate the assembly. The last one group, which include [Li and XifengYan 2015], [Chikhi et al. 2014] and [Chikhi et al. 2016], is based on external memory models. They partition the data into smaller units such that they can be processed in main memory and eventually merge the results. The element of greater weight in these solutions is the number of I/O operations when compared with random access memory times in main memory processing. The last two approaches follow an exact representation of DBG.

One aspect that has a significant impact on memory consumption is the number of elements ($k$-mers) that are necessary to process and keep in the main memory. In addition, $k$-mers present a great level of redundancy since they share $k-1$ bases. For $n$ reads, the redundancy level becomes really high.

It is important to note that the $k$ value for the best assembly is not known. The $k$ value has a mainly role in *de novo* processing. Considering the way that the graph is constructed, the $k$ value defines the minimum substring that two reads must share to be linked in a graph path. Also, $k$ greatly impacts the capacity of distinguishing between genome repetition, repetition generated by a large coverage, or by a sequencing error.

Since a reliable $k$ value for optimum assembly is not known beforehand, the researchers usually generate several assemblies by varying $k$.

Depending on $k$ and the dataset, the number of $k$-mers processed and the number of unique $k$-mers could be higher or lower. Therefore, an amount of memory $M$, enough to execute an experiment with $k_1$, could be not enough with $k_2$ ($k_1 \neq k_2$) for the same dataset.

We can succinctly define our scientific and technological problem as:

Computational requirements for DBG construction depend on $k$-mer processing and a particular $k$. The number of $k$-mers, significantly affects the execution time, while the number of unique $k$-mers significantly impacts the amount of required RAM for a DBG exact representation.

## 1.3 Research scope

We propose in this thesis a new approach to generate the DBG based on a reduction of the number of processed $k$-mers.

To explain and justify our approach, we present:

– the principal concepts associated with the problem.

– evaluation of state of the art, deepening the techniques used to satisfy the need for high computational requirements.

– Identify the main parameters that may impact in computational requirements for DBG construction.

– Implement our approach and execute a set of relevant experiments.

- Evaluate our approach with real datasets.

In addition, it is also our interest to:

- Analyze the computational requirements of our approach.
- Evaluate the feasibility of our approach in the presence of limited main memory.
- Evaluate the performance of our approach to reduce execution time.

Our research focuses on the exact representation of DBG. However, this does not mean that it cannot bring improvements when used in conjunction with an approximation technique.

## 1.4 Structure of the work

This text is structured as follows: we list and describe the fundamental concepts of our research in Chapter 2, including the challenges of de novo assembly and the construction of *de Bruijn* graphs. Chapter 3 contains a description of the state of the art and main related works. Our novel approach is described in detail in Chapter 4 through examples and computational requirements analysis. Lastly, Chapter 5 explains our implementation and analyzes the results of some experiments executed to validate our approach. Finally, Chapter 6 summarizes our contributions and outgoing works.

# 2
# Principal concepts

Next-generation sequencing (NGS) technologies have brought rapid progress for the biological research area. Nevertheless, the genome assembly problem continues to be a challenge since we need to reconstruct a whole-genome by joining a vast amount of short reads.

Assembly algorithms and their implementations are typically complex. They could require high-performance computing platforms for large genomes. Algorithmic success can depend on pragmatic engineering and heuristics formulated by empirically derived rules of thumb.

For *de novo* assembly, without a known reference genome, the complexity is higher. Some successful approaches are based on the use of *de Bruijn* graph. However, the construction and use of *de Bruijn* graph demand a large amount of main memory and execution time because of the large number of elements (nodes and edges) to process.

In this chapter will be present in details the main concepts related to *de novo* assembly, the technologies involved, and theoretical concepts about the *de Bruijn* graph structure.

## 2.1
## Next-generation Sequence Data

The field of biological research has changed rapidly since the advent of massively parallel sequencing technologies, known as next-generation sequencing (NGS) [El-Metwally et al. 2013], [Claros et al. 2012].

Some commercial DNA sequencing platforms include the Genome Sequencer from Roche 454 Life Sciences (www.454.com), the Solexa Genome Analyzer from Illumina (www.illumina.com), the SOLiD System from Applied Biosystems (www.appliedbiosystems.com), the Heliscope from Helicos (www.helicos.com), and the commercialized Polonator (www.polonator.org). A distinguishing characteristic of these platforms is that they do not rely on Sanger chemistry [Sanger et al. 1980] as did first-generation machines. With their arrival in the market in 2005 and rapidly developing since then, they have dramatically lowered the cost per sequenced nucleotide and increased throughput by orders of magnitude [Niedringhaus et al. 2011]. Their performance increased dramatically the numbers of reads generated, many hundreds of thousands or millions of reads, and all this in a relatively short time [Kleftogiannis et al. 2013] and good genome coverage. For example, the highest capacity sequencing instruments available today, such as the Illumina HiSeq x Ten[1], has an output system that generates 1.8 TB of sequence per instrument run in 3 days, with an output rate per day of 600GB and a read length of 150 (`http://allseq.com/knowledge-bank/illumina/`). A review of NGS technologies appears in [Zhou et al. 2010], [Liu et al. 2012].

NGS has brought an essential impact in various biological areas such as genomics, transcriptomics, metagenomics, proteogenomics, gene expression analysis, noncoding RNA discovery, SNP detection, and the identification of protein binding sites [El-Metwally et al. 2013]. The genome assembly problem arises because it is impossible to sequence a whole genome directly in one read using current sequencing technologies.

The application of NGS had a significant impact mainly on the sequencing projects in which there was a proper reference sequence, due to the much shorter reading length (30 – 400 bp) compared to the Sanger sequence (500 – 1000 bp), because they are comparatively very cheap. However, for assemblies with no reference genome, called *de novo*, assembling a large genome ($>$ 100 Mbp) using short readings remains a challenge. While the cost of sequencing no longer becomes a limiting factor for most large new projects, and sequence fragment assembly becomes the biggest challenge.

NGS reads can vary from 400 bp to 100 bp or less (see 454 machines, Solexa and SOLID machines) [Schatz et al. 2010], [Miller et al. 2010]. Since shorter reads deliver less information per read, the computational problem of assembling chromosome-size sequences gets worse. Consequently, for sure, high coverage is necessary to capture most of the genome with the greatest certainty. However, high coverage increases complexity and intensifies computational issues related to large data sets.

## 2.2 Genome assembly

The process of reconstructing a whole-genome by joining these reads together up to the chromosomal level is known as genome assembly.

An assembly is a hierarchical data structure that maps the sequence data to a putative reconstruction of the target genome. It groups reads into contigs and contigs into scaffolds. Contigs provide one representation by multiple

[1]that has specifically been created to bring down the price of human whole-genome sequencing to under US$ 1,000

sequence alignment of reads. The scaffolds sometimes called supercontigs or metacontigs, define the contig order and orientation and the sizes of the gaps between contigs. Scaffold topology may be a simple path or a network [Miller et al. 2010]. In the following, we comment on some assembly challenges by using NGS data:

– **Read length:**

DNA sequencing technologies share the fundamental limitation that read lengths are much shorter than even the smallest genomes. The process of determining the complete DNA sequence of an organism's genome overcomes this limitation by over-sampling the target genome with short reads from random positions.

– **Repeats:**

Assembly software is challenged by repeated sequences in the target genome. Genomic regions that share perfect repetitions may be indistinguishable, principally if the repetitions are longer than the reads. The repetition resolution is more difficult in front of sequencing errors. However, it can be assisted by high coverage and matched final readings, using an expansion strategy. Careful estimates of the repetition resolution involve the ratio of the reading length (or paired-end separation) to the repetition length, repetition fidelity, reading accuracy and reading coverage.[Miller et al. 2010]

Concerning NGS data, shorter reads have less power to resolve genomic repeats, but higher coverage increases the chance of spanning short repeats.

– **Error sequencing:**

Another essential aspect that complicates the assembly process is the error profiles for each NGS platform, compared in [Zavodna et al. 2014]. While implementations must tolerate imperfect sequence alignments to avoid missing valid joins, this can brings false-positive joins. This is a problem, especially with reads from inexact polymorphic repeats and polymorphic differences. False-positive joins can induce chimeric assemblies.

Over-sampling the target genome with short reads from random positions, and high overages are techniques used to mitigate error profiles.

– **Coverage**:

Although the sequencing accuracy for each nucleotide is high, a large number of nucleotides in the genome means that if an individual genome is only sequenced once, there will be a significant number of sequencing errors. Also, many positions in a genome contain single nucleotide polymorphisms (SNPs). Therefore, to distinguish between sequence errors and true SNPs, it is necessary to further increase sequence accuracy by sequencing individual genomes in a large number of times.

The theoretical or expected coverage is the average number of times that each nucleotide is expected to be sequenced, given a certain number of reads of a given length and the assumption that reads are randomly distributed across an idealized genome [Sims et al. 2014].

In other words, coverage represents the average number of how many times a particular base position in the genome target is sequenced. Actual empirical per-base coverage represents the exact number of times that a base in the reference is covered by a high-quality, aligned read from a given sequencing experiment. Therefore, a 30X, for example, means that each particular base position was sequenced on average 30 times.

Very low coverage induces gaps in assemblies. As coverage increases, the fraction of the genome sequenced increases while the number of gaps decreases. However, each sequencing technology has its own biases that produce gaps in coverage [Schatz et al. 2010].

Coverage variation is introduced by chance, by variation in cellular copy number between source DNA molecules, and by the compositional bias of sequencing technologies [Zhou et al. 2010]. In that sense, the assembly is complicated by non-uniform coverage of the target.

Researchers typically determine the necessary NGS coverage level based on the method they're using, as well as other factors such as the reference genome size, gene expression levels, specific application of interest, published literature, and best practices from the scientific community [Illumina, Inc. 2019].

– **Computational complexity:**

Assembly operation could require high-performance computing platforms for large genomes and the processing of larger volumes of data. Algorithms are typically complex and depend on pragmatic engineering and heuristics. Heuristics help to overcome complicated repetition patterns in real genomes, random and systematic errors in real data, and the physical limitations of real computers. Over more, the implementations and results are tied to a good parameter instantiating. In case of *de novo* assembly, using a $k$-mer based algorithms (see Section 2.3.2), the selection of $k$ value is vital. These solutions are lower sensitivity, thus could missing some true overlaps depending on $k$. The probability that true overlap spans shared $k$-mers depends on the value of $k$, the length of the overlap, and the rate of error in the reads [Zhou et al. 2010]. An appropriate value of $k$ should be large enough that most false overlaps don't share $k$-mers by chance, and small enough that most true overlaps do share $k$-mers. The choice should be robust to variation in reading coverage and accuracy.

## 2.3 De novo assembly approaches

NGS assemblers can be organized into three categories, all based on graphs. The Overlap-Layout-Consensus approach (OLC) is based on an overlap graph. The *de Bruijn* Graph (DGB) approach relies on some form of $k$-mers as vertices of the graph, and the greedy graph algorithms may use OLC or DBG. Next, we will delve into the first two.

### 2.3.1 Overlap graph

Conceptually, the overlap graph models the overlaps between reads. Its nodes represent the reads, and the edges represent the overlaps [Myers 1995]. Paths through the graph are the potential contigs.

The Overlap-Layout-Consensus (OLC) assembly strategy is used over overlap graphs [Li et al. 2011]. As it is named, OLC works through three main steps: first overlaps (O) between all the reads, then it moves out a layout (L) of all the reads and overlaps information on a graph, and finally, the consensus (C) sequence is inferred [Li et al. 2011]. OLC became successful with the wide application of Sanger sequencing technology. Various widely used assembly programs adopted OLC, an example of them is Celera Assembler [Myers et al. 2000].

### 2.3.2 The *de Bruijn* graph

The *de Bruijn* graph was defined outside the realm of DNA sequencing to represent strings from a finite alphabet. The nodes represent all possible fixed-length strings. The edges represent suffix-to-prefix perfect overlaps. [Miller et al. 2010] In the context of genome assembly, the *de Bruijn* graph in the could be defined as a graph formed by $k$-mers, following the next definitions:

**Definition 2.1** *Read: A read $r$ is a string with length $m$ that represent a genome sequence, over the alphabet $\Sigma = \{A, T, C, G\}$, with $|\Sigma| = 4$. Each* *character of the alphabet represents one of the four nitrogenous bases present in DNA: adenine (A), guanine (G), cytosine (C) and thymine (T).*

**Definition 2.2** *k-mer*
*A k-mer is a substring over a read with specific k length.*

The $k$-mer is a string whose length is $k, 1 < k < m$. $k$ defines the minimum length of a substring that two reads must share to define an overlap, linking two reads in the graph traverse as a result. Using a larger $k$ value involves more accuracy to discover repeated regions in the genome, but also increases the chances of loose overlaps in reads, causing the loss of links in the graph. Consequently, it is not easy to estimate the right $k$ value for the best assembly. The total number of $k$-mers present in one read is equal to $m-k+1$, while the total number of $k$-mers present in $n$ reads is $(m-k+1)n$. The unique $k$-mers space for $k$ value is $4^k$.

**Definition 2.3** *de Bruijn graph*

*De Bruijn graph, $G_k(V, E)$ represents overlaps between k-mers, in which:*

- *The set of vertices is defined by $V = S = \{s_1, s_2, ..., s_p\}$, where $S$ is a set of unique k-mers over a given set of reads.*
- *The set of edges is defined by $E = \{e_1, e_2, ..., e_q\}$, where $e = (s_i, s_j)$ if and only if the $k-1$ suffix of $s_i$ matches exactly the $k-1$ prefix of $s_j$. $s_i$ and $s_j$ must be adjacent k-mers in at least one read.*

The life cycle of DBG for genome assembling can be resume as:

- Construction:
  - Generate all $k$-mers.
  - Generate a node per distinct $k$-mer.
  - Generate an edge between two nodes if these $k$-mers have a $k-1$ overlapped in at least one read.
- Processing:
  - Simplification: Compaction and applying heuristics to resolve errors.
  - Traverse the graph to generate contiguous regions of genome called *contigs*. Those are the assembly result.

Vertices, may contain additional information to the $k$-mer itself, such as the read identification, position, and multiplicity.

**Definition 2.4** *Multiplicity of a node of DBG*
*The multiplicity of a node is the number of times that the $k$-mer of this node appears in the dataset.*

A lower multiplicity could indicate sequencing errors. The multiplicity could be used with the coverage to identify erroneous $k$-mers and suppress them.

In order to build a *de Bruijn* graph, traditional approaches firstly decomposes the set of reads $R$ into $k$-mers. The read length $m$ depends on the sequencing technology used, being that for smaller values of length, a lower rate of errors. Then it is needed to identify all distinct $k$-mers (set of vertices $V$) from the collection of all $k$-mers, and map all duplicate $k$-mers into the unique corresponding node in the *de Bruijn* graph, updating the edges.

An edge between two nodes is created whenever their corresponding $k$-mers are adjacent in at least one short read. In other words, an edge is created between two $k$-mers if they have an exact suffix-prefix overlap of length *k-1* that occurs in at least one read. Therefore, a way to get the set of edges $E$, once the set of vertices is built, is by creating them through scanning all the reads in the short sequences file and querying nodes in $V$.

After the construction of DGB, in the second stage, it must be traversed thought the paths of the graph in function to obtain contigs regions of the genome target.

By construction, the graph contains a path corresponding to the original sequence (Fig. 2.1). Each read induces a path. Reads with perfect overlaps induce a common path. Thus, perfect overlaps are detected implicitly without any pair-wise sequence alignment calculation (Fig. 2.2).

However, some problems are present for DBG representation:

– Repeats induce cycles, which would allow more than one possible reconstruction of the target.

– DNA is double-stranded. Therefore, the forward sequence of any given read may overlap the forward or reverse complement sequence of other reads.

– The graph contains insufficient information to disambiguate the repeats in real genomes that have complex repeat structures, including tandem repeats, inverted repeats, and imperfect repetitions.

– Palindromes sequences induce paths that fold back on themselves. A palindrome is a DNA sequence that is its own reverse complement.

– Real data includes sequencing errors that can generate false paths.

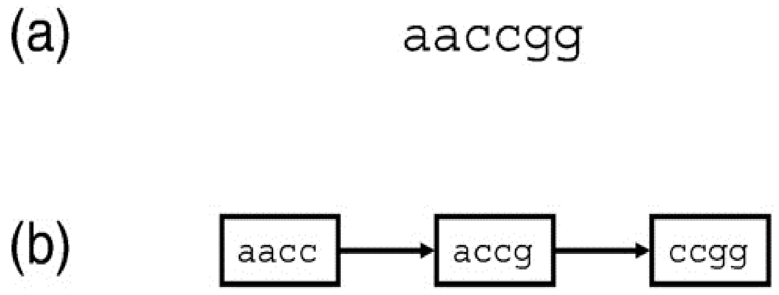


Figure 2.1: A read represented by *de Bruijn* graph. (a) Original read. (b) DBG for $k = 4$. The graph has a node for every unique $k$-mer in the read plus a directed edge for every pair of $k$-mers that overlap by $k - 1$ bases in the read. In these examples, the paths are simple because the value $k = 4$ is larger than the 2bp repeats in the read. The read sequence is easily reconstructed from the path in either graph. [Miller et al. 2010]

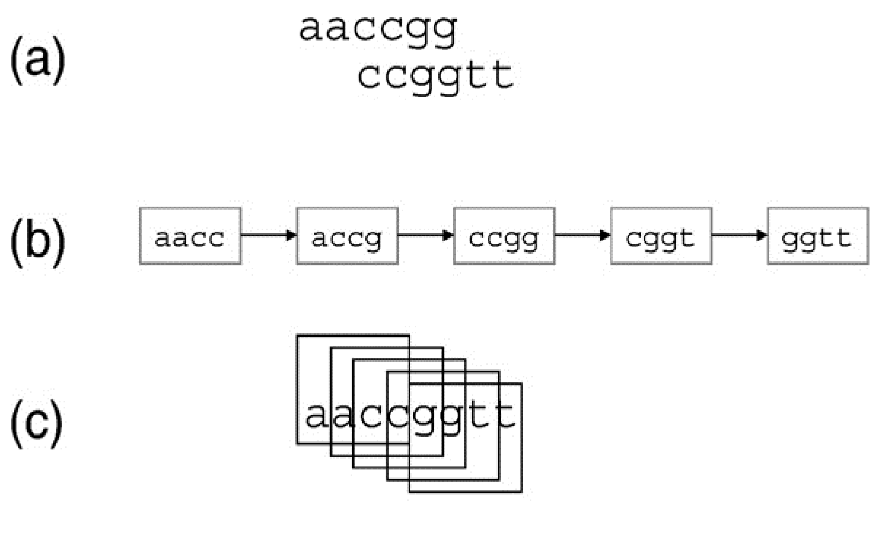


Figure 2.2: A pair-wise overlap represented by a DBG. (a) Two reads have an error-free overlap of 4 bases. (b) DBG represents both reads, with $k = 4$. The pair-wise alignment is a by-product of the graph construction. (c) The simple path through the graph implies a contig.[Miller et al. 2010]

Some problems could be identified through the topological characteristics of the graph. For example:

- Tips are short, dead-end divergences from the main path (Fig. 2.3 (a)). They are induced by sequencing error toward one end of a read. They can be induced by coverage dropping to zero.
- Bubbles are paths that diverge then converge (Fig. 2.3 (b)). They are induced by sequencing error toward the middle of a read, and by a polymorphism in the target.
- Paths that converge then diverge from the frayed rope pattern (Fig. 2.3 (c) ). They are induced by repeats in the target genome.
- Cycles are paths that converge on themselves. They are induced by repeats in the target. For instance, short tandem repeats induce small cycles.

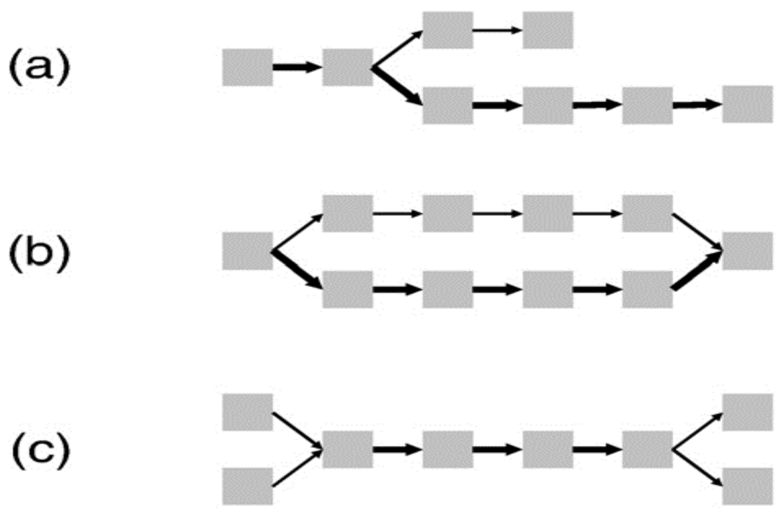


Figure 2.3: Topological characteristics that could be presented in the DBG: (a) An errant base call toward the end of a read causes a "tip" or short dead-end branch. (b) An errant base call near a read middle causes a "bubble" or alternate path. Polymorphisms between donor chromosomes would be expected to induce a bubble with parity of read multiplicity on the divergent paths. (c) Repeat sequences lead to the "frayed rope" pattern of convergent and divergent paths.[Miller et al. 2010]

Therefore, after the construction, DBG needs to be simplified and corrected to reduce the complexity and generate the *contigs*. Since the detection and elimination of errors and divergence paths are not trivial algorithms, some heuristics and approximation algorithms are used in that step.

Compared to overlap graphs, *de Bruijn* graphs are more sensitive to repeats and sequencing errors. Paths in overlap graphs converge at repeats longer than a read, but paths in *de Bruijn* graphs converge at perfect repeats of length $k$ or more, and $k$ must be less than the read length. Each single-base sequencing error induces up to $k$ false nodes in the *de Bruijn* graph. Each false node has a chance of matching some other node and thereby inducing a false convergence of paths.

The DBG does not require all-against-all overlap discovery, it does not (necessarily) store individual reads or their overlaps, and it compresses redundant sequence. Conversely, the DBG graph does contain actual sequence and the graph can exhaust available memory on large genomes.

## 2.4 Partial conclusions

Next-generation sequence data has significantly impacted several fields of bioinformatics, greatly reducing costs. However, the genome assembly continues to be a challenge for genomic researches since there is no technology capable of sequencing the whole-genome, even for the smallest genomes.

Some aspects of NGS data difficult the assembly, such as the error profiles for each NGS platform, the no-uniform coverage of the target, which difficulties the resolution of genome repetitions. It fundamental limitation that read lengths are much shorter than even the smallest genomes.

Two main approaches had lead the *de novo* assembly, OLC and *de Bruijn* graph. The second has been more addressed in the NGS era. The construction of *de Bruijn* graph is the main memory consumption step. The most important parameter in the *de Bruijn* graph is the $k$ value that impacts the accuracy of the assembly, the number of vertices and edges, and the memory requirements.

# 3
# Related works

There are some assemblers that use DBG approach, for example Velvet [Zerbino and Birney 2008] [Zerbino 2016], ALLPATHS [Butler et al. 2008], ABySS [Simpson et al. 2009], SOAPdenovo [Luo et al. 2012], and Contrail (`https://sourceforge.net/projects/contrail-bio/`). However, the *de Bruijn* approach has as a drawback that *de Bruijn* graph can require an enormous amount of memory (GB of RAM) and the construction and analysis of a *de Bruijn* graph is not easily parallelized [Schatz et al. 2010]. As a result, *de Bruijn* assemblers such as Velvet and ALLPATHS, which have been used successfully on bacterial genomes, do not scale to large genomes. For a human-sized genome, these programs would require several terabytes of RAM to store their *de Bruijn* graphs [Schatz et al. 2010], and memory requirements may be higher for more complex genome organisms, as is the case of many plants. In order to know the magnitude of CPU/RAM requirements and runtime for non-mammalian vertebrates (bird, fish, and snake species genomes), we suggest reviewing the supplementary results for Assemblathon2 paper [Bradnam et al. 2013].

This high memory consumption problem is expected to worsen in the future because the NGS data generation rate has exceeded expectations based on Moore's law [Jackman and Birol 2010], meaning that the amount of raw data is expected to grow much faster than the capacity of available memory [Kleftogiannis et al. 2013]. Based on this affirmation, we assume that: *no matter how much main memory we have available; there may be an assembling case for which this amount is not enough.*

In this chapter, we present an overview of approaches that deal with high memory consumption. We classify them into some categories and delving into external memory approaches.

## 3.1
## Main classification of approaches

Several techniques have been proposed to reduce the memory footprint for the assembly process. Those approaches can be divided into two general groups. First, include solutions that reduce the amount of data. The second group contains those solutions that increase the memory resources for the same amount of data, through partitioning and/or distributing.

The first approaches try to reduce (by removing or sampling) as much as possible the data until it is possible to execute the assembly with available resources. However, it may affect the final quality of the assembly. Conversely, the compression without loss of information also brings additional cost because of the periodic process of zipping and unzipping.

In that group, also we include those approaches that use probabilistic data structures. In that way, they can use a fixed amount of memory, independently on the number of items to be processed. Compounding the problem, there is no accurate measure to guide the reduction of the data. The memory consumption during the assembling process is highly sensitive to data and the value of $k$. Thus the success of the reduction is only checked when the assembling is achieved with the available memory.

These solutions that reduce the amount of data by removing, sampling, or using probabilistic data structures like Bloom Filters, are considered in our work **as not exact representations of DBG**.

The second group is based on greedy methods, allocating more resources until they will be sufficient to generate the assembly. These approaches assume that there are infinite resources, which is not real, for finite input data.

## 3.2 *k*-mers counters

The DBG construction implies a subrutine to identify distinct $k$-mers and get their multiplicity. Identify distinct $k$-mers problem also has been touched by counting $k$-mer tools [Marcais and Kingsford 2011], [Melsted and Pritchard 2011], [Rizk et al. 2013], [Deorowicz et al. 2013], [Li and XifengYan 2015], [Deorowicz et al. 2015]. Although the $k$-mer counter tools have as main objective to generate histograms about $k$-mers distributions, its processes has some similarities to the process to get the vertices set of the DBG. Identifying distinct $k$-mers have been approached by sorting [Deorowicz et al. 2013], [Deorowicz et al. 2015], hashing [Marcais and Kingsford 2011], [Rizk et al. 2013], [Li and XifengYan 2015] or using Bloom Filters [Melsted and Pritchard 2011], combined some times with parallel approaches to speed up the process [Deorowicz et al. 2013], [Deorowicz et al. 2015]. Some of them like [Rizk et al. 2013], [Deorowicz et al. 2013], [Deorowicz et al. 2015], and [Li and XifengYan 2015] have been focused on distribution the $k$-mers in disk partitions to counter them before, loading in main memory each partition at time. Since the objective is only to get the frequencies of the $k$-mers, $k$-mer counters have not the notion of vertices and edges. In addition, there are a lot of them that processes the $k$-mers taking into some assumptions with the aim of reducing the amount of data, for example, they do not count the $k$-mers with frequencies smaller than a given value, or unify in a single category the $k$-mers whose frequencies are larger than a given value, these criteria may not be appropriate for the DBG construction.

## 3.3 Techniques to reduce memory footprint for DBG construction

More specifically, the techniques to reduce the memory requirements for the assembly process found in the literature can be examined through the following categories:

***Pre-processing techniques:***

First, there are preprocessing techniques such as Diginorm in [Titus Brown et al. 2012], Quake [Kelley et al. 2010], ALLPATHS-LG error corrector [Gnerre et al. 2011], that try to reduce the input size removing redundant information and errors before the assembly process itself start.

***Optimized data structures for graph representation:***

To minimized the memory requirements during the $k$-mer unique identification process, were used effective indexes for identifying duplicate $k$-mers, trying to reduce the space required and the number of operations. Hash tables in memory have been used successfully for many assemblers, such as in [Zerbino 2016] [Simpson et al. 2009] [Luo et al. 2012], to identify duplicate $k$-mers. However, for a large amount of NGS data, they do not work well because the entire hash table does not fit in memory. Suffix-array is a data structure used to compute overlaps. The FM-index [Simpson and Durbin 2010] has been used to allow the compressed representation of input reads and fast computation of overlaps in string graph (equivalent to overlap graph), but it is not tested yet in the construction of *de Bruijn* graph.

The succinct bitmap data structure in [Conway and Bromage 2011] was also used to compress the representation of *de Bruijn* graph, but overall need for space will continue to increase as the graph becomes “bigger”. Other approaches are based on the idea of sparseness in genome assembly [Ye et al. 2012], where only a subset of $k$-mers present in the dataset is stored.

Bloom Filters have been arduously explored as solution to deal with DBG computationally demanding [Melsted and Pritchard 2011], [Zhang et al. 2014], [Chikhi and Rizk 2013]. Bloom Filters are used to store vertices ($k$-mers), while the edges are implicitly deduced by querying the Bloom filter for the membership of all possible extensions of a $k$-mer. However, this approach does not correspond exactly to the edges contained in the reads. Some works have been focused on propose mechanisms that avoid false positives using Bloom Filters, for example, the works in [Chikhi and Rizk 2013], and [Salikhov et al. 2014].

Partition assembly algorithms were also proposed. For example, the Minimum Substring Partitioning (MSP) [Li et al. 2013] technique allows us to split the input reads into subsequences and distributes then into this disk partitions, then processing one disk partition at a time. Moreover, the $k$-mers partitions can be processed in a distributed manner, as well as the Contrail proposed to avoid memory bottleneck.

***Extending the computational resources:***

Finally, in that category, we group solutions that proposed the use of cloud resources to overcome the memory requirements limitations. In [Kleftogiannis et al. 2013] were designed a set of assembler experiments using the GAGE data sets and a group of program assemblers. All experiments were performed in workstations on Amazon EC2 with the aim of comparing the cost of use each platform. Financial analysis reveals that the assembly of bacterial genomes, which takes a few minutes, can be processed on the cloud at a minimal cost. Assembly of medium-sized genomes costs around US$1, which is a significant improvement if we consider the cost of a machine equipped with 32GB RAM. The cost for assembly of more complex genomes is higher because such an operation requires more expensive virtual machines, and the assembly takes several hours. The average cost ranged between US$10 and US$1, decreasing with the use of programs that improve the trade-off between memory consumption and execution time.

## 3.4 External memory approaches

Among the DBG construction strategies, we have found only a few approaches that consider external memory processing. They fit into three strategies: one based on external memory sorting, second based on $k$-mers partitioning and disk distribution, and the last one based on the construction of the graph embedded into a relational database management system (RDBMS).

External memory sorting is implemented in the solution proposed by [Kundeti et al. 2010]. In this work, the authors present an efficient parallel strategy for constructing large *de Bruijn* graphs, also extensible to the out-of-core model. After the generation of all canonical edges, they sort them and remove duplicates. Then, they detect duplicate $k$-mers by sorting all the edges using radix sort algorithms in an external implementation. The authors prove that replacing the radix sorting with an external R-way merge sort they get an optimal number of I/O's equal to $\Theta(\frac{N \log(N/B)}{B \log(M/B)})$. Comparing this approach with Velvet's graph construction shows the superiority on time or RAM of the new algorithm using 8 million unpaired reads obtained from sequencing a plant genome at CSHL and $k = 21$. However, there are some questions unanswered. There is not clear how this approach manages the possibility of insufficient memory for the construction of the adjacency list of the graph. Moreover, the program is not available for testing and use in actual assembling projects.

The distributed processing over disk partitions presents, in general, three steps: distribution, process, and merging. Firstly, they distribute all $k$-mers into disk partitions (not disjoint partition for all cases). Then it processes each partition individually in the main memory. Later, it merges them to build a DBG. The Minimum Substring Partition (MSP) approach [Li et al. 2013] shows a solution that breaks the short reads into multiple small disjoint partitions based on the minimum $p$-substring of the $k$-mers. It allows consecutive $k$-mers to be distributed in the same partition, decreasing the number of I/O operations. The number of distributed elements is the number of all $k$-mers generated from the sequence $((m - k + 1)n)$.

The approaches presented in [Chikhi et al. 2014] and [Chikhi et al. 2016] are also focused on distributing the process over disk partitions but with the direct construction of a compressed graph. Given a set of short reads and a $k$, the compacted graph $G_c(S, k)$ is a graph obtained from $G(S, k)$ by compression of all its maximal non-branching paths. The work in [Chikhi et al. 2014] proposes a new pipeline using first a DSK $k$-mer counter as an input of a new algorithm to enumerate all the maximal simple paths (called BCALM) and represent them using a new data structured called DBGFM using an FM index.

First, DSK partitions the set of all $k$-mers and saves them into the disk, using one file for each partition. Then each partition is separately loaded in memory, and the count of each $k$-mer is processed. Finally, the result is obtained by merging independent solutions. Despite its excellent results, DSK assumes that its hash function can uniformly partition the set of distinct $k$-mer. This assumption could incur in not balanced partition files size, which affects the I/O throughput. This approach takes the output of DSK and distributes the unique $k$-mer set by minimizers frequency while trying to compact consecutive $k$-mers that constitute a simple path in the graph. In that way, DSK + BCALM perform two cycles of disk distribution: DSK distributes all $k$-mers in partition files and processes them, while BCALM distributes all distinct $k$-mers.

A sort of parallel version for the BCALM algorithm, called BCALM2, is presented in [Chikhi et al. 2016]. It differs from BCALM in the fact that datasets are not first processed using DSK. All $k$-mers are distributed to the disk partitions using the same concept based on the frequency of minimizers such as BCALM. However, it changes the criteria for $k$-mer distribution, allowing that the same $k$-mer could be distributed in more than one file partition.

Like the other approaches, BCALM2 generates and distributes all the $k$-mers, but in that case, a $k$-mer could be distributed in more than one partition, which increases the number of I/O operations.

Besides, a new approach to generate compacted DBG is proposed in [Minkin et al. 2016]. TwoPaCo is a two-pass paralleled algorithm based on the reduction of finding maximal non-branching paths problem into finding the set of junction positions (the position of the $k$-mers that flanking maximal non-branching paths). For that, they use boolean flags, which mark every position of the genome (all $k$-mers are marked as flanking $k$-mer). To store all edges of ordinary DBG, they use a BF data structure in the first pass. Then, for each position in the boolean flag, the corresponding edges are finding in the BF, and it is counted the in- and out-degrees. If this $k$-mer does not accomplish the flanking $k$-mer condition, then its flag is set to zero. Due to false positives that BF can generate, there is a second pass of the algorithm, using a hash table to filtered to filter positive flags. If the hash table does not fit in main memory, they propose to split the input $k$-mers into $\mathcal{L}$-parts and round the algorithm in $\mathcal{L}$-rounds. As a conclusion of the paper, the authors noted that the effectiveness of the algorithm relies on having whole-genome sequences. Because of this, it is inapplicable to the case when genomes are represented as short read fragments. Thus this approach can not be applied to *de novo* assembling genome.

The last external memory processing approach relies on the construction of the graph as functions of a relational database management system (RDBMS). [de Armas et al. 2016] described $k$-mer mapping process as part of the DBG construction. A group of published works test the viability of an RDBMS for management the main and external memory interchange in the process, as specify tools designed for optimizes deals with the I/O. A case study for $k$-mer mapping was implemented based on the algorithm of Velvet using PostgreSQL. In [de Armas et al. 2016] is proposed a model to make a performance comparison for different index structures in an ad-hoc cost model. Also, was proposed an study of indexes like B+-tree, hash over $k$-mer in [de Armas et al. 2017] and over $k$-mer $p$-minimum substring. This approach presents a significant advantage that allows incremental processing without reprocessing and recovery from failures [Silva et al. 2017] due to the operation is delegated to the management of I/O operations to the DBMS. However, the improvements in runtime given by index evaluation, the execution time resulting of the experiments could still be considered high.

Summarizing, the external memory DBG construction approaches available in the literature rely on the use of external memory by sorting, disk partitioning, or by using an RDBMS. Both works initially considering the total number of $k$-mers, to following obtain the vertices of the graph (unique $k$-mers) and corresponding edges. Working with the total number of $k$-mers implies maintaining a high level of redundancy. Consequently, high available memory resources (main or external) and a more significant number of I/O operations.

## 3.5 State of the art of plant genome assembly

The large complex plant genomes remain a particularly tricky challenge for *de novo* assembly due to a variety of biological, computational, and biomolecular reasons. Since plant genomes can be nearly 100 times larger than some of the currently sequenced animals, like birds, fish, or mammalian genomes. Also, they can have much higher ploidy, and higher rates of heterozygosity and repeats than their counterparts in other kingdoms [Schatz et al. 2012]. Due to this complexity, the computational requirements to generate a good assembly increase. Notably, the memory requirements for the execution of a *de novo* plant genomes assembling goes beyond the available RAM in desktop computers, that commonly cover the range from 8GB to 32GB. Usually, *de novo* assembly for plant fragment genome requires the execution on server computers with hundreds of GB of main memory, even cases have been reported where the use of data centers that reach 1TB is necessary.

Added to that, neither of the major efforts to expose the state-of-the-art of assembling technologies - Assemblathon 1 [Earl et al. 2011], Assemblathon 2 [Bradnam et al. 2013] and GAGE [Salzberg et al. 2012] -, used genome plants as experimental data for their competitions. Furthermore, these studies explored a range of genome from bacterial, non-mammalian vertebrates, and human chromosome 14, as well as a simulated genome derived from human chromosome 13. The data used represents a wide range of genome sizes from 3Mbp for *S. aereus* from [Salzberg et al. 2012] to 1.6Gbp for *Boa constrictor* in [Bradnam et al. 2013]). However, those estimated genome sizes are considered insignificant when compared to the estimated genome size, for example, of sugar cane.

A case of plant genome assembly was carried out by the Laboratory of Molecular Biology of Plants (IBqM), Institute of Medical Biochemistry at Federal University of Rio de Janeiro. IBqM has as one of its goals the study of the sugarcane genome into Brazilian species. The sugarcane is an essential sector in Brazil's economy. Beyond its role in the industrial production of sugar, the sugar-energy sector has a critical role in the use of renewable energy sources in Brazil, due to that only the product of sugarcane account for 15.7% of the energy supply of the country (http://www.unica.com.br/sustentabilidade/).

For those reasons, the IBqM researchers designed a group of assays combining the growing the SP70 sugarcane variety with drought, Nitrogen, and pathogen treatments. The genotype gathered of the plants was sequenced using Illumina Sequencer Technology, generating 24 libraries with five different read lengths: 32bp, 72bp, 76bp, 100bp, and 152bp, being the most representative the reads with 100bp. IBqM, in collaboration with BioBD Laboratory of the Informatics Department of PUC-Rio University, assumes the challenge of assembling approximately 300 million sequences of sugarcane. Using some pre-processing and sampling techniques, was produced a *de novo* transcriptome assembly (TR7) from sugarcane RNA-seq libraries submitted to drought and infection with *Aaa* [Santa Brigida et al. 2016]. This result allows taking account of other studies as non-coding RNA regulation [Thiebaut et al. 2017].

### 3.5.1 Summary

Given the state of the art, we listed here the main focused points that face the large amount of memory requirements of DBG construction. Those can classify as follow:

– Set structures for fast lookup with lower overhead
    (a) Hash tables
    (b) Search trees

– $k$-mer codification for lees memory per element
    (a) Lossless compression (find exact $k$-mer statistics). For example:
        (i) Jellyfish: with $f(m)$ bijective function, hash position encodes part of the $f(m)$
        (ii) DSK: classical 2bit $k$-mer codification
        (iii) KMC1: remove $p1$ and $p2$ length prefixes.
        (iv) Meraculous: lightweight hash (combination of hash family)

   (b) Lossy data compression
      (i) Bloom Filters (BF) with false positives.
      (ii) Specialized probabilistic data structures based on BF.

- External memory processing
- Increasing available resources
   (a) Parallelization
   (b) Distribution in the cloud
   (c) GPU and other memory systems

In Table 3.5.1, we summarize some approaches present in the literature, noting the data structure used to represent the DBG and the $k$-mer codification used, as possible.

| Citation | Id. of the approach. | Classification | Description of the approach | Data structure to store $k$-mers | $k$-mer codification |
|---|---|---|---|---|---|
| [Marcais and Kingsford 2011] | Jellyfish | $k$-mer counter | Main memory. Hash table merging in disk if not memory available. | Hash table. Mapping $\mathcal{U}k$ into [0, M-1], M = 2ˆl, $pos(m,i) = (hash(m) + reprobe(i)) modM$, $hash(m) = f(m) modM$. Use a quadratic reprobing function $reprobe(i) = i(i+1)/2$ | Hash key codification. $K$-mer $m$ is codifiying in $f(m)$ such as $f : Uk \rightarrow Uk$ bijection, for which it is easily compute both $f$ and its inverse. Key compression scheme that allows it to use a constant $\mathcal{O}(1)$ amount of memory per key in the hash table for most applications, regardless of the length $k$. Store $2k - l$ higher bits of $f(m)$ concatenated with bits representing the reprobe count. Given a position $x$, $k$-mer store is formed by the concatenation of [2k-l higher bits of f(m)][x - reprobe(i) mod M in l bits of f(m)] |
| [Melsted and Pritchard 2011] | BFCounter | $k$-mer counter | Bloom filter in main memory.Two pass algorithm to correct false positive. | Bloom Filter. Bloom filter with 4 times as many bits as the expected number of $k$-mers this corresponds to a memory usage of 4-bits per $k$-mer and use a and the optimal number of hash functions functions for the Bloom filter is d = 3. For the implementation it is used the Google sparsehash library and a Bloom filter library by A. Partow in (`http://www.partow.net/programming/hash-functions/index.html`). | Store a 1-byte counter for each $k$-mer and by default $k$-mers take 8-bytes of memory with a maximum $k$ of 31, although if desired, larger $k$-mers can be specified at compile time. |
| [Chapman et al. 2011] | Meracolous | assembler | Reliance on the linear U-U component of the graph as a starting point for making contigs. | Based on the construction of the U-U subgraph (compacted graph), which requires the entire $k$-mer hash to be held in memory. A novel lightweight hash for the *de Bruijn* graph. | Store $k$-mers with unique high quality extensions at both ends (i.e., those designated U-U in the previous step) in a hash where the "key" is the $k$-mer and the "value" is a two-letter code [acgt][acgt] that indicates the unique bases that immediately precede and follow the $k$-mer in the read dataset. The $k$-mer is not saved. A lightweight hash scheme uses a recursive collision strategy with multiple hash functions to avoid explicitly storing the keys themselves. |
| [Pell et al. 2012] | khmer | space-efficient solutions for dBG | Analysis of Bloom Filter, FP rate, percolation graph analysis to study the similarity in the graphs over different FP rates, graph connected-components analysis. | Bloom Filter. | |
| [Rizk et al. 2013] | DSK (Minia assembler) | $k$-mer counter | Disk distribution based on hashing. | Hash table. Function maps a $k$-mer to a numeric value in [0; H], where H is a large integer (typically 2ˆ64). The set of distinct $k$-mers values can be *uniformly* partitioned by this hash function. | Kmer is encoded using classical 2bits representation in the smallest available integer type. |
| [Deorowicz et al. 2013] | KMC1 | $k$-mer counter | Disk distribution. Partition is based on prefixes $p1$ and $p2$ of $k$-mers. Two stages processing. Sorting using the least-significant-digit (LSD) radix sort. | | $k$-mer entering in a process that are retired the $p1$ and $p2$ prefixes |
| [Chikhi and Rizk 2013] | Minia | space-efficient solutions for DBG | Bloom filter for nodes in main memory. In conjunction, cPF structure is used for a set of critical false positives, which is created and accessed on disk. | Bloom Filter. It is used 1 bit to represents more than one $k$-mer, along with the set of critical false positives. cFP structure, with a fixed amount of memory, is used to removing critical false positives. The set $S$ and $P$ are created and stored on disk. | Memory usage is approximated by $1.44log_2(16k/2.08) + 2.08$ bits/$k$-mer. It does not depend on $k$ value. cFP structure, has a fixed amount of memory. |

| | | | | | |
|---|---|---|---|---|---|
| [Zhang et al. 2014] | khmer | $k$-mer counter. | Count-Min Sketch in main memory. Not error correction is used. The measured counting error is analyzed. | Count-Min Sketch is used to storing the frequency distributions of distinct elements. The Count-Min Sketch permits online updating and retrieval of $k$-mer counts in memory which is necessary to support online $k$-mer analysis algorithms. In exchange, the use of a Count-Min Sketch introduces a systematic overcount for $k$-mers. The implementation extends an earlier implementation of a Bloom filter. | The data structure Count-Min Sketch stores only counts; $k$-mers must be retrieved from the original data set. |
| [Salikhov et al. 2014] | Cascading Bloom filter | assembler | Bloom filter with an additional representation of the set of FP using a cascading BF.Extend the idia of Minia, using a BF structure recursively to store the critical false positives . | Cascading Bloom Filters to store false positives $k$-mers | Bloom Filter |
| [Chikhi et al. 2014] | BCALM | Compacting *de Bruijn* graph + disk distribution | Pipeline based on the use of DSK first, and second the use of BCALM with the data structure based on FM index. BCALM is an algorithm to enumerate all the maximal simple paths without loading the whole graph in memory, using a distribution on disk. The input of BCALM is all $k$-mers from DSK, and the output is the maximal simple paths. | DBGFM is a data structure to store no-branch path, codified as FM index. | $k$-mers are used as in the DSK. After, it is used the concept of minimizers and $m$-compactation. |
| [Deorowicz et al. 2015] | KMC2 | $k$-mer counter | Disk distribution based on $k$-mer signatures (a carefully selected subset of all minimizers) to try to reach a more uniform distribution of $k$-mers in the bins. The amount of main memory needed is directly related to the number of $k$-mers in the largest bin.Two stages processing. Sorting using the least-significant-digit (LSD) radix sort. | | Store super $k$-mers (k + x' )-mers instead $k$-mer itself |
| [Mamun et al. 2016] | KCMBT | $k$-mer counter | Trie-based in-memory algorithm using Multiple Burst Trees. Three phases algorithm: first phase for insertion of compacted $k$-mers, the second phase to count, unzip them and insert them into $k$-mer specific trees, and the last phase employs a final traversal of these $k$-mer trees to identify all the unique $k$-mers with their counts. | Multiple Burst Trees (KCMBT), which uses cache efficient burst tries to solve this problem. Their ideas include the use of burst tries to store compacted $k$-mers, and unifying a $k$-mer and its count in a single unit. | |
| [Chikhi et al. 2016] | BCALM2 | Compacting *de Bruijn* graph + disk distribution | Based on BCALM plus parallelization. | | |
| [Minkin et al. 2016] | TwoPaCo | Compacting *de Bruijn* graph + disk distribution | Two-pass algorithm:- Filter the $k$-mers with a Bloom Filter - Process each unique $k$-mers with a hash table based on disk partitions. Must also maintain a $C$ set of junction position that represents each position of the genome using a boolean flag. Relies on having whole genome sequences, making it inapplicable to the case when genomes are represented as short read fragments. | BF with the set E of the edges of the ordinary DBG to filter the first time. After, redoing the process using a $T$ hash table with $E$ to remove FP edges. | Based on edges, insted of vertexes ($k$-mer). Use the BF to store the edges. |

| | | | | | |
|---|---|---|---|---|---|
| [Erbert et al. 2017] | Gerbil | $k$-mer counter | Disk distribution + GPU + minimizers for the efficient counting of $k$-mers for $k \geq 32$. Two-step approach. In the first step, genome reads are loaded from disk and redistributed to temporary files. In a second step, the $k$-mers of each temporary file are counted via a hash table approach. Use GPUs to accelerate the counting step. | Two hash functions, one for distributed and other for probing. In general, the combination of two independent hash functions leads to a more uniform distribution of $k$-mers. | |
| [Kokot et al. 2017] | KMC3 | $k$-mer counter | Disk distribution with some improvements over KMC and KMC2: better input/outpu (I/O) subsystem; signatures are as-signed to bins in an improved way for better balance of bin sizes; and second stage used a fast most-significant-digit radix sort. | | |
| [Rahman et al. 2017] | HaVec | space-efficient solutions for DBG. | Continuing the idea of Minia it is proposed the use of Bloom Filter, plus saving the quotient of the hash function division to verify false positives. | An auxiliary vector data structure is used to store the $k$-mers along with their neighbor information. It constructs such a graph representation that generates no false positives. For each index in the table, it is necessary to keep track of a mapping between hash values and quotients. | In the hash table, for each index, HaVec uses 40 bits, that is, 5 bytes. In order to represent the hash value of a $k$-mer, HaVec requires $2k$ bits. |
| [McVicar et al. 2017] | FPGA-solution | $k$-mer counter | Hardware approach using BF in junction with a FPGA boards for high capacity storage. $K$-mer counting is used as a problem to test FPGA-attached Hybrid Memory Cube (HMC), new memory subsystem. The HMC's high random-access rate is ideal for large Bloom filters, an efficient structure for checking membership in a set, or even counting occurrences. | Bloom Filter | |
| [Pandey et al. 2017] | Squeakr | $k$-mer counter | Representation is based upon a recently-introduced counting filter data structure CQFs. | The CQF stores an approximation of a multiset $S$ by storing a compact, lossless representation of the multiset $h(S)$, where $h : \mathcal{U} \to 0; ...; 2^p - 1$ is a hash function. To handle a multiset of up to $n$ distinct items while maintaining a false positive rate of almost $e$, the CQF sets $p = 1/4log2$ | |
| [Ghosh and Kalyanaraman 2016] [Ghosh and Kalyanaraman 2019] | FastEtch algorithm | space-efficient solutions for DBG. | Count-Min sketch, which is a probabilistic data structure for streaming data sets. The result is an approximate de Bruijn graph that stores information pertaining only to a selected subset of nodes that are most likely to contribute to the contig generation step. Besides, edges are not stored; instead, that fraction which contributes to our contig generation is detected on-the-fly. | CM sketch data-structure is a 2-D matrix of depth $d$ and width $w$, that stores $d \times w$ counts. The sketch uses $d$ hash functions: $h1...hd$: {1 . . . n} → {1 . . . w} that are from a pairwise independent family. | |

## 3.6 Partial Conclusions

Several techniques have been proposed to reduce the memory footprint for the assembly process, especially the memory requirements to construct DBG.

In this chapter, we classified those approaches into two main categories. The first group includes the approaches that try to reduce the amount of data, while the second covers those that increase the memory resources, through partitioning and/or distributing the process.

The first group includes solutions based on probabilistic data structures, the most explored Bloom filters. This data structure allows vertices to be stored independently of their number, but at the cost of false positives.

Revisiting the literature, some solutions were found that propose the use of external memory solutions to construct a graph with an exact representation. They process all the $k$-mers in external memory.

The data sets used to test these programs usually include mammals, bacteria, and human genomes, but the genome of plants has a limited presence on these works.

# 4 A novel approach for de Bruijn Graph construction

One of the aspects with the most significant impact in memory consumption for DBG construction is the number of elements ($k$-mers) that are necessary to process and the number of elements that it is necessary to maintain in main memory for fast access to obtain the nodes and edges of DBG. A vast number of $k$-mers need to be processed to identify the set of unique $k$-mers, which constitutes the set of vertices. Moreover, $k$-mers are units that encapsulate a high level of redundancy since they share $k-1$ bases.

In this chapter, we present our novel approach to construct the DBG without the necessity of processing all $k$-mers.

We first present the main idea of our approach in an intuitive way, for later formalizing it. To analyze our approach, we defined the number of elements processed, which impact on the run-time, and the number of unique elements, which defines the memory requirements. Our approach enforces the idea of using external memory only at the end of execution, when it is unavoidable, but taking advantage of available main memory.

We analyzed the number of I/O operations, which is reflected at the end of the chapter.

## 4.1 Motivation

$K$-mers are units that encapsulate a high level of redundancy. Two adjacent $k$-mers share $k-1$ bases. Thus, given a $(m-k+1)$ $k$-mers, in $m$ length read, we have $(m-k)(k-1)$ repeated bases. For $n$ reads, it becomes $(m-k)(k-1)n$ repeated bases. Because of this redundancy, there are $(m-k+1)k/m$ symbols by one symbol in dataset, substantially increases the amount of data to process. Since each vertex corresponds to a unique $k$-mer in DBG, generating and process each $k$-mer to construct the DBG implies a high level of redundancy in the process.

Regarding the $k$-mer itself, it is expected that they have a duplication level in correspondence with the coverage parameter passed in the sequencing. Since there are errors during the sequencing, the method tries to read some times the same base to get a more reliable value. However, only the unique

$k$-mers are keeping in the main memory during the DBG construction in the streaming approaches. Therefore, the number of repetitions is mapped as its multiplicity.

To illustrate, some values for the number of $k$-mers and unique $k$-mers, are exposed in the following experimental study using a sugarcane genome library with 2 million reads. The read length is 100, and we have used a sufficiently large $k$ variation, over 39 to 75. The number $k$-mers decreases while $k$ increases, as well as the number of unique $k$-mers, for same data sequences ( Fig. 4.1 and Fig. 4.2 ). Graph 4.2 also reveals that the number of unique $k$-mers does not exceed the 60% of the total number of $k$-mers for all $k$ values tested, proving the presence of high redundancy.

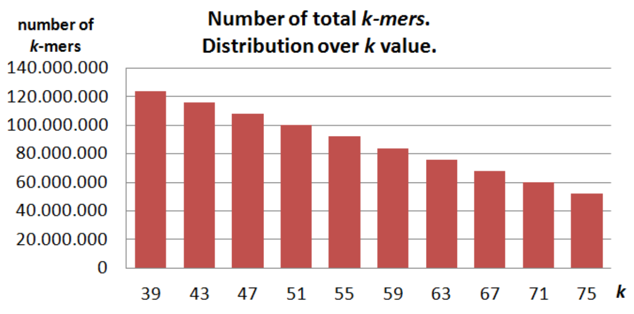


Figure 4.1: Number of total $k$-mers.

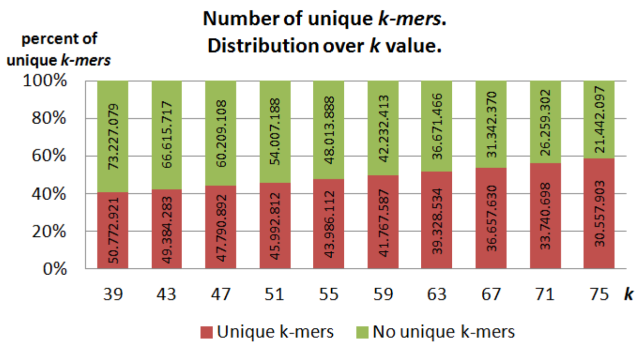


Figure 4.2: Unique $k$-mers distribution.

The construction of the DBG graph focuses on $k$-mer as its main element, since the collection of $k$-mers determines the set of $V$ and their $k-1$-length overlaps defines the edges.

The streaming approaches (4.5.1) to construct DBG, generates one $k$-mer at a time and process it independently. Therefore is executing as many operations as the number of total $k$-mers present in the dataset. However, the number of unique $k$-mers is the only one that is required to remain in the main memory.

The external memory DBG construction approaches, studied in Chapter 3, relies on the use of external memory sorting or on disk partitioning. Both works consider from the beginning the total number of $k$-mers, to following obtain the nodes (unique $k$-mers) and edges. For vertices and edges sorting in [Kundeti et al. 2010], the number of I/O operations will increase as the number of items to sort. The approaches based on $k$-mer disk partitioning generate all $k$-mers and then distribute them in partitions. Their processing focuses from the beginning on exploiting disk memory. How many partitions and memory size of each partition is unknown previously of its processing. As a partition will be at least one element to process, it will be executed at less one I/O per disk partition. The total disk size is the sum of the size of the partitions. Then, the maximum memory size needed to process a partition in the main memory is previously unknown, and it depends on the larger partition. If the largest partition does not fit in memory, it has to be split again, increasing the number of I/O operations. If some partitions are tiny, then they have to be grouped into a single partition file, increasing the number of possible I/O operations.

Working with the total number of $k$-mers implies maintaining a high level of redundancy and, consequently, a more significant amount of memory (RAM or external) and I/O operations.

In this chapter, it will be presented a novel approach to construct DBG for the genome assembly domain. This approach is based on the main idea that it is possible to build the DBG, for a specific set of sequences and $k$ value, iteratively processing a minor number of elements than the total number of $k$-mers.

## 4.2 Propositions of the new approach

Our approach for DBG construction for genome fragment assembly, is based on the following principles:

i *Find overlaps regions greater than $k$ earlier* can save the corresponding memory to store the redundant information for each $k$-mer and redundant information for consecutive $k$-mer chains that are duplicated.

ii *Avoid generate all* $k$-mers using iterative reduction steps.

iii *Use external processing, if it is necessary, only in the last steps of the current DBG construction approaches* with a minor number of elements.

This research has the hypothesis that it is possible to reduce runtime and memory requirements for DBG construction by reducing the number of elements to process iteratively from the beginning. It proposes the use of an external memory processing at final steps, for a smaller number of elements and when the number of presented elements are strictly unavoidable.

## 4.3 Main idea

Let be $R = \{r_1, r_2, ..., r_n\}$ a set of $m$ length reads, for $m << n$, and $k$ the value for the length of the $k$-mers, then:

- the number of $k$-mers in $R$ is $(m - k + 1) \times n$
- the number of bases to represent $R$ as $k$-mers is $(m - k + 1) \times n \times k$

If we have a substring $s_i$ of some $r$ in $R$, such as $0 \leq i < m - s + 1$, $|s_i| = s$ and $s > k$, then:

- the number of $k$-mers in $s_i$ is $(s - k + 1)$

then, if there at least another substring, $s_j$ of some $r$ in $R$, such as $0 \leq j < m - s + 1$, $|s_j| = s$ and $s > k$, we have also:

- the number of $k$-mers in $s_j$ is $(s - k + 1)$

In the case of $s_i = s_j$, we would not have to process these two substrings independently since they are equals, and we are interested in mapping unique substrings as vertices ($k$-length substring for DBG). If we are able to identify this kind of substring overlap early during the generation of DBG, then, it would be possible to avoid processing the repeated $k$-mers as independent units. Therefore, we can keep $s_i$, which implicitly contains $k$-mers, and depends on the number of repetitions of $s_i$, we have :

- $(s - k + 1) \times frequency(s_i)$ $k$-mers that will not have to be processed.
- $(s - k + 1) \times k \times frequency(s_i) - s$ bases that will not have to be processed.

A representation of this situation is showed in Fig. 4.3.a)

Since the construction of DBG implies to find read overlaps with a minimum size $k$ and $s > k$, it is necessary to decompose $s_i$ with the aim to analyze substrings of $s_i$. Following we analyze this context:

If there is a substring $l_g$ of $s_i$, $0 <= g < s - l + 1$, such that $|l_g| = l$, $k \leq l < s$ and $l_g$ has external copies (not only in repetitions of $s_i$), we should to decompose $s_i$ to find these $l_g$ duplications. Notwithstanding the above, it is possible to skip elements from being processed, since we would:

– $(s-k+1)\times(frequency(s_i)-1)$ $k$-mers that will not have to be processed.
– $(s-k+1)\times k\times(frequency(s_i)-1)$ bases that will not have to be processed.

Although we have to decompose $s_i$ to search overlaps of length $l$, we still have a set with a significant number of $k$-mers that we will not have to be processed. A representation of this situation is shown in Fig. 4.3.b)

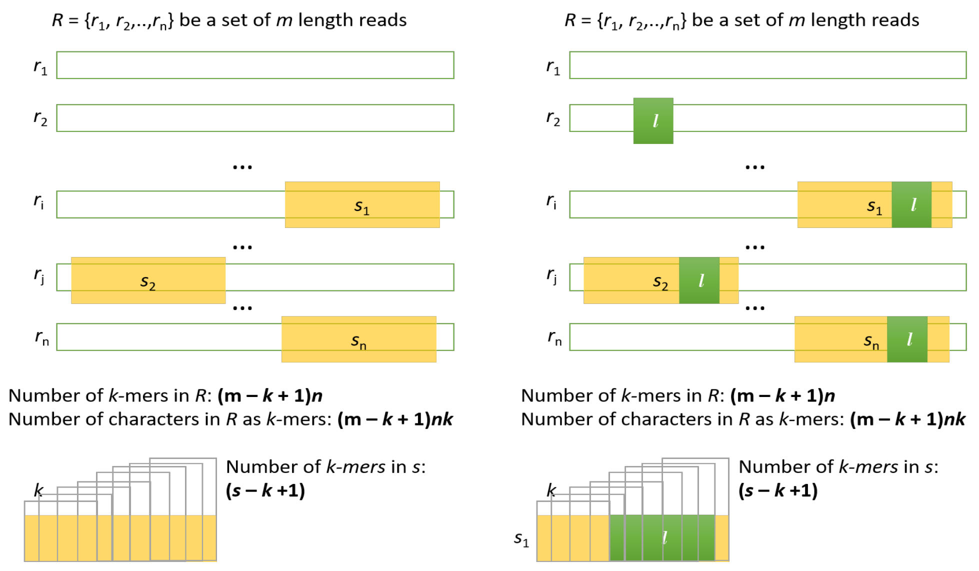

R = {r1, r2,..,rn} be a set of m length reads
r1
r2
ri
rj
rn
s1
s2
sn
l
Number of k-mers in R: (m – k + 1)n
Number of characters in R as k-mers: (m – k + 1)nk
Number of k-mers in s: (s – k +1)
k

Figure 4.3: Analysis of the number of $k$-mers and characters that could be skipped to be processed. a) case of $s_1$ with some duplications. b) case of there is a substring $l_1$ of $s_1$, such as $|l_i| = l$, $k \leq l < s$ and $l_1$ has external copies

Applying the shown basic steps, we can propose the following pipeline. First, each $m$-length read could be decomposed in $s$-length substrings. Then it is possible to find repetitions of length $s$, keeping with the set of unique elements. Each unique $s_i$, it is possible to decompose it in minor elements to find repetitions with a minor length. Iteratively, we can reduce $s$ in each step, until it is reached $k$ length. For each step, it would be possible to decompose each element results from the previous iteration in a new group of elements. Since only one copy of duplicates elements generated in the current iteration will be kept, the number of elements to process will be reduced gradually in each iteration. In the last iteration, when all decompositions have done, we will have discarded all duplicated elements. Due to that, it will be avoided to have to process all $k$-mers contained in input reads, and it is obtained the unique $k$-mers set, which corresponds to the DBG vertices set.

Besides, we intend to reduce the redundancy during the generation of $k$-mers. $K$-mers generation process implies new $k$-mer for each position in $m$-

length read, until reach $m-k$ position. In other words, $k$-mers are generated as the context of a sliding window of $k$-length, which is moved right one position each time. Therefore, an $(k-1)$-length overlap is present in two adjacent $k$-mers, that links two respective vertices in DBG. Using the same idea, we could generate $s$-length substrings for each $m$-length read, getting a $(s\text{-}1)$-length overlap between adjacent $s$-substring and generating $m-s+1$ substrings for each read. However, it implies a redundancy equivalent to using a $k$ value equal to $s$. To afford these, we define a distance between two elements based on $(k\text{-}1)$-length overlap for adjacent $s$-length substrings. The idea is to maintain only the link between two consecutive $s$-length substrings, $s_i$, and $s_j$, using the same criteria used to define an edge between two adjacent $k$-mers. At that point of view, this link corresponds to the edge generated by the tail $k$-mer included in $s_i$ and the head $k$-mer include in $s_j$. In that way, we are not going to maintain more redundancy for edges than what already existed is in the DBG edge definition. Those links would already constitute a subset of the edges set of final DBG. As we would decompose $s$-substring in elements with a length $l$, $k \leq l < s$, in the next iteration, following the same criteria defined above, it is warrantied that links for each $l$-length substring will be maintenance. Iteratively, the collection of links would be increased, adding new links to the previous iteration link set. Keeping a set of links implies that a single copy of duplicate links will always be maintained. At the last iteration, being $l = k$ would be made all decompositions until reach a set of $k$-mers and a set of links between them, being $V$ and $E$ of $G_k(V,E)$ correspondingly.

## 4.4 Novel algorithm for DBG construction

Based on the idea explained in 4.3, it is proposed a new algorithm for building the *de Bruijn* graph, avoiding to generate all $k$-mers.

**Definition 4.1** ***dk-mer.*** *A dk-mer is a substring of a genome piece with specified d length (also called dimension in this thesis), with $d \geq k$, over the alphabet of bases $\Sigma = \{A, T, C, G\}$.*

*Two dk-mers, $dk-mer_1$ and $dk-mer_2$ are adjacent if they share $k-1$ bases ($k-1$ suffix of the first is equal to the $k-1$ prefix of the second) and they are adjacent in at least one read.*

**Definition 4.2** ***extra-compacted de Bruijn Graph.*** *An extra-compacted de Bruijn Graph $G_{d,k}(V_{d,k}, E_{d,k})$ is a graph $G(V,E)$ in which the set of vertices $V$ corresponds to unique dk-mers of length minor or equals to d, and the set of edges $E$ corresponds to unique edges of dk-mers. Two dk-mers have an edge if they are adjacent, sharing $k-1$ overlap.*

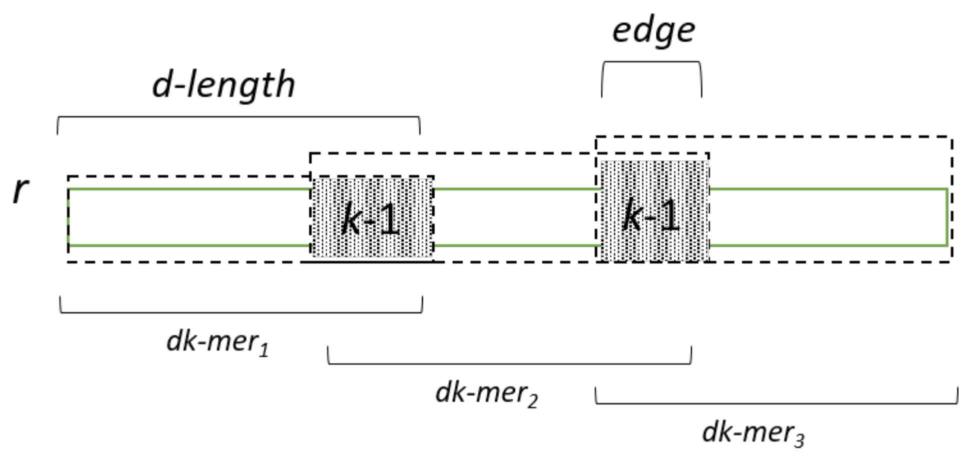


Figure 4.4: $dk$-mers representation over read $r$. The $dk$-mers have dimension $d$. Adjacent $dk$-mers share $k - 1$ bases which defines an edge.

The extra-compacted *de Bruijn* Graph definition is introduced in this work to refer to the graph generated in each intermediate iterations. The last iteration generates a DBG. The new algorithm steps are described below:

- Search overlaps regions with length $d_1$, $k < d_1 < m$, generating one vertex for each unique $d_1k$-mers and applying the suffix-prefix overlap of $(k - 1)$ length criteria to generate the edges. The result is $d_1k$-mers vertices and edges sets of extra-compacted *de Bruijn* Graph $G_{d_1,k}$.
- Search overlaps regions with size $d_2$, $k < d_2 < d_1$, decomposing each vertex in $V_{d_1,k}$ into $d_2k$-mers. Generate one vertex for each unique $d_2k$-mers to get a set of vertices and apply the suffix-prefix overlap of $(k-1)$ length criteria to get the set of new edges. The union of new edges and $E_{d_1,k}$ generates $E_{d_2,k}$. The result is $d_2k$-mers vertices and edges sets of extra-compacted *de Bruijn* Graph $G_{d_2k}$.
- Search iteratively duplicated regions with size $d_i$, $k < d_i < d_2 < d_1$, decomposing each vertex in $V_{d_{i-1},k}$ from into $d_ik$-mers. Generate one vertex for each unique $d_ik$-mers to get a set of vertices and apply the suffix-prefix overlap of $(k - 1)$ length criteria to get the set of edges, adding the edges from $G_{d_{i-1},k}$. The result is $d_ik$-mers vertices and edges sets of extra-compacted *de Bruijn* Graph $G_{d_i,k}$.
- Search for $k$ overlaps at last iteration with $d_z = k$, and $d_z < .. < d_i < ... < d_2 < d_1$, decomposing each vertex in $V_{d_{z-1},k}$ into $d_zk$-mers. Generate one vertex for each unique $d_zk$-mers to get a set of vertices and apply the suffix-prefix overlap of $(k - 1)$ length criteria to get the set of edges, adding the edges from $G_{d_{z-1},k}$. Since edges were generated using the suffix-prefix overlap of $(k - 1)$ length criteria that appear at least in one

read, and $V_{d_{z-1},k}$ corresponds to the set of unique $k$-mers due to $d_z = k$, the result of this steps is a DBG $G_k(V, E)$ (2.3).

It is worth to note that the suffix-prefix overlap of $(k-1)$ length criteria for edges mentioned above, implies that this overlap exists in at least one read.

The figure 4.5 illustrates the general process showing how the new algorithm, through some iterations, generates the same DBG $G_k(V, E)$ that is generated using traditional approaches.

In traditional approaches, from right to left in the figure, each $k$-mer is processed in the collection $C_v$ of all $k$-mers in reads $R$. In that way, it is obtained $V$, which is a set of $k$-mers from $C_v$ (set implies not duplication), and $E$, which is a set of edges from $C_e$, the collection of all edges extracted from $R$.

Our approach, from left to the right in the figure, starts using a dimension value $d_1$ to generate a set of vertices $V_{d_1,k}$ of $G_{d_1,k}(V_{d_1,k}, E_{d_1 k})$, discriminating a certain number of elements of $C_v$ that would be processed in traditional approaches. In that sense, we can say that we reduce the redundancy at this level of $d_1$. Being *extract* a function that extracts the $k$-mers implicitly contained in a set of vertices, we have $C_{V_{d_1,k}} = extract(V_{d_1,k})$, such as that $|C_{V_{d_1,k}}| < |C_v|$. Besides, it is generated the set of edges $E_{d_1,k}$ of $G_{d_1,k}(V_{d_1,k}, E_{d_1,k})$, we have that $E_{d_1,k} \subset E$. In terms of collections, to be possible to compare, if we define $C_{E_{d_1,k}} = E_{d_1,k}$, then we have $|C_{E_{d_1,k}}| < |C_e|$.

From the resulted $G_{d_1,k}(V_{d_1,k}, E_{d_1,k})$, then it is decomposing each $V_{d_1,k}$ using a new dimension value $d_2$ which $d_2 < d_1$, obtaining $G_{d_2,k}(V_{d_2,k}, E_{d_2,k})$. In terms of vertices it is possible to see that $|C_{V_{d_1,k}}| < |C_{V_{d_2,k}}| < |C_v|$. In terms of edges, since $E_{d_2,k}$ is the result to add new edges to $E_{d_1 k}$, doing $C_{E_{d_2,k}} = E_{d_2,k}$, we have $E_{d_1,k} \subset E_{d_2,k} \subset E$ and $|C_{E_{d_1,k}}| < |C_{E_{d_2,k}}| < |C_e|$. Following the above reasoning, for $z$ iterations, it is resulted in $|C_{E_{d_1,k}}| < |C_{E_{d_2,k}}| < ... < |C_{E_{d_i,k}}| < ... < |C_{E_{d_z,k}}| < |C_e|$ and $|C_{V_{d_1,k}}| < |C_{V_{d_2,k}}| < ... < |C_{V_{d_i,k}}| < ... < |C_{V_{d_z,k}}| < |C_v|$, showing how our algorithm for each iteration always process a minor number of elements than $|C_v|$ and $|C_e|$.

In each iteration will be obtained a set of $d_i k$-mers which elements, implicitly, including a subcollection of the collection of a total number of $k$-mers and edges, in a compacted way, and were the redundancy iteratively will be eliminated until it is reached the same DBG as the traditional approaches do.

***Algorithm sketch***

The approach is presented through the next algorithm sketch, divided in three sections (see Algorithms 1, 2 and 3). Algorithm in 1 is the main block, while the others are detailed parts of the main.

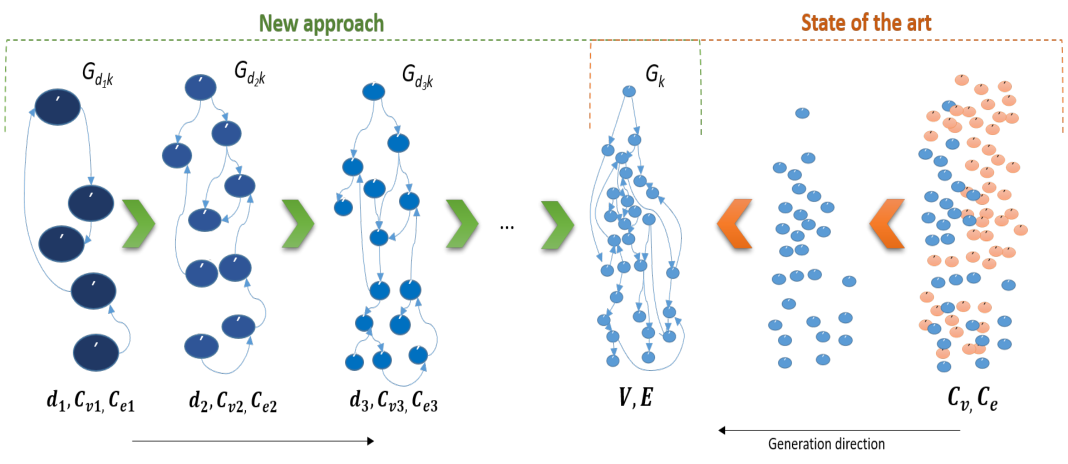


Figure 4.5: DBG construction representation. On the left hand, from left to right, is represented how the proposed new algorithm generates DBG. From right to left, the traditional generation approaches processing the entire collection of $k$-mers. In the middle, the intersection is the final DBG.

Given $R = \{r_1, r_2, .., r_n\}$ a set of $m$ length reads

**Algorithm 1** Routine *buildDBG*

```
 1: $i = 1$
 2: $d_i$ such as $k \leq d_i < m$
 3: $initialize\ V_i$
 4: $initialize\ E_i$
 5: for each $s$ substring length $d_i$ and overlap $k-1$ in $R$ do
 6:     if $s$ does not exist in $V_i$ then
 7:         $V_i = V_i \cup buildVertice(s)$
 8:         $updateEdges(E_i,\ edge(s_{current},\ s_{next})\ )$
 9: $update(d_i,\ step)$
10: while $d_i \geq k$ do
11:     $initialize\ V_i$
12:     $initialize\ E_i$
13:     for each $v$ in $V_i$ do
14:         $decompose(d_i,\ v)$
15:     $update(d_i,\ step)$
```

**Algorithm 2** Routine *decompose*$(d_i,\ v)$

```
1: for each $s$ substring length $d_i$ and overlap $k-1$ in $v$ do
2:     if $s$ does not exist in $V_i$ then
3:         $V_i = V_i \cup buildvertice(s)$
4:     $updateEdges(E_i,\ edge(s_{current},\ s_{next})\ )$
5:     if $s$ is tail then
6:         $update\ associated\ vertices\ from\ E_{i-1}$ in $E_i$
```

In this algorithm sketch, we used $s$ instead of $d_ik$-mer for simplicity.

The step related to updating associated vertices from $E_{i-1}$ in $E_i$ has the objective to collect the vertices generated in the previous iteration and transform in edges according to the current iteration transitively. In that way, the graph created in each iteration will converge to DBG.

In that sense, it is essential to define data structures for $V_i$ and $E_i$, with the lowest overhead costs and asymptotic constant search time. The algorithm has two important input parameters, $d_1$, and function to update $d_i$. This function could depend on a second parameter, the *step*, to reduce $d_i$, which can influence the number of $k$-mers that will be avoided from being processed. As $d_i$ come close to $k$, the contraction of $dk$-mers will be increased, it could take less advantage of the approach. On the other hand, if it used a high value for $d$, and there are no large regions duplicated, then the algorithm may lose its advantages over traditional approaches due to multiple iterations without reducing the amount of data to process.

This new algorithm has the following specifications:

1. Input of the algorithm: fragment sequences file (reads) for iteration with $i = 1$

2. Output of the algorithm: *de Bruijn* Graph for last iteration for $d_z = k$

Intermediate iterations will have:

1. Input: vertices and edges of previous iteration for $i > 1$ .

2. Output:

   – $G_{d_i,k}$ with vertices and edges updated for this iteration.
   – Number of nodes (unique number of $dk$-mers).
   – Number of total $dk$-mers processed.

3. At the end of each iteration, $d$ is updated using a function $d_{i+1} = update(d_i, step)$

### 4.4.1 Extra-compacted de Bruijn Graph decomposition analysis

#### 4.4.1.1 Decomposition in details

In each iteration it is getting the $d_i k$-mers by decomposition of each read for $i = 1$, or by decomposition of each $d_{i-1}k$-mer in $V_{d_i,k}$ for $i > 1, i \leq z$. A detail of how to generate each $dk$-mer is showed in Algorithm 3, which is an expand of how is obtained $s$ in line 5 for Algorithm 1 and line 1 for Algorithm 2. The term *process* is used to encapsulate the line block from 6 to 8 in Algorithm 1 and 2 to 6 in Algorithm 2.

**Algorithm 3** *Detailed decomposition of* $(s,d_i)$

```
1: $len = length(s)$
2: $alfa = k - 1$
3: $ganma = d_i - alfa$
4: if $len > dimension$ then
5:     //decompose
6:     for $j = 0$ ; $j < len - d_i + 1; i+ = ganma$ do
7:         //processing complete $dk$-mers with $length = d$
8:         $dk - mer =$ new $dk - mer(source = s, startpoint = i, length = d_i)$
9:         $process(dk - mer)$
10:    if $(len - d_i)$ $mod$ $ganma > 0$ then
11:        //processing final $dk$-mers with $length < d$
12:        $dk - mer =$ new dk-mer (source = s, startpoint = i, len - i)
13:        $process(dk - mer)$
14: else
15:    //processing $dk$-mers with $length < d$ inherited from a previous
    iteration
16:    $process(s)$
```

In the pseudo-algorithm 3 was defined $\alpha$ as **overlap value** between two consecutive $dk$-mer, such as $\alpha = k - 1$.

A visual representation of the decomposition is showed in a sequence of figures from ( 4.6 to 4.12). For the purpose of illustration of how the algorithm decomposition goes, we applied the algorithm over just one read $r$ with 79 bases with the input parameter $k = 12$. Also, it is assumed that this read has no repeated regions. We use the function $update(d_{i-1}, step)$ (Equation 4.4.1.1) with $step = 10$ to update $d$ in each iteration for $i > 1$, and set $d_1 = 64$. For instance, we described the input and the output of each iteration $i$ such as a list of elements ($r$ for $i = 0$, otherwise $dk$-mers), using the absolute indexes over $r$ and the length of the element. Hence, each element ($dk$-mer) is specified by a triple: [*start position index*, *last position index*, *length*]. The edges are specified by a tuple as [*start position index*, *last position index*], in which is contained the first and the last index of the vertices that it links.

$$d_i = update(d_{i-1}, step) = \begin{cases} k & d_{i-1} - step < k \\ d_{i-1} - step & \text{otherwise.} \end{cases}$$

The input of the first iteration (see Fig. 4.6), is one read $r$ and $d_1 = 64$. From this input were generated two $dk$-mers by decomposition of $r$, one $dk$-mer contains 64 bases, from 0 to 63 position, and the second $dk$-mer contains 26 bases, from 53 to 78 position. Since there are no duplications, $V_{d_1,k} = \{[0, 63, 64], [53, 78, 26]\}$ for the extra-compacted $G_{d_1,k}$ generated in this iteration. Besides, just one edge was generated, $E_{d_1,k} = \{[0, 78]\}$. In the illustration, the orange sequences represent the input $dk$-mers, while green represent generated $dk$-mers, which constitute the output of the iteration. Also, shaded regions represent the edges. The elements outputted in each iteration will be registered in Table A.1 in the Appendix.

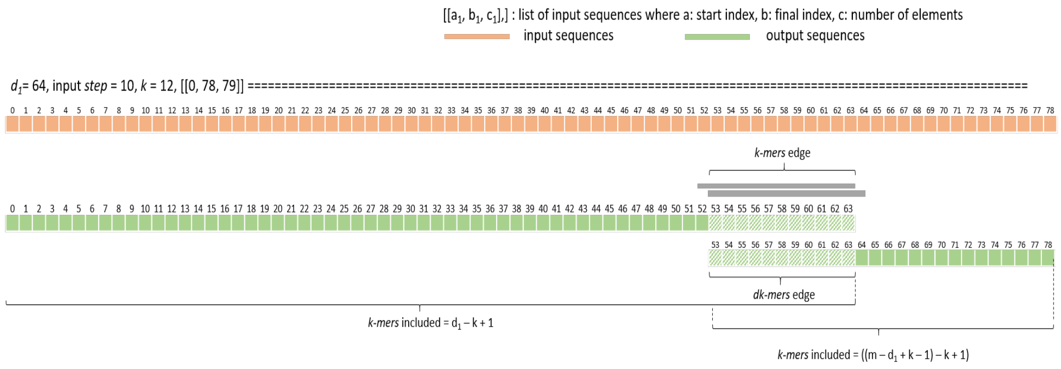

[[a1, b1, c1],] : list of input sequences where a: start index, b: final index, c: number of elements
input sequences
output sequences
d1= 64, input step = 10, k = 12, [[0, 78, 79]]
k-mers edge
dk-mers edge
k-mers included = d1 – k + 1
k-mers included = ((m – d1 + k – 1) – k + 1)

Figure 4.6: Iteration 1: Getting $dk$-mers with $d_1 = 64$ from one read with $m = 79$.

The input of the second iteration (Fig. 4.7), $i = 2$, is the output of iteration $i = 1$. Dimension value was updated to $d_2 = 54$. From the first element in the input, were generated two $dk$-mers, one $dk$-mer containing 54 bases, from 0 to 53 position, and the second $dk$-mer containing 21 bases, from 43 to 63 position. Because the second element of the input have a minor length than $d_2$, it is passed to the set of elements generated without be decomposed. Since there are no duplications, $V_{d_2,k} = \{[0, 53, 54], [43, 63, 21], [53, 78, 26]\}$ for the extra-compacted $G_{d_2,k}$ . Due to one vertex with edge was decomposed, this edge must be updated became [43, 78] from [0,78]. Also, a new edge [0,53] was generated, which was added to $E_{d_1,k}$. Then, $E_{d_2,k} = \{[0, 63], [43, 78]\}$

Third and fourth iterations (Fig. 4.8 and Fig. 4.9) occurs in the similar way as $i = 2$. For $i = 3$, dimension value was updated to $d_3 = 44$. From the input [[0, 53, 54], [43, 63, 21], [53, 78, 26]], just only the first element could be

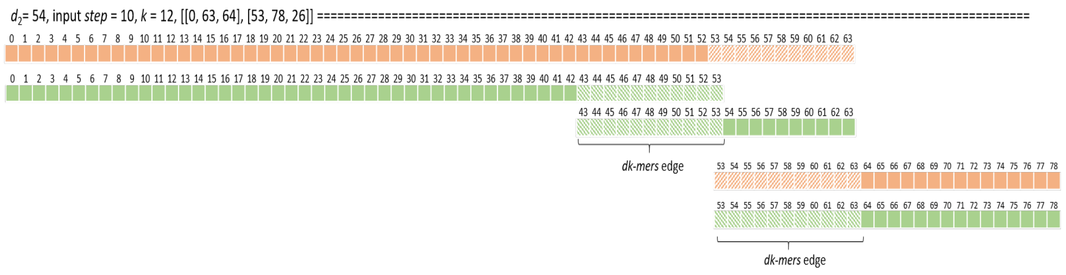

$d_2$= 54, input *step* = 10, *k* = 12, [[0, 63, 64], [53, 78, 26]]
*dk-mers* edge
*dk-mers* edge

Figure 4.7: Iteration 2: Getting $dk$-mers with $d_2 = 54$ from a set of two $dk$-mers.

decomposed because its length is greater than $d_3$, generating two $dk$-mers. The rest of the elements are included in the result, but they are no decomposed. Also the edge associated with the $dk$-mer decomposed is updated (Table A.1). Similar, from the output of third iteration, for $i = 4$, and $d_4 = 34$, is only decomposed the first element of the input. The resulting $V_{d_4,k}$ and $E_{d_4,k}$ are listed in Table A.1). At that point the length of $dk$-mers begin to converge to 21 bases.

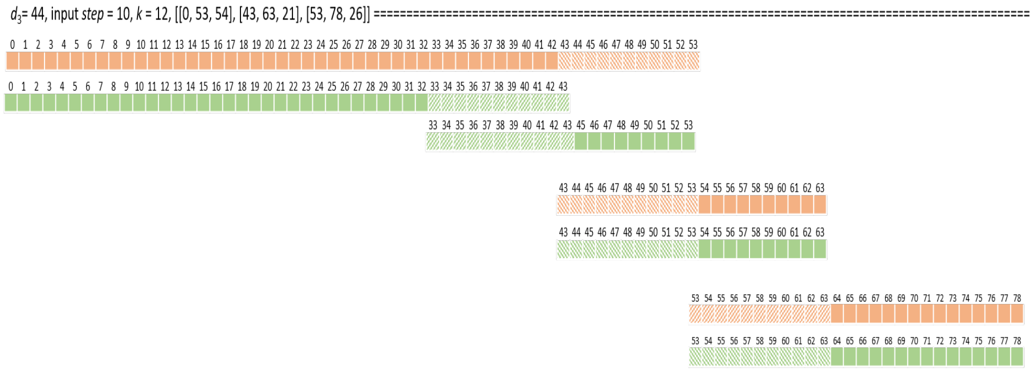

$d_3$= 44, input *step* = 10, *k* = 12, [[0, 53, 54], [43, 63, 21], [53, 78, 26]]

Figure 4.8: Iteration 3: Getting $dk$-mers with $d_3 = 44$ from a set of three $dk$-mers.

In the iteration $i = 4$, with $d_4 = 24$ (Fig. 4.10), were decomposed two elements from the input. From [0, 53, 54] it is generated two new $dk$-mers, [0, 23, 24] and [13, 33, 21], while from [53, 78, 26] is generated two new $dk$-mers, [53, 76, 24], [66, 78, 13]. The resulting set of vertices and edges are listed in Table A.1.

Iteration $i = 5$, is the first iteration that manages to decompose most of the input elements, since $d_5 = 14$ is getting significantly closer to the value of $k$. In Fig. 4.11, is shown how the first and the penultimate element from the input were decomposed in 5 $dk$-mers, while the remaining elements with length 21 were decomposed in 4 $dk$-mers. Just one element from the input was not decomposed. The resulting set of vertices and edges are listed in Table A.1.

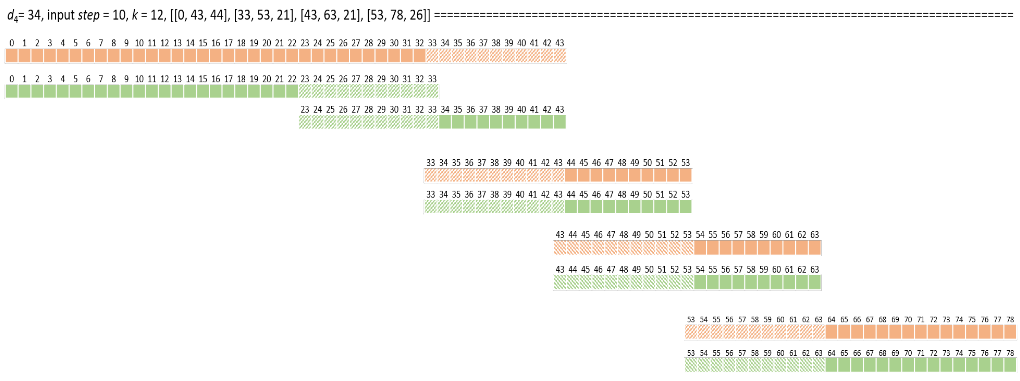

d4= 34, input step = 10, k = 12, [[0, 43, 44], [33, 53, 21], [43, 63, 21], [53, 78, 26]]

Figure 4.9: Iteration 4: Getting $dk$-mers with $d_4 = 34$ from a set of four $dk$-mers.

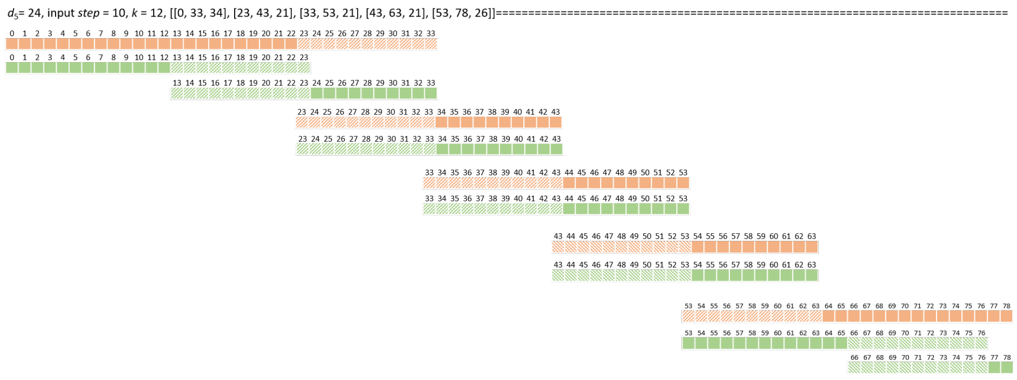

d5= 24, input step = 10, k = 12, [[0, 33, 34], [23, 43, 21], [33, 53, 21], [43, 63, 21], [53, 78, 26]]

Figure 4.10: Iteration 5: Getting $dk$-mers with $d_5 = 24$ from a set of five $dk$-mers.

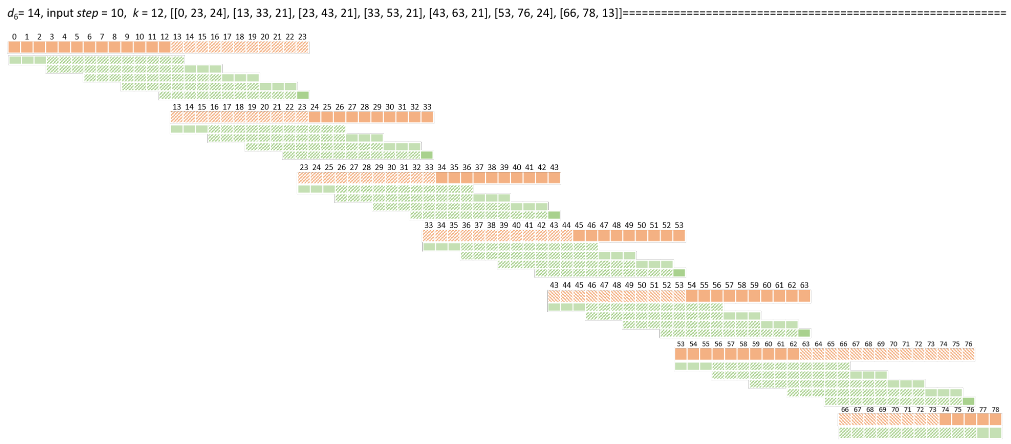

d6= 14, input step = 10, k = 12, [[0, 23, 24], [13, 33, 21], [23, 43, 21], [33, 53, 21], [43, 63, 21], [53, 76, 24], [66, 78, 13]]

Figure 4.11: Iteration 6: Getting $dk$-mers with $d_6 = 14$ from a set of seven $dk$-mers.

Since $d$ reached its last possible number using the reduction by $step = 10$ ($d_5 = 14$, such as $14 - 10 < k$, then $d_5 = k$), then $d_6 = 12$. This is the last iteration (Fig. 4.12), in which will generated finals $k$-mers and corresponding

DBG. All elements from the input that have not reach yet the length equals to $k$, will be decomposed in $dk$-mers (in this last iteration $dk$-mers $= k$-mer). At the end of this iteration, its obtained 68 $k$-mers and 67 edges. The elements of vertices and edges are listed in Table A.1.

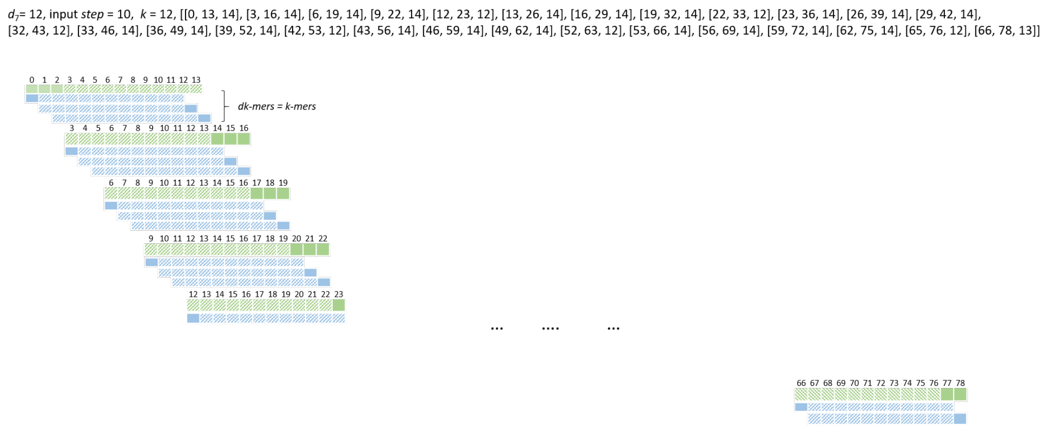

d7= 12, input step = 10, k = 12, [[0, 13, 14], [3, 16, 14], [6, 19, 14], [9, 22, 14], [12, 23, 12], [13, 26, 14], [16, 29, 14], [19, 32, 14], [22, 33, 12], [23, 36, 14], [26, 39, 14], [29, 42, 14], [32, 43, 12], [33, 46, 14], [36, 49, 14], [39, 52, 14], [42, 53, 12], [43, 56, 14], [46, 59, 14], [49, 62, 14], [52, 63, 12], [53, 66, 14], [56, 69, 14], [59, 72, 14], [62, 75, 14], [65, 76, 12], [66, 78, 13]]
dk-mers = k-mers
...
....
...

Figure 4.12: Iteration 7: Getting $dk$-mers with $d_7 = 12$ from a set of twenty seven $dk$-mers.

Summarizing, as we can see in Table 4.1, the number of $dk$-mers processed in each iteration is minor that the number of total $k$-mers, 68 in iteration 6. Because we decided to use a read without duplications for simplicity, in this case, our algorithm is inefficient. The total number of $k$-mers of the read is equal to the number of unique $k$-mers, because of no repetitions. Traditional approaches would have processed 68 $k$-mers while our algorithm had to process an amount of 48 elements until the last iteration with 68 elements processed, totaling 116 elements. In this case, our algorithm will bring gains over traditional approaches. The number of elements processed will be a subject of discussion in the following subsections.

The results for a simulation using others values for *step* to update $d$, keeping constant $d_1$, $k$, the same input read length, and the condition of no duplications, is showed in Table A.2 in the Appendix. It is possible to view how variates the number of elements generated in each iteration, and the total elements treated in the process as a whole. In that sense, for the smallest number of iterations, the smallest number of elements in the process as a whole will be processed. However, other variables of our model could impact the result of each iteration, like the replication factor and the number of unique elements that will be discussed in the following sections. These values that depend on the distribution of duplications will determine the number of skipped $k$-mers, i.e., the number of $k$-mers that will be avoided from being processed.

Table 4.1: Iteration details.

| $m$ | $k$ | $i$ | $d$ | input | | output | |
|---|---|---|---|---|---|---|---|
| | | | | number of elements | length of elements | number of elements | length of elements |
| 79 | 12 | 1 | 64 | 1 | 79 | 1 | 64 |
| | | | | | | 1 | 26 |
| | | 2 | 54 | 1 | 64 | 1 | 54 |
| | | | | 1 | 26 | 1 | 26 |
| | | | | | | 1 | 21 |
| | | 3 | 44 | 1 | 54 | 1 | 44 |
| | | | | 1 | 26 | 1 | 26 |
| | | | | 1 | 21 | 2 | 21 |
| | | 4 | 34 | 1 | 44 | 1 | 34 |
| | | | | 1 | 26 | 1 | 26 |
| | | | | 2 | 21 | 3 | 21 |
| | | 5 | 24 | 1 | 34 | 2 | 24 |
| | | | | 1 | 26 | 4 | 21 |
| | | | | 3 | 21 | 1 | 13 |
| | | 6 | 14 | 2 | 24 | 20 | 14 |
| | | | | 4 | 21 | 1 | 13 |
| | | | | 1 | 13 | 6 | 12 |
| | | 7 | 12 | 20 | 14 | 68 | 12 |
| | | | | 1 | 13 | | |
| | | | | 6 | 12 | | |

### 4.4.2 Analysis of the number of vertices per iteration

A certain number of $dk$-mers will be generated in each iteration $i$ for $n$ reads and $d_i \in D = \{d_1, d_2, d_3, ..., d_{z-1}, d_z\}, d_z = k$. The set of these elements corresponds to the number of nodes of each $G_{d_i,k}(V_{d_i,k}, E_{d_i,k})$. Since decomposition may not generate only entire $d$ length $dk$-mers, we define:

**Definition 4.3** *Complete dk-mer:*

*dk-mers that have $d_i$ length for $i$ iteration*

**Definition 4.4** *Partial dk-kmer:*

*dk-mers have length $< d_i$ for $i$ iteration*

Be $m$ the length of one sequence of bases, for specific $d$, and $\alpha = k - 1$ (constant for all iterations), the number of *completed dk-mers* generated from these sequence is given by $f(d, m)$ (equation 4.4.2), while the number of *partial dk-mers*, is given by $f'(d, m)$ (equation 4.4.2). With not duplications, we have that the number of vertices of $G_{dk}$ for these sequence will be is $f(d) + f'(d)$:

Number of completed $dk$-mers for sequence of length $m$ and $d$ (for clarifications view Fig. 4.13):

$$f(d,m) = \begin{cases} 0 & m < d \\ \left\lfloor \left(\frac{m-d}{d-\alpha}\right) + 1 \right\rfloor & \text{otherwise.} \end{cases}$$

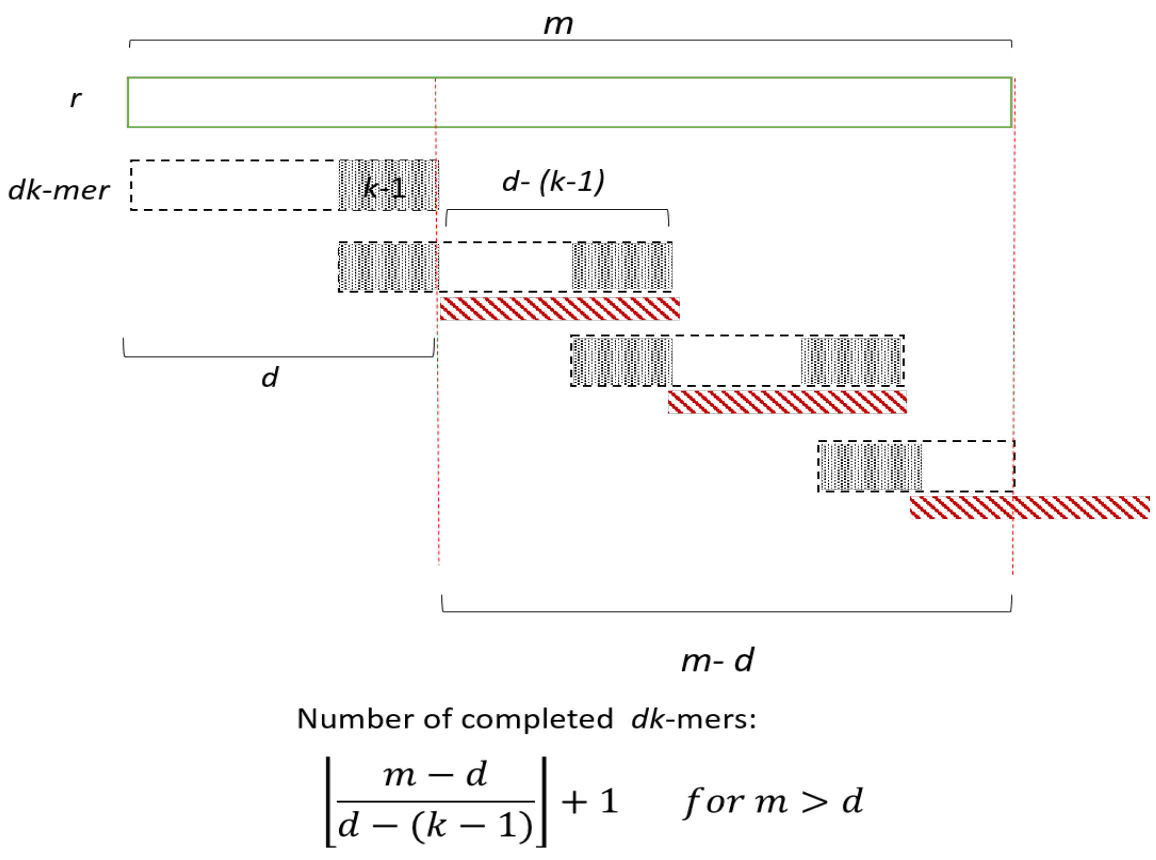

m
r
dk-mer
k-1
d- (k-1)
d
m- d
Number of completed dk-mers:
$\left\lfloor \frac{m-d}{d-(k-1)} \right\rfloor + 1 \quad for\ m > d$

Figure 4.13: Number of complete $dk$-mer. Formula representation.

Number of partial $dk$-mer for sequence of length $m$ and $d$:

$$f'(d,m) = \begin{cases} 1 & m < d \quad \text{or} \quad (m-d) \mod (d-\alpha) > 0 \\ 0 & \text{otherwise.} \end{cases}$$

The length of partial $dk$-mer of sequence of length $m$ and $d$ is given by:

$$e(d,m) = \begin{cases} m & m < d \\ ((m-d) \mod (d-\alpha)) + \alpha & (m-d) \mod (d-\alpha) > 0 \\ 0 & \text{otherwise.} \end{cases}$$

The number of $dk$-mer generated in each iteration depends on the number and length of the *dk*-mers generated in the previous iteration. Also, as we always keep with the set of unique *dk*-mers, the number of *dk*-mers generated will be filtered by a factor of duplication, decreasing the number of *dk*-mers that will be the inputted in the next iteration. For comprehension, we will start to analyze the number of *dk*-mers assuming that there are no duplications.

Being $m$ the length of input reads of $i = 0$, for each iteration $i$, we defined $W_i = (w_{i,1}, w_{i,2}, ..., w_{i,m-1}, w_{i,m})$ as a $m$-dimensional vector, where $w_{i,j}$ contains the number of *dk*-mers with length $j$, for $j \in [1, m]$. Then, $\forall i, j$, $W_i$ is initialized such as $w_{i,j} = 0$, and be $W_0$, such as $w_{0,m} = n$, assuming that all $n$ reads have length $m$.

At the end of the iteration, $W_i$ will be updated through $t$, as follow:

$$t(W_i, W_{i-1}, d_i) = \begin{cases} \forall j, w_{i-1,j} > 0, j > d_i : & \\ \quad w_{i,d} = f(d_i, j) * w_{i-1,j} & \text{number of completed } dk\text{-mers} \\ \quad w_{i,e(d_i,j)} = f'(d_i, j) * w_{i-1,j} & \text{number of partial } dk\text{-mers} \\ & \\ \forall j, w_{i-1,j} > 0, j \leq d_i : & \\ \quad w_{i,j} = w_{i-1,j} \quad \text{partial } dk\text{-mers inherited} & \end{cases}$$

Then, for each iteration, the total number of *dk*-mers is:

$$g(i) = \sum_{j=1}^{m} w_{i,j} \quad \text{in } W_i\text{, total number of } dk\text{-mers} \tag{4-1}$$

where $w_{i,j}$ is updated through $t(W_i, W_{i-1}, d_i)$, being $W_0$ such as $\forall i$, $j \neq m$, $w_{0,j} = 0$, $w_{0,m} = n$.

The accumulated number of elements processed until given iteration $i$ is:

$$\mathcal{G} = \sum_{i=1}^{z} g(i) \quad \text{for } d \in D = \{d_1, d_2, d_3, ..., d_i, ..., d_{z-1}, d_z\}, d_z = k \tag{4-2}$$

Due to the coverage sequencing and replication inside genome sequences, we assume that exists an important number of *dk*-mer duplications in each iteration. The duplications degree will impact on the number of vertices for each $G_{d_i,k}$, and the degree of $dk$-mer duplications for the next iteration. At that point, taking into account duplications in our analysis, we introduce a concept of $dk$-mers replication factor $B$, to measure the degree of $dk$-mers duplications. The $dk$-mers replication factor is the value by which we divide the number of $dk$-mers for one iteration and gives us the unique number of $dk$-mers for this iteration, i.e., the number of vertices for each $G_{d_ik}$. Then, the number of elements in one iteration depends on the number of unique elements in previous iterations for each $dk$-mer length. Moreover, the number of duplications in iteration $i+1$ will be reduced by the filtrations of duplications in iteration $i$, because duplicated $d_{i+1}k$-mers include duplicated $d_ik$-mers, but not all duplications in $d_ik$-mers will be filtered in previous iteration.

Taking into account the unique elements, for each iteration $i$, we defined $\mathcal{U}_i = (u_{i,1}, u_{i,2}, ..., u_{i,m-1}, u_{i,m})$ as a $m$-dimensional vector, where $u_{i,j}$ contains the number of unique $dk$-mers with length $j$, for $j \in [1, m]$. Then, $\forall i, j$, $\mathcal{U}_i$ is initialized such as $u_{i,j} = 0$, and be $\mathcal{U}_0$, such as $u_{0,m} = n$, assuming that all $n$ reads have length $m$.

Including the replication factor in our analysis, for each $dk$-mer length, there is a specific replication factor $\beta$-length that impacts in the values of $\mathcal{U}_i$ and $W_i$. At the end of the iteration, $\mathcal{U}_i$ will be updated through $t'$ (4.4.2). The $t'$ function takes into account the impact of $\beta$, as follow:

$$t'(\mathcal{U}_i, \mathcal{U}_{i-1}, d_i) = \begin{cases} \forall j, u_{i-1,j} > 0, j > d_i : \\ \quad u_{i,d} = \frac{f(d_i,j) * u_{i-1,j}}{\beta_{i,j}} \quad \text{number of completed } dk\text{-mers} \\ \quad u_{i,e(d_i,j)} = \frac{f'(d_i,j) * u_{i-1,j}}{\beta_{i,j}} \quad \text{number of partial } dk\text{-mers} \\ \\ \forall j, u_{i-1,j} > 0, j \leq d_i : \\ \quad u_{i,j} = u_{i-1,j} \quad \text{partial } dk\text{-mers inherited} \end{cases}$$

$$g'(i) = \sum_{j=1}^{m} u_{i,j} \quad \text{in } \mathcal{U}_i\text{, number of unique } dk\text{-mers for } i \text{ iteration} \tag{4-3}$$

where $u_{i,j}$ is updated through $t'(\mathcal{U}_i, \mathcal{U}_{i-1}, d_i)$, being $\mathcal{U}_0$ such as $\forall i$, $j \neq m$, $u_{0,j} = 0$, $u_{0,m} = n$.

Then, number of nodes of $G_{d_i,k}$ is equal to $g'(i)$ in 4-3. The accumulated number of unique $dk$-mers processed until the last iteration is given by 4-4 taking into account the replication factor:

$$\mathcal{G}' = \sum_{i=1}^{z} g'(i) \quad \text{for } d \in D = \{d_1, d_2, d_3, ..., d_i, ..., d_{z-1}, d_z\}, d_z = k \tag{4-4}$$

As mentioned earlier, the number of $dk$-mer processed in each iteration depends on the number and the length of the $dk$-mers generated in the previous iteration, which are the unique elements resulting from the previous iteration. In that sense, $W_i$ will be updating using, $t(W_i, U_{i-1}, d_i)$, instead of $t(W_i, W_{i-1}, d_i)$. In that way, the cumulative number of elements processed until the given iteration $i$, $\mathcal{G}$, comprises duplication in the $dk$-mers.

For traditional approaches, the number of elements processed ($k$-mers) to build the graph is $N = (m - k + 1)n$. There is an expectation that our algorithm could reduce the number of elements to be processed and brings a reduction in computing resources needs.

In terms of $k$-mers, due to the reduction of $dk$-mers by replication factor, there will a number of $k$-mers $P$ that will not need to be processed. For an amount of $dk$-mers in $w_{i,j}$, for specific length $j$ with $\beta_j$ replication factor, we have $(j-k+1)$ $k$-mers inside each $dk$-mer, and the number of $k$-mers that will be avoided being processed is given by equation 4-5. Throughout our work, we will refer to the number of $k$-mers that will be avoided being processed as skipped $k$-mers.

The number of $k$-mers that will be skipped from being processed for a $dk$-mer with length $j$ is:

$$\frac{w_{i,j}(\beta_j - 1)}{\beta_j}(j - k + 1) \tag{4-5}$$

Then, for each iteration, the number of skipped $k$-mers for all $j$ lengths presents, is given by equation 4-6 and the accumulated number of skipped $k$-mers until iteration $z$ is given by equation 4-7

Number of skipped $k$-mers:

$$p(i) = \sum_{j=1}^{m} \frac{w_{i,j}(\beta_j - 1)}{\beta_j}(j - k + 1) \quad \text{in iteration } i \tag{4-6}$$

$$\mathcal{P}(z) = \sum_{i=1}^{z} p(i) \quad \text{accumulated until iteration } z \tag{4-7}$$

## 4.5 Computational requirements for extra-compacted DBG

In order to analyze the computational requirements for extra-compacted DBGs generated in each iteration, first, an analysis of the computational requirements for DBG is made.

### 4.5.1 Computational requirements for DBG construction

The construction of DBG has the objective of identifying the set of vertices and edges of the graph. The set of vertices is the set of $k$-mers from the collection of all $k$-mers. In function to get the set of vertices, it is necessary to identify the set of distinct $k$-mers $V$ from a collection of all $k$-mers, $C_v$, such $|C_v| = (m - k + 1) \times n$, assuming the same $m$ length for all reads and get is multiplicity.

We classified the approaches to construct the graph found in literature in two classes, one based on sorting the vertices, and the other based on a streaming generation and comparison of the vertices.

In the first one, called by us as *sorting approach*, each $k$-mer from $|C_v|$ is sorted and all duplicates are removed. The memory requirements of this approach during the sorting phase is given by $M_{v,sorted}$, which corresponds to the memory needed for each $k$-mer times $|C_v|$ (see equation 4-8).

$$M_{v,sorted} = |C_v| * size(k - mer) \tag{4-8}$$

In terms of the number of $k$-mers processed, being $N$ the number of $k$-mers, and $N = |C_v| = (m - k + 1) \times n$, $M_{v,sorted}$ is given by equation 4-9.

$$M_{v,sorted} = N * size(k - mer) \tag{4-9}$$

The execution time in the main memory is given in function of the number of elements $N$ and the cost of sorting and delete the duplicates. It will be determined by the algorithm and data structure used to sort the vertices and edges.

The second solution, called by us as *streaming approach*, involves the use of specialized data structures to maintain the set $V$. $K$-mers are generated one at a time, it is asked about its previous existence, and if it is not yet present in the graph, it is inserted. A priory, because only the subset of $k$-mers is required to remain in memory ($\varphi$), the memory for streaming approach $M_{v,streaming}$, for a specific data structured (DT), is smaller than the sorting approach. However, the memory overhead ($\omega$) for used data structure must be taken into account such is showed in the equation 4-10.

$$M_{v,streaming} = N\varphi * size(k - mer) + \omega, \quad 0 < \varphi < 1 \tag{4-10}$$

The execution time in main memory is given in function of $N$, and include the cost of generating the elements, and the time of searching and insertion over the data structure. Given the fact that the number of elements to searching is higher than the number of elements that will be inserted, assuming that exists duplication, we only use in the formulation the first one variable (see Equation 4-11).

$$T_{v,streaming} = \mathcal{O}(N(search\_time)) \tag{4-11}$$

For our research, we define the memory needed to get $V$ as $M_V = minimum(M_{v,sorting}, M_{v,streaming})$

The number of distinct $k$-mers ($N\varphi * size(k - mer)$) impacts directly in the memory requirement for main memory processing, while the number of all $k$-mers processed ($N$) impact in the time.

**The identification of the set of edges *E*** is the process that from a $C_e$ collection of all binary relation $u \to v$ between two $k$-mers such as $N = |C_e| = (m - k) \times n \to$ relations, it gets the set $E$ of distinct relations corresponded with $V$. With the aim to get the set of edges, we found in literature three approaches:

(a) From the collection of all edges, get the set of edges.

(b) For consecutive $k$-mers in a read, at time of $k$-mer generation, create an edge, compare with the set of the edges, and insert if it was not previous appear.

(c) For each distinct $k$-mer in $V$, after obtaining $V$, find possible edges searching over the rest $k$-mers in the set.

Edges generation also affects memory consumption and run time. This influence will be tightly defined by the number of edges processed (for example, the number of edges processed in is higher than those of ) and the structured data used during this processing. However, because the number of edges for each $k$-mer is bounded by a maximum of 4 in each sense, we evaluated the execution time for graph construction in terms of the number of $k$-mers, $N$, which is the variable that defines the input size and grows more.

The memory needed to store the graph using adjacency list in streaming approach will be defined as:

$$M_G = N\varphi(size(k-mer) + c * size(edge)) + \omega, \quad 0 < \varphi < 1, \quad c = 4 \tag{4-12}$$

The *size(k-mers)* and *size(edge)* depends on the codification of vertices (including $k$-mer data, multiplicity) and the codification of edges.

Summarizing, computational requirements in the construction of DBG, are influenced by several aspects:

– The number of total $k$-mers $N$, such as $N = (m - k + 1)n$ assuming that all reads has the same length $m$.

– It is not known which is the $k$ value for best assembly.

– The number of unique $k$-mers $N\varphi$, $0 < \varphi < 1$, without knowing in advance.

– High level of redundancy between adjacent $k$-mers.

– Size overhead of data structure used to identify the set of $V$.

– Search time of the data structure used to store $V$.

– $K$-mer codification (vertex codification).

– $K$-mer adjacency codification (edges codification).

## 4.5.2 Memory requirements for extra-compacted DBG

It is required that the available main memory be enough to fit $G_{d_i,k}$ for each iteration. The data structure that contains the set of $V_{d_i,k}$ and $E_{d_i,k}$, needs to be in main memory to minimize the search and insertion time. Using an adjacency list, the amount of memory need to store a $G_{d_i,k}$ for $i$ iteration is defined in 4-13:

$$M_{G_{d_i,k}} = g'(i) * (size(dk - mer) + c * size(edge)) + \omega,$$
$$c = 4 \quad \text{due to the vertex has at most 4 edges} \tag{4-13}$$

The *size(k-mers)* and *size(edge)* depends on the codification of vertices (including *dk*-mer data, multiplicity) and the codification of edges. The data structure used defines $\omega$. Given the fact that not all *dk*-mers have the same length in $i$ iteration, it is assumed that all *dk*-mers have the greater length $d$. Then, emphasizing, assuming a constant number of edges, and exact representation of *dk*-mers, the number of unique *dk*-mers and they length will determine the memory needed in each iteration.

### 4.5.3
### Time complexity analysis

In streaming traditional approaches, each sequence of $n$ is decomposed in $(m - k + 1)$ elements ($k$-mer) that are processed. Each element is searched in the structure. If it is no found, then it is inserted; otherwise, the multiplicity of the element is updated in the data structure. Assuming that structure have a constant asymptotic time for search and insertion, the time complexity is (equation 4-14):

$$\mathcal{O}((m - k + 1)n) \tag{4-14}$$

Our approach proposes a similar strategy, but with a different number of elements ($dk$-mers) for $I$ number of iterations. The asymptotic analysis shows the time for our approach: In first iteration each $n$ read, $m$-length sequences will be decomposed in at most $(f(m, d_1) + 1)$ elements which will be processed (see equation 4-15). Each element will be queried in the data structure. If it exists, then its multiplicity just will be updated, else, it will be inserted as a new element. As in equation 4-14 we assume that it is used a structure with a constant asymptotic time for search and insertion.

In the next iteration, the set of elements resulting from $i = 1$, will be traveled and each element will be decomposed in at most $f(d_1, d_2)+1$ elements. For simplification will be use the upper bounded $d_1$ (in 4-17), since it is the maximum possible length of elements results of iteration $i = 1$. During the third iteration, it will be repeated the process. Each element in the set resulting from $i = 2$, will be decomposed in at most $f(d_2, d_3) + 1$ elements. As before, for simplification, we will use $d_2$ to represent this value. Also, we introduce $n_1$

to represent the number of elements results from the previous iteration, such as $n_1 = nd_1$ elements (see 4-18). Accordingly with this reasoning, for next iteration we used $n_2 = n_1d_2$ each iteration, and its is used $d_3$. Consequently, for the overall process, in each iteration, the elements of the set resulting from the previous iteration will continue to be decomposed, until it is reached the last iteration with $i = z$ and $d = k$. In each iteration, for the worst case, the number of elements processed will be a little greater than the number of elements processed in the previous iteration. Therefore, the overall time of the algorithm is upper bounded by the time of processing the iteration with the highest amount of elements times the number of iterations (see equation 4-20). The number of elements in last iterations is upper bounded by the number of $k$-mers, hence the overall time of the algorithm $T = \mathcal{O}(((m - k + 1) * n)I)$. Because the number of iterations $I$ does not depend on the number of elements, it is possible to say that, in the worst case, our approach has an asymptotic time equivalent to the traditional methods.

In practice, the execution time of our approach could have a variability given the number of iterations, and the exact number of elements processed in each iteration. In real datasets, in the presence of duplicated elements that reduce the number of elements in each iteration, a reduction in execution time is expected. Moreover, it is expected that $P$ has a direct influence on runtime, so a large number of accumulated skipped $k$-mer will decrease it.

During $i = 1$, each read will be decompose using $d_1$

$$\mathcal{O}((f(m,d)+1)*n) \quad for \quad i=1 \tag{4-15}$$

With the sake of simplification

$$\mathcal{O}(Fn) \quad for \quad i=1 \tag{4-16}$$

During $i = 2$, each element resulting from $i = 1$ will be decompose using $d_2$

$$\mathcal{O}(Fn + Fnd_1) \quad for \quad i=2 \tag{4-17}$$

During $i = 3$, each element resulting from $i = 2$ will be decompose using $d_3$

$$\mathcal{O}(Fn + Fnd_1 + Fn_1d_2) \quad for \quad i=3 \tag{4-18}$$

Consequently, for next iteration:

$$\mathcal{O}(Fn + Fnd_1 + Fn_1d_2 + Fn_2d_3) \quad for \quad i=3 \tag{4-19}$$

Consequently for the all process:

$$\mathcal{O}(Fn + Fnd_1 + Fn_1d_2 + Fn_2d_3 + ... + Fn_{z-1}k) \quad for \quad i=z, d=k \tag{4-20}$$

Since the last iteration will have the greater number of elements to process:

$$\mathcal{O}((Fn_{z-1}k)*I) \quad \text{where } I \quad \text{is the number of iterations} \tag{4-21}$$

Last iteration is bounded by the number of $k$-mers, hence

$$\mathcal{O}((m-k+1)*nI) \quad for \quad k \leq I < m \tag{4-22}$$

(4-23)

The analysis of memory and execution time for our approach based on the number of elements are independent of the data structure used, i.e., we analyze the problem making an abstraction of that data structure used, assuming the same data structure and elements codification if we compare our approach with others.

## 4.6 Processing pipeline. Profits of our approach

As was stated in section 4.5.2, the number of unique $dk$-mers in each iteration determine the memory needed for the iteration, $M_{G_{d_i,k}}$. To describe the different execution flows of our approach and its profits is introduced the pipeline definition in the following.

**Definition 4.5** *Iteration pipeline is a sequence of executions using the same update function and step parameter to get* $d_{i+1}$

In case that main memory available $M$ is insufficient to DBG construction, i.e. $M < M_G$ (Case A) , then $M < M_{G_{k,k}}$. Even, in previous iterations $M$ could be less than $M_{G_{d_i,k}}$. In both cases, the algorithm will have reached a $p_1 = p(i)$ $k$-mers that will not have to be processed. At that point it possible:

1. Stop the current pipeline and use the output as input for processing in the external memory model. In this case, being $p_1 = p(i)$, $p_1$ $k$-mers where suppressed, which will be avoided being processed, reducing the number of I/O operations.

2. Stop the current pipeline, and start another pipeline using a different *update* function that makes $g'(i)$ less, such as $M > M_{G_{d_i,k}}$.

   – Start from the output of the last successful iteration, i.e. $i-1$, and uses a new value for $d_i$, such as $M > M_{G_{d_i,k}}$. Therefore, will be possible to obtain $p_2 > p_1$. It is possible to extend the pipeline for selected *update* function, or even change the function some times in a tuning process, which the aim to reduce as much possible $p(i)$.
   – Start from the input sequences, since $i = 1$ with other $d_1$ or/and *update* function parameters with the aim to get $p_3$ such as $p_3 > p_1$.

3. In cases where an external memory solution is undesirable, our approach will also have benefits when using strategies with a non-exact representation of DBG, reducing the number of items that were processed and globally processing a more significant number of items in an exact approach, without the presence of false positives, for example.

Even for executions that $M > M_G$, (Case B) , our approach could have a profit. As the amount of processed items is expected to be less than the total amount of $k$-mers, the execution time will be reduced.

## 4.7 External memory processing at last step

Our approach promotes the idea to process the graph as much as possible in the main memory, reducing the number of duplicated $k$-mers in each iteration. Only when the available main memory becomes insufficiently to store the structure of $G_{d_i,k}$, the use of an external memory solution is suggested. At that time, large duplicate regions have already been identified. This will allow avoiding to process a significant amount of duplicated $k$-mers in external memory, and consequently, reducing the number of I/O operations. Moreover, before using external memory processing, it is possible to apply an intermediate tuning solution to reduce even more the amount of data to processes in external memory (section 4.6).

Using our approach is possible to build a macro representation of DBG ( i.e., the extra-compacted DBG) in main memory and, if only if necessary, use a solution in external memory in the last iterations. In that sense, our vision is to be able to take better advantage of the available RAM, reaching a higher percentage of processing before going to processing using external memory.

For an iteration $i$, given $i < z$, in which $M$ is not sufficient to storing $G_{d_i,k}$, it is possible to export $V_{d_{i-1},k}$ and $E_{d_{i-1},k}$, to be used as input of external solutions. The $dk$-mers into $V_{d_{i-1},k}$ could be exported so that they can be seen as a set of reads $R$ with multiplicities for other solutions. The set of edges $E_{d_{i-1},k}$ is a subset of final $E$. Then, the set $E$ of external solution could be initialized making $E = E_{d_{i-1},k}$.

The **external memory** model [Aggarwal and Vitter 1988] is also called the "I/O Model" or the "Disk Access Model" (DAM). An external memory model is commonly applied in algorithms developed to manage a massive amount of data. It simplifies the memory hierarchy to just two levels. The CPU is connected to a fast cache of size $M$; this cache, in turn, is connected to a much slower disk of effectively infinite size. Both cache and disk are divided into blocks of size $B$, so there are $M$ blocks in the cache. Transferring one block from cache to disk $B$ (or vice versa) costs 1 unit. Memory operations on blocks resident in the cache are free. Thus, the fundamental goal is to minimize the number of transfers between cache and disk. [Massachusetts Institute of Technology 2012].

In that scene, using the case of sorting approach in [Kundeti et al. 2010], the collection of all $k$-mers and edges will be sorted to identify the set of $V$ and $E$ correspondingly using an optimal number of I/O in the equation 4-24.

$$\Theta(\frac{N\log{(N/B)}}{B\log{(M/B)}}) \quad \text{I/O operations to get } V\text{, given } N \text{ } k\text{-mers} \tag{4-24}$$

Using our approach, if there exist an iteration $i$, given $i < z$, in which $M$ is not sufficient to store $G_{d_i,k}$, it is possible to build the DBG using this sorting approach decreasing the I/O as shown in equation 4-25. In that sense, $N$ is reduced by the number of $k$-mers that will be avoided to processed $P(i)$.

$$\Theta(\frac{(N-P(i))\log{((N-P(i))/B)}}{B\log{(M/B)}}) \quad \text{I/O operations to get } V \tag{4-25}$$

Since the number of edges to process was reduced as the number of $k$-mers during previous iterations, the same analysis for I/O could be applied to the set of edges.

Now, we turn to the case of using a partition processing approach in external memory. In that case, we analyzed the I/O in its three main steps: distribution, processing, and merging.

In the first step, the collection of all $k$-mers and edges implicit will be distributed in $n$ partitions. The number of elements and the criteria used to distribution will determine the size and the number of partitions. The distribution of the collections is hard to know until the factual data has been distributed. This fact can cause a re-partition in case the amount of data is more extensive than what is supported to be processed in the main memory. The number of I/O operations in this step is shown in equation 4-26.

$$\mathcal{IO}_1 = \sum_{j=1}^{j=n} \frac{size\_of\_part_j}{B} \quad \text{I/O operations in distribution phase} \tag{4-26}$$

In the second step, each partition is read from the disk and processed. Then the results are writing in the disk in a compiled partition. Compiled partitions are less than initial partitions. The I/O of processing step is shown in equation 4-27. It is used $\gamma$ to represents the I/O operations consumed during the write of compiled partitions.

$$\mathcal{IO}_2 = (1+\gamma)\sum_{j=1}^{j=n}\frac{size\_of\_part_j}{B} \quad 0<\gamma<1 \quad \text{I/O operations in processing step} \tag{4-27}$$

Finally, the merging step reunites all compiled partitions to generate a result, as is shown in equation 4-28.

$$\mathcal{IO}_3 = \gamma\sum_{j=1}^{j=n}\frac{size\_of\_part_j}{B} \quad 0<\gamma<1 \quad \text{I/O operations in merging step} \tag{4-28}$$

The overall partition strategy I/O is given by equation 4-29

$$\mathcal{IO} = 2(1+\gamma)\sum_{j=1}^{j=n}(\frac{size\_of\_part_j}{B}) \quad 0<\gamma<1 \quad \text{I/O operations in partition strategy} \tag{4-29}$$

Therefore, during the execution of our approach, if there exist an iteration $i$, given $i < z$, in which $M$ is not sufficient to store $G_{d_i,k}$, it is possible to build the DBG using this distribution approach decreasing the I/O as shown in equation 4-30. The number of $k$-mers impacts in the number and size of partitions, given a reduction of initial $k$-mers by $P(i)$. To represent this reduction, we used the variable $\mu$, with $\mu > 1$.

$$\mathcal{IO} = 2(1+\gamma)\sum_{j=1}^{j=n'}(\frac{size\_of\_part_j}{\mu_j B}) \quad 0<\gamma<1 \quad \text{I/O operations in partition strategy} \tag{4-30}$$

Finally, so that external memory implementations for the construction of DBG can use our approach, we propose that the former implements an input interface, such as:

- The sequence of $dk$-mers in $V_{d_{i-1},k}$ will treated as reads.
- Initialize the multiplicity for each unique $k$-mer with the multiplicity of the $dk$-mer.
- Initialize the set of edges with $E_{d_{i-1},k}$.

## 4.8 Partial conclusions

In this chapter was addressed our novel approach to the construction of *de Bruijn* graph.

The motivation, steps, and gains of our approach were detailed in the course of the chapter. Runtime and memory requirements were also deeply discussed.

As was explained, our approach is viable to build the graph, in a macro way, with available memory. Only in the last steps, if necessary, it is proposed the use of an external memory solution to finish generating DBG, bringing a decrease in the number of I/O operations. The detailed pipeline also shows the profit possibilities of our approach, including tuning the parameters of $d$ and the *update* function.

# 5
# Implementation and results

We investigate in this chapter the way our theoretical ideas and contributions may behave in practice. Initially, we explain the data structures used for DBG codification in our experiments. Mainly, we discuss in detail our choices for the extra-compacted DBG with a sparse hash implementation. We conclude with the practical results obtained when applying our strategy to three different species datasets: bumblebee, human chromosome 14, and sugar cane. Our analysis includes not only time execution performance and a comparison with popular assemblers but also memory usage and the impact of the proposed modifications concerning efficacy and efficiency.

## 5.1
## DBG implementation

One of the most critical decisions during the implementation was to select the data structure for the extra-compacted DBG. Our approach focuses on the exact representation of $k$-mers and DBG. Therefore, the extra-compacted DBG in each iteration requires a precise representation of $dk$-mers. This condition impacts significantly in the data structure selection to store $G_{d,k}$, and the codification used for vertices and edges of the graph.

### 5.1.1
### Vertices and edges codification

For vertex codification we propose the use of classical 2 bits representation for base in the smallest available integer type for $dk$-mers as is used for $k$-mer in [Rizk et al. 2013], [Zerbino and Birney 2008] [Ye et al. 2011].

Each $dk$-mer will need $2^{\lceil log_2(2d) \rceil}$ bytes.

We may define the amount of memory needed at each iteration to store each element independently as:

$$g'(i) \times 2^{\lceil log_2(2d) \rceil} \tag{5-1}$$

Moreover, we also used a compacted representation of $dk$-mers as a strategy to minimize the amount of memory for the vertices. We implemented this strategy by dividing the $dk$-mers into two parts, and saving only once the $k-1$ duplicated prefixes of $dk$-mers.

The multiplicity of vertices (see 2.4) will be obtaining by accumulating it during decomposition. Given a vertex $v_{i-1,1}$ in iteration $i-1$, given a multiplicity $x$ implies that for all $k$-mer in that vertex have $x$ repetitions. In iteration $i$, for each $dk$-mer decomposed from $v_{i-1,1}$, if not found in the set of current vertices $V_{d_i,k}$, it is inserted with multiplicity $x$. Otherwise, the multiplicity in $V_{d_i,k}$ for this $dk$-mer will increase by $x$.

Finally, our approach also focuses on the exact representation of edges. The above means that the edges correspond exactly to the sequences of bp in the reads, and it is used an exact data codification. To represent the edges we propose the use of the optimized variant used in [Simpson et al. 2009] and [Ye et al. 2011]. To store the adjacency information between $k$-mers, a four bits extension for each $k$-mers is used to map every possible one-base extension, $'A','C','G','T'$, in a single direction.

## 5.1.2 Extra-compacted DBG representation

For each iteration, the executed operations include searches and insertions, no deletions occur. It is lookup-intensive processing since each *dk-mer* must be searched in the set of vertices to update its multiplicity, and inserted only if not previously exists.

Hash tables are widely used as a solution for DBG construction (*e.g.* [Zerbino and Birney 2008] [Luo et al. 2012] [Li et al. 2013] [Zerbino and Birney 2008]). Hash tables has a theoretical $\mathcal{O}(1)$ search time. It's primary advantage is that lookup, insertion and deletion operations run in constant time, on average.

To store $G_{d_i,k}$ we decided to use the google sparse hash implementation from (`https://github.com/sparsehash/sparsehash`), which offers substantial trade-offs between memory and speed. This implementation shows the low memory usage in benchmarks [Welch 2009] and [Neustar, Inc. 2016], roughly half the memory of Boost implementation (`https://www.boost.org/doc/libs/1_62_0/doc/html/unordered.html`), the next most memory-conservative implementation. Concerning the execution time, it is in the average of the implementations, which still basically means very fast [Welch 2009].

The Google sparse hash map uses a sparse array to minimize the overhead of empty buckets [Penman's 2017]. Dynamic arrays only store addressed entries, no matter how ample the array address space may be. In this way, the structure will need memory only for the entries that are used, plus some bookkeeping overhead. Free buckets have 2 bits of overhead when using 32-bit pointers, 2.7bits for 64-bit pointers. Despite using a sparse array, Google's implementation preserves the constant time complexity of lookup, insertion, and deletion operations through intelligent bitmap manipulation techniques. However, it is important to note that these operations may be slower by a constant factor. It implements open-addressing schemes to resolve the collision, using a quadratic probing to find available buckets.

Our initial design used a map such that entries are formed by:

– key: sequence of a $dk$-mer.
– data: multiplicity and edges of this $dk$-mer.

Since a *dk-mer* can be too long for initial iterations, we have designed a nested data structure schema (see in Fig. 5.1) for compacting $dk$-mers, also facilitating a fast graph traversal for the next assembling steps. At the first level, we implemented a main sparse hash map, which maps entries as follows:

– key: sequence of $k-1$-length prefix of $dk$-mer.
– data: pointer to another a nested sparse hash map.

At the second level, we implemented nested sparse hash maps, whit the following entries:

– key: sequence that corresponds to the remaining part of a $dk$-mer.
– data: multiplicity and edges of this $dk$-mer.

With this structure, the $k-1$ prefixes shared by two or more different $dk$-mers, are stored only once in the main map, and the remaining part of those *dk-mers* are saved independently in the nested maps. As a result, the length of the independent elements and the general memory are reduced. For the remaining part of those $dk$-mers, the same strategy could be applied, nesting another level to save even more memory. However, our implementation has only two levels.

We will call the above-nested structure a hash map table ($HT$).

One $HT$ is necessary to represent one $G_{d_i,k}$. Therefore, for $i > 1$, while vertices in $G_{d_{i-1},k}$ are decomposed and inserted into $G_{d_i,k}$, it would be necessary to have a $HT$ for each graph.

For $i = 2$, assuming $HT_1$ contains the input of the current iteration, and $HT_2$ the output, the length of a $d_1k$-mer from $HT_1$ could be:

– Greater than $d_2$: In this case, it is decomposed in a sequence of $d_2k$-mers. The first $d_2k$-mer in the sequence will have the same $k-1$ prefix as $d_1k$-mer. Therefore, it will have the same entry in the main table as $d_1k$-mer.

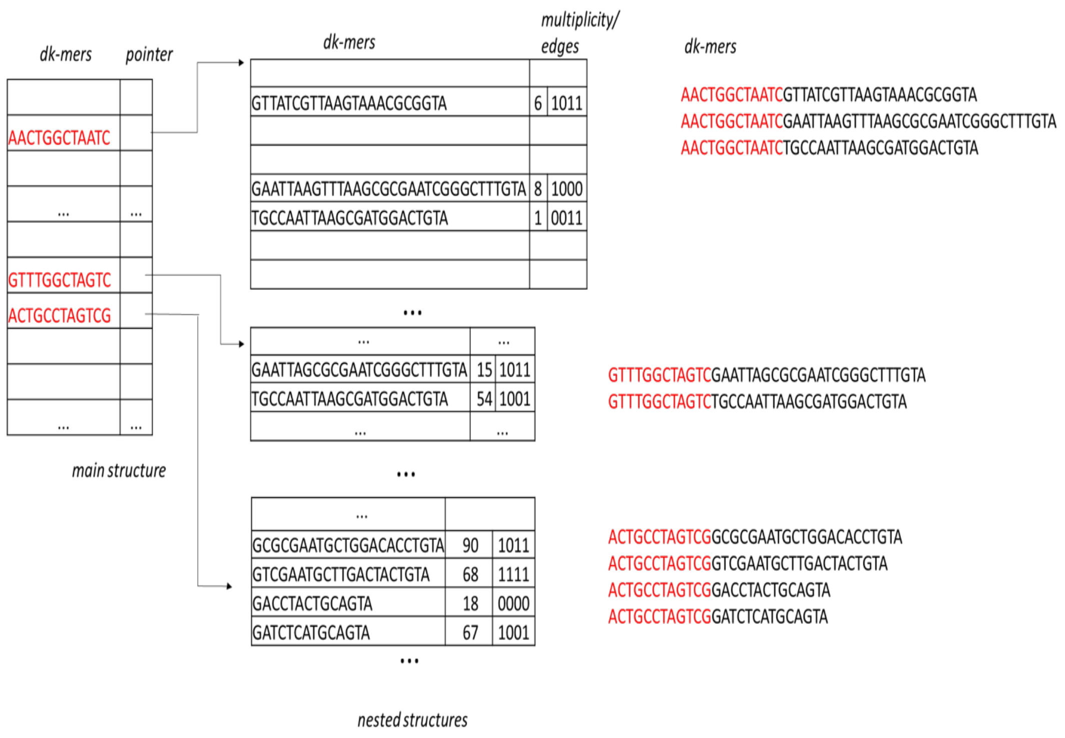

dk-mers
pointer
AACTGGCTAATC
...
GTTTGGCTAGTC
ACTGCCTAGTCG
main structure
dk-mers
multiplicity/ edges
GTTATCGTTAAGTAAACGCGGTA 6 1011
GAATTAAGTTTAAGCGCGAATCGGGCTTTGTA 8 1000
TGCCAATTAAGCGATGGACTGTA 1 0011
GAATTAGCGCGAATCGGGCTTTGTA 15 1011
TGCCAATTAAGCGATGGACTGTA 54 1001
GCGCGAATGCTGGACACCTGTA 90 1011
GTCGAATGCTTGACTACTGTA 68 1111
GACCTACTGCAGTA 18 0000
GATCTCATGCAGTA 67 1001
nested structures
dk-mers
AACTGGCTAATCGTTATCGTTAAGTAAACGCGGTA
AACTGGCTAATCGAATTAAGTTTAAGCGCGAATCGGGCTTTGTA
AACTGGCTAATCTGCCAATTAAGCGATGGACTGTA
GTTTGGCTAGTCGAATTAGCGCGAATCGGGCTTTGTA
GTTTGGCTAGTCTGCCAATTAAGCGATGGACTGTA
ACTGCCTAGTCGGCGCGAATGCTGGACACCTGTA
ACTGCCTAGTCGGTCGAATGCTTGACTACTGTA
ACTGCCTAGTCGGACCTACTGCAGTA
ACTGCCTAGTCGGATCTCATGCAGTA

Figure 5.1: Nested data structure used for extra-compacted DBG. The main structure contains a prefix of $dk$-mers and a pointer to a nested structure. Nested structure, contains the remaining of $dk$-mer sequence, along with the multiplicity of the vertex and a edge map bit codification

Thus, its corresponding entry in the main map will be repeated in both data structures.

– Smaller than $d_2$: It will not be decomposed; rather, it will be copied to $HT_2$. The main map entry and the nested map entry will be repeated in both data structures.

The maintenance of two $HT$ in an iteration brings a waste of memory as a consequence of repeated data. Also, intermediate deletions in $HT_1$ are inefficient, since main entries must exist until all nested elements are processed. Furthermore, deletions are expensive in the middle of the evaluation and can induce rehashes.

Consequently, we proposed and implemented a strategy to use only one $HT$ to store both $G_{d_{i-1},k}$ and $G_{d_i,k}$. At the beginning of the iteration $i$, the $HT$ only contains $G_{d_{i-1},k}$. It will be traverse to the last element, and each $dk$-mer visited and processed will be marked. New elements will be inserted in $G_{d_i,k}$ as they appear, inserting new main entries as necessary. It is important to note that elements in $G_{d_i,k}$ will have length less or equal to $d_i$. The $dk$-mers from $G_{d_{i-1},k}$ with a length less than $d_i$, will not be copied, they will remain in the same state, conceptually it is as if they had been inserted in $G_{d_i,k}$. New recently inserted elements can be visited. However, since their length is less than $d_i$, they will be treated as elements of $G_{d_{i-1},k}$ with length less than $d_i$, without taking any action. At the end of the iteration, all $dk$-mers with a length greater than $d_i$ and marked will be cleaned up.

## 5.2 Experimental results

We implemented a test prototype using the C++ programming language developed using IDE Netbeans 8.1. We have run our executions in a virtual machine hosted in private cloud infrastructure, using one virtual machine with Ubuntu 18.04, one CPU core Intel Xeon E312xx 2.2GHz, with 33GB of RAM and 500GB of HD.

The datasets used in our experiments include three groups of organisms:

***Sugar cane libraries.*** Fragment libraries collected from Brazilian sugarcane species kept by UFRJ's Institute of Medical Biochemical (IBqM):

– R03 with $n = 8,520,922$ and $m = 72bp$

– R06 with $n = 5,298,464$ and $m = 72bp$

– R10 with $n = 5,723,392$ and $m = 76bp$

where $n$ represent the number of reads, and $m$ is the read length.

***Human Chromosome 14***. Fragment library of Human Chromosome 14 (Ch14) available in `http://gage.cbcb.umd.edu/data/`:

– H1: Library 1 with $n = 18,166,705$ and $m = 101bp$ in average.

– H2: Library 2 with $n = 18,166,798$ and $m = 101bp$ in average.

***Bombus impatiens (bumblebee)***. Fragment library of Bombus impatiens available in `http://gage.cbcb.umd.edu/data/`:

– B2: Library 2 with $n = 120,000,000$ and $m = 124bp$ in average.

Each one of these three configurations was executed in ABYSS to construct the DBG. We have noticed that, for each execution, the number of vertices and edges of DBG produced by ABYSS were equivalent to the DBG output that our approach generates.

Table 5.1: shows our set of planned experiments that helps with the comprehension of our actual contributions.

Results for each execution are available in the next sections and in the Section B of the Appendix.

Table 5.1: Experiments description.

| No. | Datasets | $k$ | $d_1$ | *step* | Goal |
|---|---|---|---|---|---|
| 1 | R03, R06, R10, H1, H2 | 12 | 64 | 10 | Test the approach, and proof that it is viable. Measure unique $dk$-mers, memory and runtime. |
| 2 | R10 | 15 | 52, 55, 58, 61, 64 | 10 | Shows how the $d_1$ impact in the number of $k$-mers skipped from being processed, the accumulated number of processed elements and unique $dk$-mers. |
| 3 | H1 | 11 | 64 | 5, 8, 10, 15 | Shows how each *step* impact in the number of $k$-mers skipped from being processed, the accumulated number of elements processed and unique $dk$-mers. |
| 4 | R03, R06, R10, H1, H2 | 12 | 12 | | Measure runtime of traditional approaches in our implementation. |
| 5 | R03, R06, R10, H1, H2 | 12, 13 | | | Comparing our approach with the requerimentes for DBG construction of other assemblers like ABYSS and Velvet. |
| 6 | Bee | 31 | 100, 55, 35, 31 | 10 | Proving our approach in case that DBG does not fit in main memory. |

## 5.2.1 Number of skipped $k$-mers at each iteration

Figure 5.2 shows the number of $k$-mers processed using our approach over the total number of $k$-mers. These tests reveal that our approach reduces the need for processing a significant amount of $k$-mers. In Table 5.2 we show the percentage of skipped $k$-mers. For all datasets, the percent of $k$-mers that did not have to be processed was over 70%. The accumulate value $P(i)$ means the number of elements that are skipped from being processed. It also shows the remaining elements to process if at that point, the execution needs to be processed in external memory, or even by another approach (see Section 4.6). The dataset R03 obtains the highest percentage, although the highest amount of skipped $k$-mers was given for the Human Ch14 libraries. We may explain this behavior as the number of reads in human libraries is, at least, two times the number of reads when compared to the other datasets.

**Analysis of $d$. How this affects the number of skipped $k$-mers.**

To analyze the impact of $d$ value in the number of saved $k$-mers, five

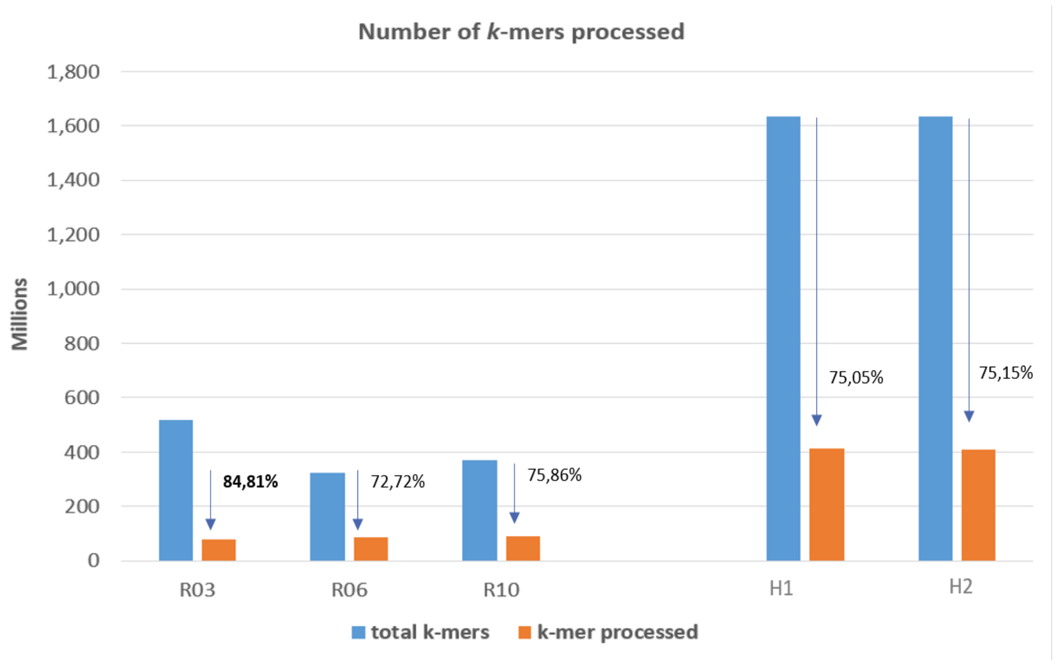


Figure 5.2: Number of $k$-mers processed using our approach compared to overall number of $k$-mers

Table 5.2: Saved number of $k$-mers in Experiment 1.

| Datasets | $N$: total number of $k$-mers | $P(z)$: skipped $k$-mers | $k$-mers processed | % of skipped $k$-mers |
|---|---|---|---|---|
| R03 | 519.776.242 | 440.839.371 | 78.936.871 | 84,81 |
| R06 | 323.206.304 | 235.029.980 | 88.176.324 | 72,72 |
| R10 | 372.020.480 | 282.215.035 | 89.805.445 | 75,86 |
| H1 | 1.635.003.450 | 1.227.017.635 | 407,985,815 | 75,05 |
| H2 | 1.635.011.820 | 1.228.669.547 | 406,342,273 | 75,15 |

configurations over R10 were applied and executed, updating $d$ at each iteration through $update(d) = d_{i-1} - 10$ (Function 4.4.1.1). Depending on the initial $d_1$, the execution may have more or less number of iterations. As we can see in Fig. 5.3, all iterations have the same trend over the cumulative percentage of skipped $k$-mers for different $d_i$ values. The execution that has the greater number of skipped $k$-mers was the one whose last iteration had a $d$ that eventually came closer to $k$. The average of the replication factor for execution, meanwhile, showed almost constant behavior overall executions, varying from 1.23 to 1.57, with an average of 1.31.

The observed variation in the number of skipped $k$-mers demonstrated that it is possible to fine-tune $d_1$ to obtain betters results. Unfortunately, with these results, it was not possible to define a tendency between the number of accumulated processed elements and the maximum number of unique $dk$-mers for all iterations in correspondence with the $d_1$ value.

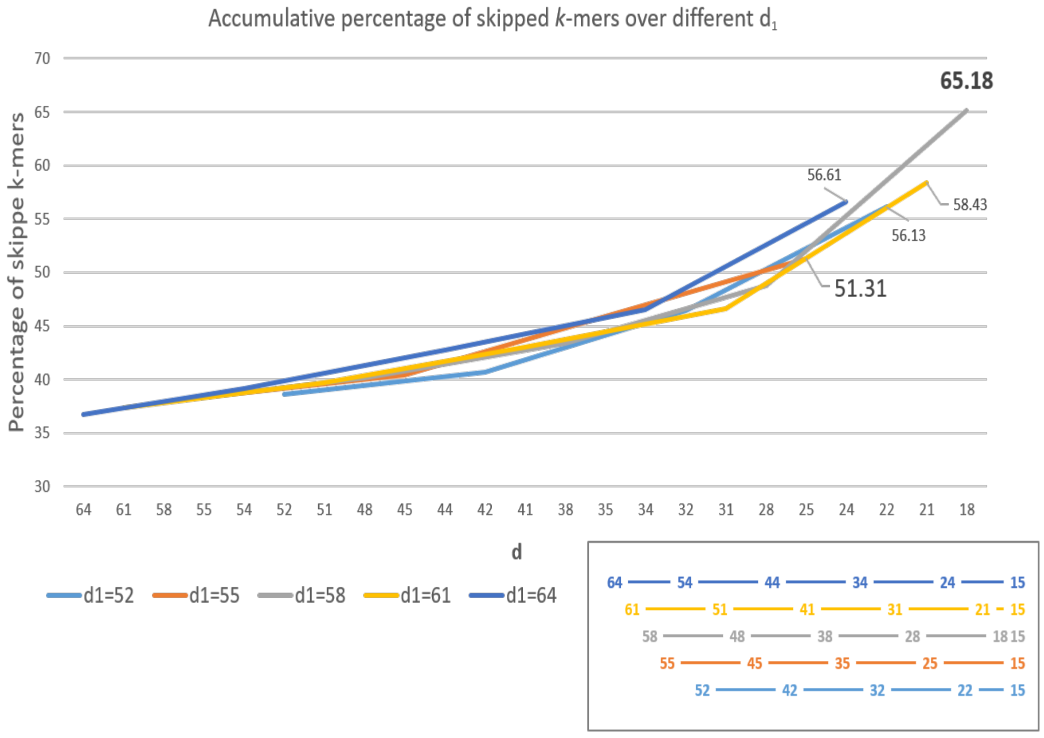


Figure 5.3: Comparison of cumulative percentage of skipped $k$-mers over different executions starting with different $d_1$

Table 5.3: Summary of executions for Experiment 2 to analyze how impact $d_1$. R10 with $k$=15, varying $d_1$.

| $d_1$ | $\mathcal{G}$: accumulated number of elements processed | $g'(i)_{max}$: maximum of unique $dk$-mers | $P(i)$: accumulated skipped $k$-mers | $P(i)/N$ (%) |
|---|---|---|---|---|
| 52 | 228,206,307 | 27,574,544 | 199,178,456 | 56.13 |
| 55 | 226,958,946 | 16,663,215 | 182,085,221 | 51.31 |
| 58 | 231,130,061 | 36,752,813 | 231,296,700 | 65.18 |
| 61 | 235,668,807 | 29,291,765 | 207,335,327 | 58.43 |
| 64 | 227,049,233 | 17,768,889 | 200,898,166 | 56.61 |

**Analysis of *update* function.**

We are also interested in studying the way the the function that updates $d$ can affect the number of skipped $k$-mers. With this goal in mind, four configurations were executed over H1 dataset, using the function in 4.4.1.1 with a constant $k = 11$ and varying the *step* values. Detailed results are presented in B.7.

As we can see in Figure 5.4, all iterations have the same tendency over the cumulative percentage of skipped $k$-mers for different $d_i$ values. However, the value for the greater *step* $= 15$ had a representative decrease compared to the rest of values. Table 5.4 shows that not for more exhaustive searches, it is possible to obtain better results for accumulative skipped $k$-mers.

As in the previous subsection, the executions that had the highest number of skipped $k$-mers were those that the last iteration got $d$ closer to $k$. In that case, it happened for *step*=10 and *step*=5. For the latter, a more exhaustive one, it achieved a slightly higher percentage.

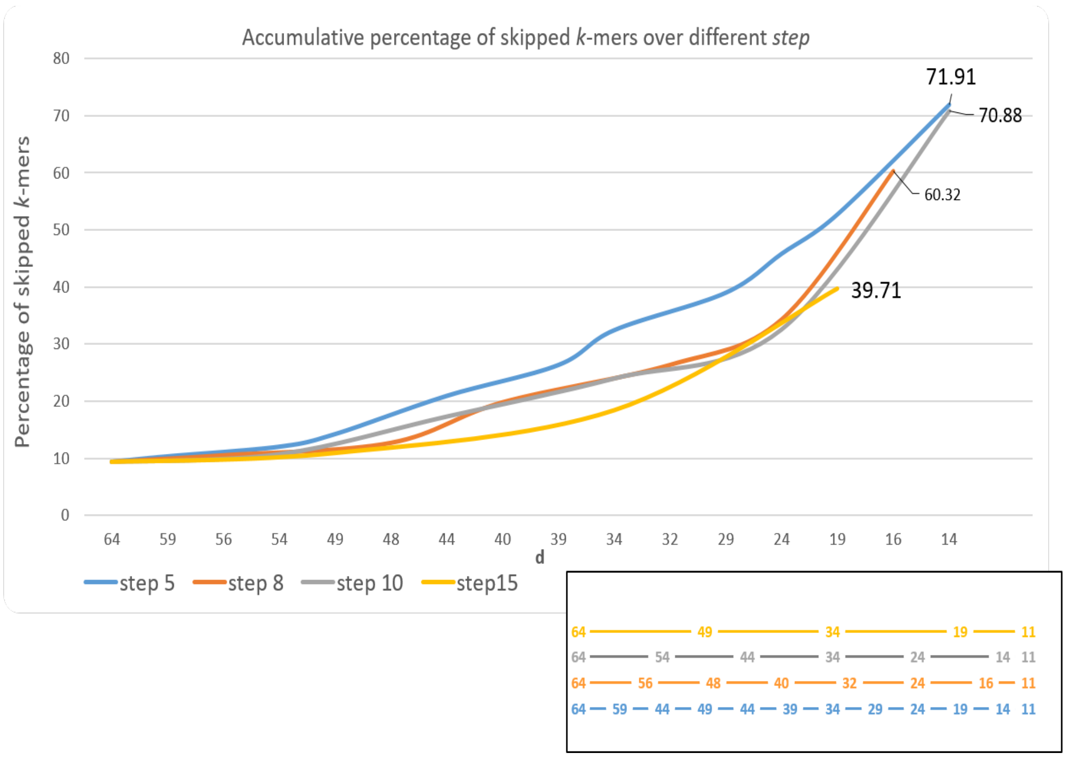


Figure 5.4: Comparison of cumulative percentage of skipped $k$-mers over different executions with different *step*

Again, with these results, it was not possible to define a trend between the number of accumulated processed elements and the maximum number of unique $dk$-mers for all iterations in correspondence with the *step* value. As in previous analysis, the variation showed in the number of skipped $k$-mers, demonstrated that it is possible to fine-tune the function or its parameters, like the *step*. We could obtain betters results and a reduction of I/Os operations, in case it is needed.

Table 5.4: Summary of executions for Experiment 3 to analyze how impact the *step*. H1 with $k$=11, varying *step* parameter for *update* function.

| *step* | $\mathcal{G}$: accumulated number of elements processed | $g'(i)_{max}$: maximum of unique $dk$-mers | $P(i)$: accumulated skipped $k$-mers | $P(i)/N$ (%) |
|---|---|---|---|---|
| 15 | 1,069,719,344 | 105,203,153 | 656,412,143 | 39.71 |
| 10 | 851,702,944 | 86,794,358 | 1,171,691,777 | 70.88 |
| 8 | 1,053,403,745 | 92,690,341 | 997,267,892 | 60.32 |
| 5 | 1,276,742,648 | 97,421,806 | 1,188,872,847 | 71.91 |

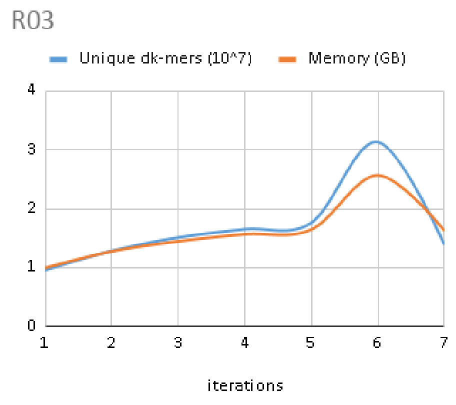
## 5.2.2 Memory analysis

With the aim to study the memory used in the process, we have to analyze the behavior of unique $dk$-mers. For each iterations, we can see how the memory requirements fits to the number of unique $dk$-mers (see Table 5.5), reaching its highest value for iterations which greater number of unique $dk$-mers.

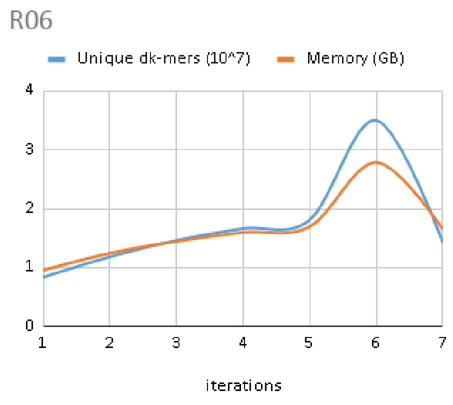
Table 5.5: Datasets R03, R06, R10, H1 and H2. Memory used in each execution. Experiment 1. $k = 12$, $d_1 = 64$, $d$ is updated decremented by 10 in each iteration. $g'(i)$ is the number of unique dk-mers and $Mem.$ is the memory used in GB.

| i | R03 | | R06 | | R10 | | H1 | | H2 | |
|---|---|---|---|---|---|---|---|---|---|---|
| | $g'(i)$ | *Mem* | $g'(i)$ | *Mem* | $g'(i)$ | *Mem* | $g'(i)$ | *Mem* | $g'(i)$ | *Mem* |
| 1 | 9,604,846 | 1.00 | 8,385,453 | 0.96 | 6,807,003 | 0.91 | 29,919,316 | 2.33 | 29,939,425 | 2.34 |
| 2 | 12,840,309 | 1.28 | 11,842,147 | 1.25 | 9,784,098 | 1.15 | 42,743,049 | 3.09 | 42,778,322 | 3.09 |
| 3 | 15,113,476 | 1.45 | 14,626,588 | 1.45 | 12,292,626 | 1.34 | 61,420,685 | 4.19 | 61,469,230 | 4.19 |
| 4 | 16,517,971 | 1.57 | 16,634,211 | 1.60 | 14,595,400 | 1.50 | 77,411,456 | 5.12 | 77,471,870 | 5.12 |
| 5 | 17,616,768 | 1.65 | 18,114,159 | 1.70 | 16,810,095 | 1.65 | 89,340,932 | 5.81 | 89,413,646 | 5.81 |
| 6 | 31,321,779 | 2.57 | 35,010,061 | 2.79 | 34,635,823 | 2.78 | 69,397,710 | 4.73 | 69,430,201 | 4.73 |
| 7 | 13,881,227 | 1.63 | 14,231,519 | 1.65 | 14,654,968 | 1.68 | 13,736,339 | 1.60 | 13,741,195 | 1.60 |

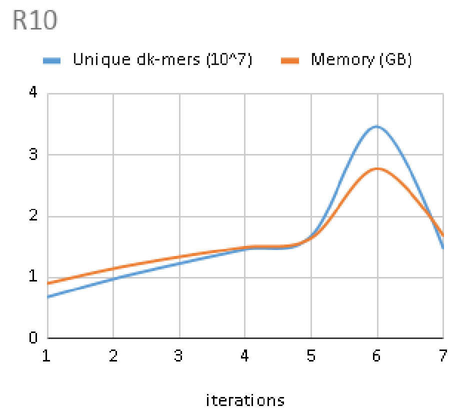
In case of datasets R03, R06, R10, they reach their maximum number of unique $dk$-mers for $i = 6$ with more than 30 millions of unique $dk$-mers, demanding 2.57GB, 2.79GB and 2,78GB of RAM to store those elements. In the same way, the datasets corresponding to Human Ch14 have similar behavior with respect to the memory requirements and reach their peak for $i = 5$, using 5.81GB for around 89 million of $dk$-mers. The graphic for the number of unique $dk$-mers versus the memory used (Fig. 5.5) shows that the curves for the number of unique $dk$-mers and memory have similar behavior for each dataset.



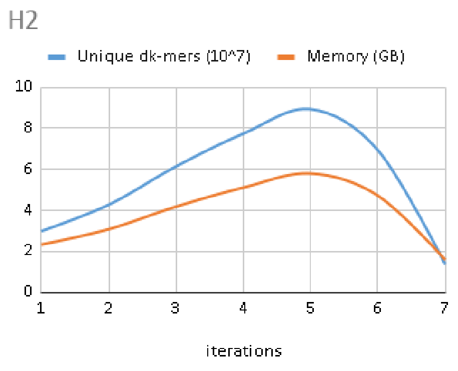
5.5(a): Memory and unique *dk*-mers for R03



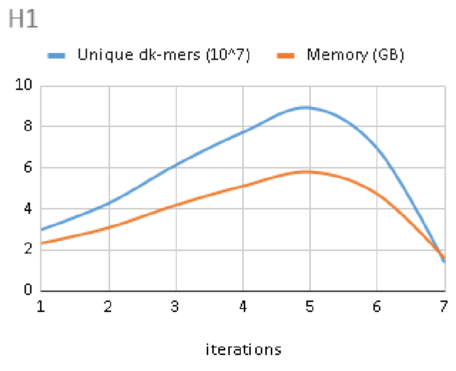
5.5(b): Memory and unique *dk*-mers for R06



5.5(c): Memory and unique *dk*-mers for R10



5.5(d): Memory and unique *dk*-mers for H1



5.5(e): Memory and unique *dk*-mers for H2

Figure 5.5: Memory compared with the number of unique *dk*-mers.

Given a hypothetical $X$ software, which uses the traditional approach to build DBG, with exact codification for unique $k$-mers, then, if the available memory $M$ is not enough for $X$ to build the DBG (Case A in 4.6), for $k = 12$, our approach could process up to at least iteration 2 for R03 and R06, and until iteration 4 for R10, skipping to being processed the 35.59% and 14.60 %

of $k$-mers for R03 and R06 correspondingly, and 42.77% of $k$-mers for R10.

In fact, our implementation is even more performing. The results show the behavior of our data structure optimized for long $dk$-mers, to take advantage of $k-1$ prefix duplication of unique $dk$-mers. While $d$ is going to approximate to $k$, it is more likely that a minor number of $dk$-mers shares the same $k-1$ prefix. When $d$ reaches the $k$ value, at most, four elements reside in each nested table, weighting more the overhead of the table. These are the reason that for the last iteration, the memory used is higher than other iterations with a greater number of unique $dk$-mers. For example, for R03 in the last iteration, the number of unique $dk$-mers is around 13 million for 1.63GB of memory. However, until the iteration $i = 4$, while the number of unique $dk$-mers grows from 9 million to more than 16 million, the memory requirements do not over 1.63GB. The same behavior is present in R06 and R10.

In the case of Human Ch14 libraries, with the parameters used in the experiments, no iteration had the number of unique $dk$-mers less than the number of $k$mers for the DBG built. Therefore, in the presence of $X$, being $M$ insufficient to $X$ build the DBG (Case A in 2), our approach could not be executed with experimented configuration. However, another configuration (Case A in 2) could be tested to see if the process can be carried out until some point, at least to skipping from being processed a set of $k$-mers.

### 5.2.3 Time analysis

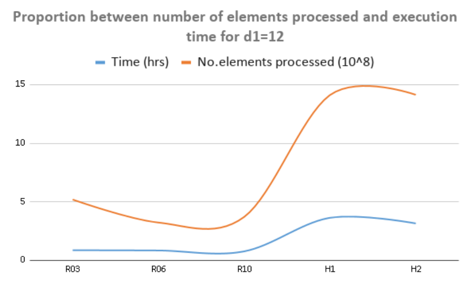
In order to analyze the execution time, we run Experiment 4 (Table 5.1) using our prototype, with $k$=12 and $d_1 = 12$ to simulate the execution of the traditional approach, since each $dk$-mer will be a $k$-mer with overlaying $k-1$. Following, we compare the runtime results between Experiment 1 and 4, to analyze how the number of elements processes affect the execution time. The values (see Table 5.6) evidence that our approach builds the DBG with a minor time, while the number of elements processed decreases.

Also, the curves in Fig. 5.6 shows the proportion between number of elements processed and the execution time for Experiment 1 (Fig. 5.6(b)) and Experiment 4 (Fig. 5.6(a)). Both graphs show that the two curves follow a similar behavior, but the curve of the graph corresponding to "No. of elements processed" is more pronounced than the other.

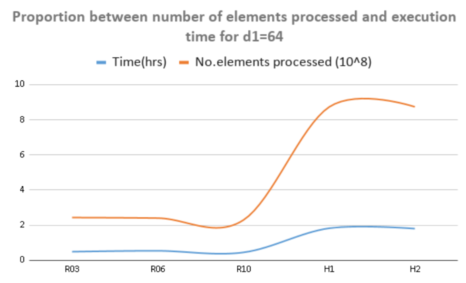
Returning to our hypothetical software $X$, for H1 and H2 datasets, we previously commented that in case that memory $M$ is not sufficient for $X$ to build the DBG (Case A in 4.6), our approach with experimented configuration would not be able to execute. However, if $M$ is sufficient to meet

Table 5.6: Datasets R03, R06, R10, H1 and H2. Time comparison, runtime DBG construction versus runtime DBG construction of our approach. Experiment 4.

| Datasets | $d_1 = 12$ (without no iterations ) | | $d_1 = 64$ (with 7 iterations) | |
|---|---|---|---|---|
| | Time(hrs) | $\mathcal{G}$: accumulated number of elements processed | Time(hrs) | $\mathcal{G}$: accumulated number of elements processed |
| R03 | 0.87 | 519,776,242 | 0.50 | 244,084,848 |
| R06 | 0.85 | 323,206,304 | 0.55 | 241,004,332 |
| R10 | 0.78 | 372,020,480 | 0.46 | 230,123,404 |
| H1 | 3.64 | 1,411,173,072 | 1.83 | 872,853,845 |
| H2 | 3.16 | 1,412,922,661 | 1.81 | 873,518,772 |



5.6(a): Execution for $d_1 = 12$. DBG traditional construction



5.6(b): Execution for $d_1 = 64$. DBG construction through multiple iterations using our approach.

Figure 5.6: Proportion between number of elements processed and execution time.

the requirements of this execution (Case B in 4.6), our approach would have gains over execution time.

## 5.2.4 Comparison with other assemblers

To evaluate the performance of our approach, we compared its results with common assemblers. In that case, we select ABYSS [Simpson et al. 2009] and Velvet [Zerbino 2016] as they are commonly used. In competitions such as Assemblathon and Gage they appear as assemblers most frequently used among those selected by competing teams. In both cases, they construct an exact representation of the DBG (in case of ABYSS we executed the version with hast table instead of the BF version).

In the case of ABYSS, it starts generating all $k$-mers and save the uniques in a hash table using a 2bit codification and a bitmap for edges representation. After getting the set of vertices $V$, it is traversed, and the edges are generated not over the reads but tested for each $k$-mer the existence of all possible extensions in $V$ .

Velvet, in turn, has different processing and data structures. Firstly, it generates all $k$-mers and saves them into a hash table, specifically into a splay tree that resides in each bucket to manage collisions. For each $k$-mer, the position in the read and the read id is tracked, generating the Roadmap file. After, using the Roadmap and the sequences files as inputs, are created the vertices and the edges to finish the graph generation.

In Experiment 5, we tried to execute the assembly for the same datasets used in Experiment 1 using ABYSS and Velvet with $k = 12$. Then, for these executions, we measure the time and memory until the step of building DBG.

In Table 5.7, we present the memory and run time resulting from the execution of our approach and ABYSS. Firstly, we present the results of our implementation using $d_1 = 12$, for a unique iteration, which is corresponded to the construct directly the DBG (see Experiment 2). Secondly, it is presented the results of the execution of our implementation for $d_1 = 64$ (see Experiment 1), which corresponds to the execution of our approach with seven iterations. For the last, it is shown memory peaks for the intermediate extra-compacted DBGs and the iteration $i$ in which they occur. In the same way, we showed the minor memory needed, and the memory of the last iteration, which corresponds to the final DBG. Finally, the values for ABYSS are presented.

It is possible to compare ABYSS with the execution of our implementation for $d_1 = 12$, which is corresponded to the construct directly the DBG. As well as, ABYSS could be compared with the execution of our implementation using our approach through some iterations.

Table 5.7: Comparison of our approach with ABYSS. Experiment 5.

| Datasets | Our approach ($d_1 = 12$ without no iterations ) | | Our approach ($d_1 = 64$ with 7 iterations) | | | | | | ABYSS | |
|---|---|---|---|---|---|---|---|---|---|---|
| | Time (hrs) | Mem. | Time (hrs) | Mem. max | i | Mem. min | i | Mem. for last i | Time (hrs) | Mem. |
| R03 | 0.87 | 1.63 | 0.50 | 2.57 | 6 | 1.00 | 1 | 1.63 | 1.11 | 1.01 |
| R06 | 0.85 | 1.65 | 0.55 | 2.79 | 6 | 0.96 | 1 | 1.65 | 1.03 | 1.02 |
| R10 | 0.78 | 1.68 | 0.46 | 2.78 | 6 | 0.91 | 1 | 1.68 | 1.14 | 1.04 |
| H1 | 3.64 | 1.60 | 1.83 | 5.81 | 5 | 1.60 | 7 | 1.60 | 5.61 | 1.00 |
| H2 | 3.16 | 1.60 | 1.81 | 5.81 | 5 | 1.60 | 7 | 1.60 | 5.37 | 1.00 |

As shown in graph Fig. 5.7, the execution time follows the same behaviour in the three cases. Nevertheless, ABYSS presents longer times, which are around twice the times for our implementation with $d_1 = 64$. The execution using our approach with $d_1 = 64$ and $step = 10$, got almost half of the time to directly generate the DBG using $d_1 = 12$. Comparing ABYSS with our implementation for $d_1 = 12$, given that both construct the graph using the same traditional approach, it may be visible a significant difference of ABYSS for Humans Ch14 datasets. It is worth to note that ABYSS generates all vertices firstly and visits each one after to "found" its possible edges. Instead, our approach generates the set of edges while getting the vertices.

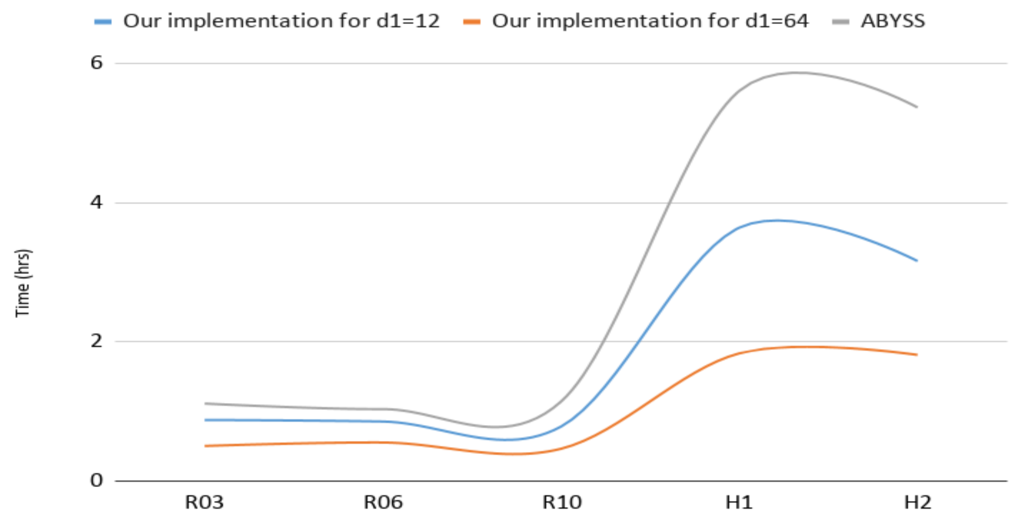


Figure 5.7: Comparison of execution time between our implementation and ABYSS

Another possible fair comparison would be between the memory used by ABYSS and the memory used by our implementation for $d_1 = 12$, when it generated the final DBG directly. The latest is equal to the required memory for final iteration $i$=7, when it is used $d_1 = 64$. In that sense, our implementation follows the same trend that ABYSS (see Fig. 5.8). It is appreciable an overhead of around 0.6GB for executions with our implementation, which is justifiable by the fact that our structure is not optimized for final $k$-mer. Rather, for intermediates long $dk$-mers that could take the best advantage of its design.

Notwithstanding, the distribution of unique $dk$-mers for tested parameters requires up to three times the memory needed for final DBG for human datasets. This point was previously discussed in section 5.2.2.

Turning to Velvet, to our surprise, it does not permit executions when $k$ is an even number. Thus, we decided use $k = 13$.

As evidenced, Velvet's memory consumption is higher than that reported using our approach for each tested dataset. Even for human datasets, Velvet

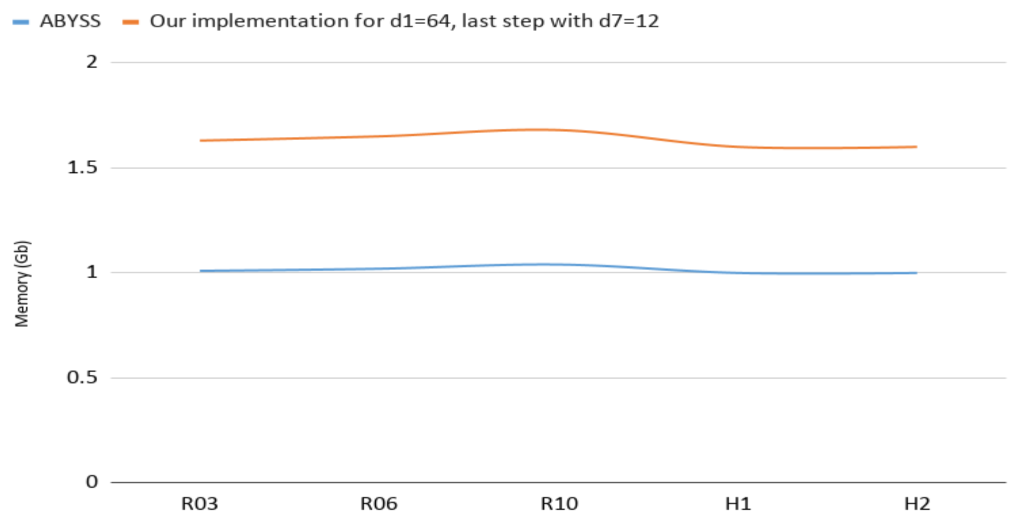


Figure 5.8: Comparison of memory between our implementation and ABYSS

Table 5.8: Comparison of our approach with Velvet. Experiment 5.

| Datasets | Our approach ($d_1 = 64$ with 7 iterations) | | | | | | Velvet | |
|---|---|---|---|---|---|---|---|---|
| | Time (hrs) | Mem. max | i | Mem. min | i | Mem. for last i | Time (hrs) | Mem. |
| R03 | 0.60 | 4.39 | 7 | 1.23 | 1 | 4.39 | 1.26 | 5.71 |
| R06 | 0.69 | 4.57 | 7 | 1.19 | 1 | 4.57 | 1.16 | 9.87 |
| R10 | 0.78 | 4.75 | 7 | 1.17 | 1 | 4.75 | 1.45 | 11.65 |
| H1 | 2.39 | 7.27 | 5 | 3.26 | 1 | 4.83 | over 1.15 | over 9.92 |
| H2 | 2.37 | 7.27 | 5 | 3.26 | 1 | 4.83 | over 1.37 | over 9.99 |

was not able to finish the execution. For over 1 hour for both datasets and 9GB of memory allocated, it could not assign more memory to continue its execution, reporting:

```
velvetg: Can't calloc 1917461804 InsertionMarkers totalling
17257156236 bytes: Cannot allocate memory
```

Finally, we selected an experiment (Experiment 6 in Table 5.1) to illustrate the advantages of our approach in case the memory is not enough (Case A in 4.6). For that case we used the dataset for Bombus impatiens (bumblebee), B2, and execute our experiment using $k = 31$, estimating 2,820,000,000 $k$-mers for 30 millions reads.

In the case of ABYSS, after 3.45hrs, with a load hash factor of 715400895 / 2147483648 = 0.333 using 32.6GB, informs a memory problem:

```
sparsehash FATAL ERROR: failed to allocate 27 groups.
```

Turning to Velvet, during the execution of *velveth*, was reported in 0.65hrs:

```
[2343.799240] Inputting sequence 15535000 / 30000000
[2343.825408] No more memory for memory chunk!
```

Using our approach, we done following executions (see details in Table 5.9 ):

Table 5.9: B2. Number of skipped $k$-mers for Experiment 6.

| i | k | d | $g'(i)$: number of unique $dk$-mers | $g(i)$: total number of $dk$-mers | $p(i)$: skipped $k$-mers | $P(i)$: accumulated skipped $k$-mers | $P(i)/N$ (%) | $Mem.$ (Gb) |
|---|---|---|---|---|---|---|---|---|
| $d_1 = 55$ , $step = 2$ and $step = 10$ | | | | | | | | |
| 1 | 31 | 55 | 99,761,700 | 120,000,000 | 497,591,712 | 497,591,712 | 17.65 | 21.77 |
| $d_1 = 100$ , $step = 10$ | | | | | | | | |
| 1 | 31 | 100 | 56,708,579 | 60,000,000 | 172,900,804 | 172,900,804 | 6.13 | 13.07 |
| 2 | 31 | 90 | 81,504,041 | 84,667,129 | 75,602,880 | 248,503,684 | 8.81 | 16.77 |
| 3 | 31 | 80 | 103,426,703 | 108,583,151 | 88,753,440 | 337,257,124 | 11.96 | 21.67 |
| 4 | 31 | 70 | 123,535,137 | 129,576,089 | 87,236,300 | 424,493,424 | 15.05 | 26.07 |
| 5 | 31 | 60 | 142,207,182 | 148,790,297 | 83,998,330 | 508,491,754 | 18.03 | 30.47 |

For $d_1 = 55$, $step = 2$, the first iteration complete the execution in 0.8hrs bringing a gain of 17.65% $k$-mers skipped to being processed. However, if only the first iteration could be executed, the memory was insufficient for $d_2 = 53$. Then, we tested change *step* to 10.

For $d_1 = 55$, $step = 10$, as in the previous one execution, a result higher than 17.65% could not be achieved, because it was only possible to execute the 1st iteration given the existing memory.

At this moment, we turn to new value for $d_1$. Then was executed $d_1 = 100$, using $step = 10$. At this time it was possible to improve the result, obtaining 18.03%, what means 10,900,042 skipped $k$-mers more that the previous result. The last iteration completed was $i = 5$, for $d_5 = 60$ using 30.47GB of memory.

## 5.3 Partial conclusions

In this chapter, we presented details about the implementation used to validate our approach. Some experiments were made in function to analyze the number of skipped $k$-mers, i.e, $k$-mer that will not need to be processed, and how the parameters $d_1$ and *step* could affect them. Also, a subsection is dedicated to memory and time consumption, showing how they depend heavily on the number of unique $dk$-mers and the number of total elements processed.

Also, we could somehow validate our implementation through the comparison with ABYSS and Velvet, two typical used assemblers, proving that the results are not out of the range of traditional approaches.

Finally, we showed a real case in which the memory is not enough (Case A in 4.6), reaching 18.03% of skipped $k$-mers, what means 508,491,754 $k$-mers were skipped to being processed in external memory solution.

# 6
# Conclusions

The creation and manipulation of the *de Bruijn* graph have been identified as the step with most memory consumption for *de novo* assembly. The computational requirements of DBG construction depends on the process of a **huge** number of elements, $k$-mers.

The computational requirements for the construction of DBG are influenced by several aspects:

- The number of total $k$-mers $N$, given by $(m - k + 1)n$ assuming that all reads has the same length $m$.
- It is not known which is the $k$ value for best assembly.
- The number of unique $k$-mers $N\varphi$, $0 < \varphi < 1$, without knowing in advance.
- High level of redundancy between adjacent $k$-mers.
- Size overhead of data structure to used to identify the set of $V$.
- Search time of the data structure used to store $V$.
- $K$-mer codification (vertex codification).
- $K$-mer adjacency codification (edges codification).

In this thesis, we present a new approach to construct *de Bruijn* Graph without the necessity of process all $k$-mers. Through an iterative sequence of reductions, it is possible to process the graph as much as possible in the main memory, and only when the available main memory becomes insufficiently, will be using an external memory solution. Then, large duplicate regions had been already identified, avoiding processing a significant amount of duplicated $k$-mers in external memory, reducing the number of I/O operations.

A prototype of our approach was implemented, and some tests were executed. The analysis of the number of skipped $k$-mers in each iteration showed that our approach saves a significant number of $k$-mers to be processed. In order to need to externalize the process at some point, these results suggest that a significant number of I/O operations will be saved, improving the computational requirements.

Also, we have shown that the number of processed elements affects the execution time, while the unique number significantly impacts the amount of memory required for an exact representation of DBG.

Finally, it is shown the impact of parameters in the amount of skipped $k$-mers, which opens a new possibility for parameter tuning.

## 6.1 Contributions

Our proposed approach, as far as we know, is unique and innovative in the sense that combines two principles for DBG construction:

- the reduction of the number of $k$-mers to be processed, bringing a positive impact on the run time for both RAM-only and external memory model processing.
- postpose the external memory processing (if needed) for the last steps of the algorithm, reducing the total number of $k$-mers to be externalized and, consequently, the number of I/O operations.

Furthermore, this thesis brings a set of additional contributions, as listed below:

- Identification and formalization of the main variables that impact the DBG construction.
- A survey of *de Bruijn* graph approaches, emphasizing the data structures, processing algorithms, and disk distribution algorithms used.
- Possibility of tuning the parameters to obtain a higher number of $k$-mers that do not have to be processed.
- A proposed implementation of the approach.
- Proposition and implementation of data structure optimized for large $dk$-mers.
- Performance evaluation of the approach in real genome datasets, including three kinds of organisms: animals, humans, and plants.

## 6.2 Future work

- Study of how much a $dk$-graph for specific $i$ iteration is approximated to a final DBG, and how it can be exploited in the assembly process.
- Study of the distributions of unique $dk$-mers to estimate $d_1$ and *step*.
- Study of replication factor in order to estimate time execution.

# Bibliography references

[Aggarwal and Vitter 1988] Aggarwal, A. and Vitter, Jeffrey, S. (1988). The Input/Output Complexity of Sorting and Related Problems. *Commun. ACM*, 31(9):1116–1127.

[Bradnam et al. 2013] Bradnam, K. R. et al. (2013). Assemblathon 2: evaluating de novo methods of genome assembly in three vertebrate species. *GigaScience*, 2(1):1–31.

[Butler et al. 2008] Butler, J., MacCallum, I., Kleber, M., Shlyakhter, I. A., Belmonte, M. K., Lander, E. S., Nusbaum, C., and Jaffe, D. B. (2008). ALLPATHS: De novo assembly of whole-genome shotgun microreads. *Genome Research*, 18(5):810–820.

[Chapman et al. 2011] Chapman, J. A., Ho, I., Sunkara, S., Luo, S., Schroth, G. P., and Rokhsar, D. S. (2011). Meraculous: de novo genome assembly with short paired-end reads. *PloS one*, 6(8):e23501–e23501.

[Chikhi et al. 2014] Chikhi, R., Limasset, A., Jackman, S., Simpson, J. T., and Medvedev, P. (2014). *On the Representation of de Bruijn Graphs*, pages 35–55. Springer International Publishing, Cham, Switzerland.

[Chikhi et al. 2016] Chikhi, R., Limasset, A., and Medvedev, P. (2016). Compacting de Bruijn graphs from sequencing data quickly and in low memory. *Bioinformatics*, 32(12):i201.

[Chikhi and Rizk 2013] Chikhi, R. and Rizk, G. (2013). Space-efficient and exact de Bruijn graph representation based on a Bloom filter. *Algorithms for Molecular Biology*, 8(1):22.

[Claros et al. 2012] Claros, M. G., Bautista, R., Guerrero-Fernández, D., Benzerki, H., Seoane, P., and Fernández-Pozo, N. (2012). Why assembling plant genome sequences is so challenging. *Biology*, 1(2):439.

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

[Cook and Zilles 2009] Cook, J. J. and Zilles, C. (2009). Characterizing and optimizing the memory footprint of de novo short read DNA sequence assembly.

In *International Symposium on Performance Analysis of Systems and Software. ISPASS 2009*, pages 143–152.

[de Armas et al. 2016] de Armas, E. M., Haeusler, E. H., Lifschitz, S., de Holanda, M. T., da Silva, W. M. C., and Ferreira, P. C. G. (2016). K-mer Mapping and de Bruijn graphs: The case for velvet fragment assembly. In *2016 IEEE International Conference on Bioinformatics and Biomedicine (BIBM)*, pages 882–889.

[de Armas et al. 2017] de Armas, E. M., Silva, M. V. M., and Lifschitz, S. (2017). A Study of Index Structures for K-mer Mapping. In *Proceedings Satellite Events of the 32nd Brazilian Symposium on Databases. Databases Meet Bioinformatics Workshop*, pages 326–333.

[Deorowicz et al. 2013] Deorowicz, S., Debudaj-Grabysz, A., and Grabowski, S. (2013). Disk-based k-mer counting on a PC. *BMC Bioinformatics*, 14(1):160.

[Deorowicz et al. 2015] Deorowicz, S., Kokot, M., Grabowski, S., and Debudaj-Grabysz, A. (2015). KMC 2: fast and resource-frugal k-mer counting. *Bioinformatics*, 31(10):1569.

[Earl et al. 2011] Earl, D. et al. (2011). Assemblathon 1: A competitive assessment of de novo short read assembly methods. *Genome Research*, 21(12):2224–2241.

[El-Metwally et al. 2013] El-Metwally, S., Hamza, T., Zakaria, M., and Helmy, M. (2013). Next-generation sequence assembly: Four stages of data processing and computational challenges. *PLoS Comput Biol*, 9(12):1–19.

[Erbert et al. 2017] Erbert, M., Rechner, S., and Müller-Hannemann, M. (2017). Gerbil: a fast and memory-efficient k-mer counter with GPU-support. *Algorithms for Molecular Biology*, 12(1):9:1–9:12.

[Ghosh and Kalyanaraman 2016] Ghosh, P. and Kalyanaraman, A. (2016). A Fast Sketch-based Assembler for Genomes. In *Proceedings of the 7th ACM International Conference on Bioinformatics, Computational Biology, and Health Informatics*, BCB '16, pages 241–250, New York, NY, USA. ACM.

[Ghosh and Kalyanaraman 2019] Ghosh, P. and Kalyanaraman, A. (2019). FastEtch: A Fast Sketch-Based Assembler for Genomes. *IEEE/ACM Trans. Comput. Biology Bioinform.*, 16(4):1091–1106.

[Gnerre et al. 2011] Gnerre, S., Maccallum, I., Przybylski, D., Ribeiro, F. J., Burton, J. N., Walker, B. J., Sharpe, T., Hall, G., Shea, T. P., Sykes, S., Berlin, A. M., Aird, D., Costello, M., Daza, R., Williams, L., Nicol, R., Gnirke, A., Nusbaum, C., Lander, E. S., and Jaffe, D. B. (2011). High-quality draft assemblies of mammalian genomes from massively parallel sequence data. *Proceedings of the National Academy of Sciences of the United States of America*, 108(4):1513–1518. 21187386[pmid].

[Illumina, Inc. 2019] Illumina, Inc. (2019). Sequencing coverage. `https://www.illumina.com/science/technology/next-generation-sequencing/plan-experiments/coverage.html`. Retrieved 2019-10-1.

[Jackman and Birol 2010] Jackman, S. D. and Birol, I. (2010). Assembling genomes using short-read sequencing technology. *Genome biology*, 11(1):202–202. 20128932[pmid].

[Kelley et al. 2010] Kelley, D. R., Schatz, M. C., and Salzberg, S. L. (2010). Quake: quality-aware detection and correction of sequencing errors. *Genome Biology*, 11(11):R116.

[Kleftogiannis et al. 2013] Kleftogiannis, D., Kalnis, P., and Bajic, V. B. (2013). Comparing Memory-Efficient Genome Assemblers on Stand-Alone and Cloud Infrastructures. *PLoS ONE*, 8(9).

[Kokot et al. 2017] Kokot, M., Dlugosz, M., and Deorowicz, S. (2017). KMC 3: counting and manipulating k-mer statistics. *Bioinformatics*, 33(17):2759–2761.

[Kundeti et al. 2010] Kundeti, V., Rajasekaran, S., and Dinh, H. (2010). Efficient parallel and out of core algorithms for constructing large bi-directed de Bruijn graphs. *ArXiv e-prints*.

[Li et al. 2009] Li, R. et al. (2009). De novo assembly of human genomes with massively parallel short read sequencing. *Genome Research*.

[Li et al. 2013] Li, Y., Kamousi, P., Han, F., Yang, S., Yan, X., and Suri, S. (2013). Memory Efficient Minimum Substring Partitioning. *Proc. VLDB Endow.*, 6(3):169–180.

[Li and XifengYan 2015] Li, Y. and XifengYan (2015). MSPKmerCounter: A Fast and Memory Efficient Approach for K-mer Counting. *ArXiv e-prints*.

[Li et al. 2011] Li, Z., Chen, Y., Mu, D., Yuan, J., Shi, Y., Zhang, H., Gan, J., Li, N., Hu, X., Liu, B., Yang, B., and Fan, W. (2011). Comparison of the two major classes of assembly algorithms: overlap–layout–consensus and de-bruijn-graph. *Briefings in Functional Genomics*, 11(1):25–37.

[Liu et al. 2012] Liu, L., Li, Y., Li, S., Hu, N., He, Y., Pong, R., Lin, D., Lu, L., and Law, M. (2012). Comparison of Next-Generation Sequencing Systems. *Journal of biomedicine & biotechnology*, 2012:251364.

[Luo et al. 2012] Luo, R. et al. (2012). SOAPdenovo2: an empirically improved memory-efficient short-read de novo assembler. *GigaScience*, 1(1):1–6.

[Mamun et al. 2016] Mamun, A.-A., Pal, S., and Rajasekaran, S. (2016). KCMBT: a k -mer Counter based on Multiple Burst Trees . *Bioinformatics*, 32(18):2783.

[Marcais and Kingsford 2011] Marcais, G. and Kingsford, C. (2011). A fast, lock-free approach for efficient parallel counting of occurrences of k-mers. *Bioinformatics*, 27(6):764–770.

[Massachusetts Institute of Technology 2012] Massachusetts Institute of Technology, E. D. (2012). Lecture notes in Advanced Data Structures, MIT course number 6.851. `https://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-851-advanced-data-structures-spring-2012/calendar-and-notes/MIT6_851S12_L7.pdf`.

[McVicar et al. 2017] McVicar, N., Lin, C., and Hauck, S. (2017). K-Mer Counting Using Bloom Filters with an FPGA-Attached HMC. In *25th IEEE Annual International Symposium on Field-Programmable Custom Computing Machines, FCCM 2017, Napa, CA, USA, April 30 - May 2, 2017*, pages 203–210.

[Melsted and Pritchard 2011] Melsted, P. and Pritchard, J. K. (2011). Efficient counting of k-mers in DNA sequences using a bloom filter. *BMC Bioinformatics*, 12(1):333.

[Metzker 2010] Metzker, M. L. (2010). Sequencing technologies - the next generation. *Nature reviews. Genetics*, 11(1):31–46.

[Miller et al. 2010] Miller, J. R., Koren, S., and Sutton, G. (2010). Assembly algorithms for next-generation sequencing data. *Genomics*, 95(6):315–327. 20211242[pmid].

[Minkin et al. 2016] Minkin, I., Pham, S. K., and Medvedev, P. (2016). TwoPaCo: An efficient algorithm to build the compacted de Bruijn graph from many complete genomes. *CoRR*, abs/1602.05856.

[Myers 1995] Myers, E. W. (1995). Toward Simplifying and Accurately Formulating Fragment Assembly. *Journal of Computational Biology*, 2(2):275–290. PMID: 7497129.

[Myers et al. 2000] Myers, E. W., Sutton, G. G., Delcher, A. L., Dew, I. M., Fasulo, D. P., Flanigan, M. J., Kravitz, S. A., Mobarry, C. M., Reinert, K.

H. J., Remington, K. A., Anson, E. L., Bolanos, R. A., Chou, H.-H., Jordan, C. M., Halpern, A. L., Lonardi, S., Beasley, E. M., Brandon, R. C., Chen, L., Dunn, P. J., Lai, Z., Liang, Y., Nusskern, D. R., Zhan, M., Zhang, Q., Zheng, X., Rubin, G. M., Adams, M. D., and Venter, J. C. (2000). A Whole-Genome Assembly of Drosophila. *Science*, 287(5461):2196–2204.

[Neustar, Inc. 2016] Neustar, Inc. (2016). Big memory, part 3.5: Google sparsehash! `https://research.neustar.biz/2011/11/27/big-memory-part-3-5-google-sparsehash/`. Retrieved 2019-09-10.

[Niedringhaus et al. 2011] Niedringhaus, T. P., Milanova, D., Kerby, M. B., Snyder, M. P., and Barron, A. E. (2011). Landscape of next-generation sequencing technologies. *Analytical chemistry*, 83(12):4327–4341.

[Pandey et al. 2017] Pandey, P., Bender, M. A., Johnson, R., and Patro, R. (2017). deBGR: an efficient and near-exact representation of the weighted de Bruijn graph. *Bioinformatics*, 33(14):i133–i141.

[Pell et al. 2012] Pell, J., Hintze, A., Canino-Koning, R., Howe, A., Tiedje, J., and Brown, C. T. (2012). Scaling metagenome sequence assembly with probabilistic de bruijn graphs. *Proceedings of the National Academy of Sciences of the United States of America*, 109:13272–7.

[Penman's 2017] Penman's, T. (2017). Sparsehash internals. `http://tristanpenman.com/blog/posts/2017/10/11/sparsehash-internals/#hash-collisions`.

[Rahman et al. 2017] Rahman, M. M., Sharker, R., Biswas, S., and Rahman, M. (2017). HaVec: An Efficient de Bruijn Graph Construction Algorithm for Genome Assembly. *International Journal of Genomics*, 2017:1–12.

[Rizk et al. 2013] Rizk, G., Lavenier, D., and Chikhi, R. (2013). DSK: k-mer counting with very low memory usage. *Bioinformatics*, 29(5):652–653.

[Salikhov et al. 2014] Salikhov, K., Sacomoto, G., and Kucherov, G. (2014). Using Cascading Bloom Filters to Improve the Memory Usage for de Brujin Graphs. *Algorithms for molecular biology : AMB*, 9:2.

[Salzberg et al. 2012] Salzberg, S. L. et al. (2012). GAGE: A critical evaluation of genome assemblies and assembly algorithms. *Genome Research*, 22(3):557–567.

[Sanger et al. 1980] Sanger, F., Coulson, A., Barrell, B., Smith, A., and Roe, B. (1980). Cloning in single-stranded bacteriophage as an aid to rapid DNA sequencing. *Journal of Molecular Biology*, 143(2):161 – 178.

[Santa Brigida et al. 2016] Santa Brigida, A. B., Rojas, C. A., Grativol, C., de Armas, E. M., Entenza, J. O. P., Thiebaut, F., Lima, M. d. F., Farrinelli, L., Hemerly, A. S., Lifschitz, S., and Ferreira, P. C. G. (2016). Sugarcane transcriptome analysis in response to infection caused by Acidovorax avenae subsp. avenae. *PLOS ONE*, 11(12):1–30.

[Schatz et al. 2010] Schatz, M. C., Delcher, A. L., and Salzberg, S. L. (2010). Assembly of large genomes using second-generation sequencing. *Genome Research*, 20(9):1165–1173.

[Schatz et al. 2012] Schatz, M. C., Witkowski, J., and McCombie, W. R. (2012). Current challenges in de novo plant genome sequencing and assembly. *Genome Biology*, 13(4):1–7.

[Silva et al. 2017] Silva, M. V. M., de Holanda, M. T., Haeusler, E. H., de Armas, E. M., and Lifschitz, S. (2017). VelvetH-DB: Persistência de Dados no Processo de Montagem de Fragmentos de Sequências Biológicas. In *Proceedings Satellite Events of the 32nd Brazilian Symposium on Databases. Databases Meet Bioinformatics Workshop*, pages 334–341.

[Simpson and Durbin 2010] Simpson, J. T. and Durbin, R. (2010). Efficient construction of an assembly string graph using the FM-index. *Bioinformatics (Oxford, England)*, 26(12):i367–i373.

[Simpson et al. 2009] Simpson, J. T., Wong, K., Jackman, S. D., Schein, J. E., Jones, S. J., and Birol, I. (2009). ABySS: A parallel assembler for short read sequence data. *Genome Research*, 19(6):1117–1123.

[Sims et al. 2014] Sims, D., Sudbery, I., Ilott, N. E., Heger, A., and Ponting, C. P. (2014). Sequencing depth and coverage: key considerations in genomic analyses. *Nature Reviews Genetics*, 15:121 EP –. Review Article.

[Thiebaut et al. 2017] Thiebaut, F., Rojas, C. A., Grativol, C., Calixto, E. P. d. R., Motta, M. R., Ballesteros, H. G. F., Peixoto, B., de Lima, B. N. S., Vieira, L. M., Walter, M. E., de Armas, E. M., Entenza, J. O. P., Lifschitz, S., Farinelli, L., Hemerly, A. S., and Ferreira, P. C. G. (2017). Roles of Non-Coding RNA in Sugarcane-Microbe Interaction. *Non-coding RNA*, 3(4):25. 29657296[pmid].

[Titus Brown et al. 2012] Titus Brown, C., Howe, A., Zhang, Q., Pyrkosz, A. B., and Brom, T. H. (2012). A Reference-Free Algorithm for Computational Normalization of Shotgun Sequencing Data. *arXiv e-prints*, page arXiv:1203.4802.

[Welch 2009] Welch, N. (2009). Hash table benchmarks. `http://incise.org/hash-table-benchmarks.html`. Retrieved 2019-09-10.

[Ye et al. 2011] Ye, C., Cannon, C. H., Ma, Z. S., Yu, D. W., and Pop, M. (2011). SparseAssembler2: Sparse k-mer Graph for Memory Efficient Genome Assembly. *arXiv e-prints*, page arXiv:1108.3556.

[Ye et al. 2012] Ye, C., Sam Ma, Z., Cannon, C., Pop, M., and Yu, D. (2012). Exploiting sparseness in de novo genome assembly. *BMC bioinformatics*, 13 Suppl 6:S1.

[Zavodna et al. 2014] Zavodna, M., Bagshaw, A., Brauning, R., and Gemmell, N. J. (2014). The Accuracy, Feasibility and Challenges of Sequencing Short Tandem Repeats Using Next-Generation Sequencing Platforms. *PLOS ONE*, 9(12):1–14.

[Zerbino 2016] Zerbino, D. (2016). Velvet software. EMBL-EBI. `https://www.ebi.ac.uk/zerbino/velvet/`. Retrieved 2019-6-15.

[Zerbino and Birney 2008] Zerbino, D. R. and Birney, E. (2008). Velvet: Algorithms for de novo short read assembly using de Bruijn graphs. *Genome Research*, 18(5):821–829.

[Zhang et al. 2014] Zhang, Q., Pell, J., Canino-Koning, R., Howe, A. C., and Brown, C. T. (2014). These Are Not the K-mers You Are Looking For: Efficient Online K-mer Counting Using a Probabilistic Data Structure. *PLOS ONE*, 9(7):1–13.

[Zhou et al. 2010] Zhou, X., Ren, L., Meng, Q., Li, Y., Yu, Y., and Yu, J. (2010). The next-generation sequencing technology and application. *Protein & cell*, 1(6):520–536. 21204006[pmid].

# A
# Decomposition details

Table A.1: Decomposition details for $G_{d_i,k}$.

| $i$ | $V_{d_i,k}$ | $E_{d_i,k}$ |
|---|---|---|
| 0 | [0, 78, 79] | [] |
| 1 | [0, 63, 64], [53, 78, 26] | [0,78] |
| 2 | [0, 53, 54], [43, 63, 21], [53, 78, 26] | [0,63], [43,78] |
| 3 | [0, 43, 44], [33, 53, 21], [43, 63, 21], [53, 78, 26] | [0,53], [33,63], [43,78] |
| 4 | [0, 33, 34], [23, 43, 21], [33, 53, 21], [43, 63, 21], [53, 78, 26] | [0,43], [23,53], [33,63], [43,78] |
| 5 | [0, 23, 24], [13, 33, 21], [23, 43, 21], [33, 53, 21], [43, 63, 21], [53, 76, 24], [66, 78, 13] | [0,33, [13,43], [23,53], [33,63], [43,76], [53,78] |
| 6 | [0, 13, 14], [3, 16, 14], [6, 19, 14], [9, 22, 14], [12, 23, 12], [13, 26, 14], [16, 29, 14], [19, 32, 14], [22, 33, 12], [23, 36, 14], [26, 39, 14], [29, 42, 14], [32, 43, 12], [33, 46, 14], [36, 49, 14], [39, 52, 14], [42, 53, 12], [43, 56, 14], [46, 59, 14], [49, 62, 14], [52, 63, 12], [53, 66, 14], [56, 69, 14], [59, 72, 14], [62, 75, 14], [65, 76, 12], [66, 78, 13] | [0, 16], [3, 19], [6, 22], [9, 23], [12, 26], [13, 29], [16, 32], [19, 33], [22, 36], [23, 39], [26, 42], [29, 43], [32, 46], [33, 49], [36, 52], [39, 53], [42, 56], [43, 59], [46, 62], [49, 63], [52, 66], [53, 69], [56, 72], [59, 75], [62, 76], [65, 78] |
| 7 | [0, 11, 12], [1, 12, 12], [2, 13, 12], [3, 14, 12], [4, 15, 12], [5, 16, 12], [6, 17, 12], [7, 18, 12], [8, 19, 12], [9, 20, 12], [10, 21, 12], [11, 22, 12], [12, 23, 12], [13, 24, 12], [14, 25, 12], [15, 26, 12], [16, 27, 12], [17, 28, 12], [18, 29, 12], [19, 30, 12], [20, 31, 12], [21, 32, 12], [22, 33, 12], [23, 34, 12], [24, 35, 12], [25, 36, 12], [26, 37, 12], [27, 38, 12], [28, 39, 12], [29, 40, 12], [30, 41, 12], [31, 42, 12], [32, 43, 12], [33, 44, 12], [34, 45, 12], [35, 46, 12], [36, 47, 12], [37, 48, 12], [38, 49, 12], [39, 50, 12], [40, 51, 12], [41, 52, 12], [42, 53, 12], [43, 54, 12], [44, 55, 12], [45, 56, 12], [46, 57, 12], [47, 58, 12], [48, 59, 12], [49, 60, 12], [50, 61, 12], [51, 62, 12], [52, 63, 12], [53, 64, 12], [54, 65, 12], [55, 66, 12], [56, 67, 12], [57, 68, 12], [58, 69, 12], [59, 70, 12], [60, 71, 12], [61, 72, 12], [62, 73, 12], [63, 74, 12], [64, 75, 12], [65, 76, 12], [66, 77, 12], [67, 78, 12] | [0, 12], [1, 13], [2, 14], [3, 15], [4, 16], [5, 17], [6, 18], [7, 19], [8, 20], [9, 21], [10, 22], [11, 23], [12, 24], [13, 25], [14, 26], [15, 27], [16, 28], [17, 29], [18, 30], [19, 31], [20, 32], [21, 33], [22, 34], [23, 35], [24, 36], [25, 37], [26, 38], [27, 39], [28, 40], [29, 41], [30, 42], [31, 43], [32, 44], [33, 45], [34, 46], [35, 47], [36, 48], [37, 49], [38, 50], [39, 51], [40, 52], [41, 53], [42, 54], [43, 55], [44, 56], [45, 57], [46, 58], [47, 59], [48, 60], [49, 61], [50, 62], [51, 63], [52, 64], [53, 65], [54, 66], [55, 67], [56, 68], [57, 69], [58, 70], [59, 71], [60, 72], [61, 73], [62, 74], [63, 75], [64, 76], [65, 77], [66, 78] |

Table A.2: Number of elements generated in each iteration varying *step* for *update* function.

| d | *step* = 5 | | | *step* = 10 | | | *step* = 15 | | | *step* = 20 | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | i | No. | Length | i | No. | Length | i | No. | Length | i | No. | Length |
| *input* | | | | *input* | | | *input* | | | *input* | | |
| | 0 | 1 | 79 | 0 | 1 | 79 | 0 | 1 | 79 | 0 | 1 | 79 |
| *output* | | | | *output* | | | *output* | | | *output* | | |
| 64 | 1 | 1 | 64 | 1 | 1 | 64 | 1 | 1 | 64 | 1 | 1 | 64 |
| | | 1 | 26 | | 1 | 26 | | 1 | 26 | | 1 | 26 |
| 59 | 2 | 1 | 59 | | | | | | | | | |
| | | 1 | 26 | | | | | | | | | |
| | | 1 | 16 | | | | | | | | | |
| 54 | 3 | 1 | 54 | 2 | 1 | 54 | | | | | | |
| | | 1 | 26 | | 1 | 21 | | | | | | |
| | | 2 | 16 | | 1 | 26 | | | | | | |
| 49 | 4 | 1 | 49 | | | | 2 | 1 | 49 | | | |
| | | 1 | 26 | | | | | 2 | 26 | | | |
| | | 3 | 16 | | | | | | | | | |
| 44 | 5 | 1 | 44 | 3 | 1 | 44 | | | | 2 | 1 | 44 |
| | | 1 | 26 | | 2 | 21 | | | | | 1 | 31 |
| | | 4 | 16 | | 1 | 26 | | | | | 1 | 26 |
| 39 | 6 | 1 | 39 | | | | | | | | | |
| | | 1 | 26 | | | | | | | | | |
| | | 5 | 16 | | | | | | | | | |
| 34 | 7 | 1 | 34 | 4 | 1 | 34 | 3 | 1 | 34 | | | |
| | | 1 | 26 | | 3 | 21 | | 3 | 26 | | | |
| | | 6 | 16 | | 1 | 26 | | | | | | |
| 29 | 8 | 1 | 29 | | | | | | | | | |
| | | 1 | 26 | | | | | | | | | |
| | | 7 | 16 | | | | | | | | | |
| 24 | 9 | 2 | 24 | 5 | 2 | 24 | | | | 3 | 4 | 24 |
| | | 8 | 16 | | 4 | 21 | | | | | 2 | 18 |
| | | 1 | 13 | | 1 | 13 | | | | | 1 | 13 |
| 19 | 10 | 2 | 19 | | | | 4 | 5 | 19 | | | |
| | | 10 | 16 | | | | | 4 | 18 | | | |
| | | 1 | 13 | | | | | | | | | |
| 14 | 11 | 14 | 14 | 6 | 20 | 14 | | | | | | |
| | | 13 | 13 | | 1 | 13 | | | | | | |
| | | | | | 6 | 12 | | | | | | |
| 12 | 12 | 12 | 68 | 7 | 68 | 12 | 5 | 68 | 12 | 4 | 68 | 12 |

# B
# Results

**Experiments for analyses the number of *k*-mers avoiding to process in each iteration (Section 5.2.1)**

Experiments with $k = 12$, $d_1 = 64$, $update(d_i) = d_{i-1} - 10$ over:

***Sugar cane libraries.*** Fragment libraries collected from Brazilian sugarcane species by IBqM of UFRJ with $n$ number of reads, and $m$ is the read length:

- R03 with $n = 8,520,922$ and $m = 72$ (Table B.1)
- R06 with $n = 5,298,464$ and $m = 72$ (Table B.2)
- R10 with $n = 5,723,392$ and $m = 76$ (Table B.3)

Table B.1: Dataset R03. Number of skipped $k$-mers. Experiment 1.

| i | k | d | $g'(i)$: number of unique $dk$-mers | $g(i)$: total number of $dk$-mers | $p(i)$: skipped $k$-mers | $P(i)$: accumulated skipped $k$-mers | $P(i)/N$ (%) |
|---|---|---|---|---|---|---|---|
| 1 | 12 | 64 | 9,604,846 | 17,041,844 | 150,295,814 | 150,295,814 | 28.92 |
| 2 | 12 | 54 | 12,840,309 | 16,107,994 | 34,675,825 | 184,971,639 | 35.59 |
| 3 | 12 | 44 | 15,113,476 | 19,282,882 | 54,244,493 | 239,216,132 | 46.02 |
| 4 | 12 | 34 | 16,517,971 | 21,010,378 | 62,355,237 | 301,571,369 | 58.02 |
| 5 | 12 | 24 | 17,616,768 | 21,074,014 | 38,444,116 | 340,015,485 | 65.42 |
| 6 | 12 | 14 | 31,321,779 | 70,630,865 | 100,823,886 | 440,839,371 | 84.81 |
| 7 | 12 | 12 | 13,881,227 | 78,936,871 | 65,055,644 | | |

***Human Chromosome 14***. Fragment library of Human Chromosome 14 (Ch14) available in `http://gage.cbcb.umd.edu/data/` with $n$ number of reads, and $m$ is the read length.:

- H1 with $n = 18,166,705$ and $m = 101$ in average. (Table B.4)
- H2 with $n = 18,166,798$ and $m = 101$ in average. (Table B.5)

Table B.2: Dataset R06. Number of skipped $k$-mers. Experiment 1.

| i | k | d | $g'(i)$: number of unique $dk$-mers | $g(i)$: total number of $dk$-mers | $p(i)$: skipped $k$-mers | $P(i)$: accumulated skipped $k$-mers | $P(i)/N$ (%) |
|---|---|---|---|---|---|---|---|
| 1 | 12 | 64 | 8,385,453 | 10,596,928 | 31,343,315 | 31,343,315 | 9.70 |
| 2 | 12 | 54 | 11,842,147 | 13,380,550 | 15,851,739 | 47,195,054 | 14.60 |
| 3 | 12 | 44 | 14,626,588 | 16,823,071 | 26,371,883 | 73,566,937 | 22.76 |
| 4 | 12 | 34 | 16,634,211 | 19,415,901 | 38,908,032 | 112,474,969 | 34.80 |
| 5 | 12 | 24 | 18,114,159 | 20,570,360 | 27,445,703 | 139,920,672 | 43.29 |
| 6 | 12 | 14 | 35,010,061 | 72,041,198 | 95,109,308 | 235,029,980 | 72.72 |
| 7 | 12 | 12 | 14,231,519 | 88,176,324 | 73,944,805 | | |

Table B.3: Dataset R10. Number of skipped $k$-mers. Experiment 1.

| i | k | d | $g'(i)$: number of unique $dk$-mers | $g(i)$: total number of $dk$-mers | $p(i)$: skipped $k$-mers | $P(i)$: accumulated skipped $k$-mers | $P(i)/N$ (%) |
|---|---|---|---|---|---|---|---|
| 1 | 12 | 64 | 6,807,003 | 11,446,784 | 136,662,950 | 136,662,950 | 36.74 |
| 2 | 12 | 54 | 9,784,098 | 10,555,137 | 9,293,103 | 145,956,053 | 39.23 |
| 3 | 12 | 44 | 12,292,626 | 13,484,271 | 13,145,708 | 159,101,761 | 42.77 |
| 4 | 12 | 34 | 14,595,400 | 15,939,353 | 14,348,854 | 173,450,615 | 46.62 |
| 5 | 12 | 24 | 16,810,095 | 18,172,179 | 13,911,612 | 187,362,227 | 50.36 |
| 6 | 12 | 14 | 34,635,823 | 70,720,235 | 94,852,808 | 282,215,035 | 75.86 |
| 7 | 12 | 12 | 14,654,968 | 89,805,445 | 75,150,477 | | |

Table B.4: H1. Number of skipped $k$-mers. Experiment 1.

| i | k | d | $g'(i)$: number of unique $dk$-mers | $g(i)$: total number of $dk$-mers | $p(i)$: skipped $k$-mers | $P(i)$: accumulated skipped $k$-mers | $P(i)/N$ (%) |
|---|---|---|---|---|---|---|---|
| 1 | 12 | 64 | 29,919,316 | 33,259,320 | 151,988,621 | 151,988,621 | 9.30 |
| 2 | 12 | 54 | 42,743,049 | 44,174,724 | 20,572,265 | 172,560,886 | 10.55 |
| 3 | 12 | 44 | 61,420,685 | 68,073,331 | 103,932,120 | 276,493,006 | 16.91 |
| 4 | 12 | 34 | 77,411,456 | 86,833,283 | 107,681,659 | 384,174,665 | 23.50 |
| 5 | 12 | 24 | 89,340,932 | 102,942,652 | 139,442,520 | 523,617,185 | 32.03 |
| 6 | 12 | 14 | 69,397,710 | 353,415,098 | 703,400,450 | 1,227,017,635 | 75.05 |
| 7 | 12 | 12 | 13,736,339 | 184,155,437 | 170,419,098 | | |

Table B.5: H2. Number of skipped $k$-mers. Experiment 1.

| i | k | d | $g'(i)$: number of unique $dk$-mers | $g(i)$: total number of $dk$-mers | $p(i)$: skipped $k$-mers | $P(i)$: accumulated skipped $k$-mers | $P(i)/N$ (%) |
|---|---|---|---|---|---|---|---|
| 1 | 12 | 64 | 29,939,425 | 33,289,534 | 17,465,919 | 152,684,377 | 9.34 |
| 2 | 12 | 54 | 42,778,322 | 44,210,681 | 20,535,333 | 173,219,710 | 10.59 |
| 3 | 12 | 44 | 61,469,230 | 68,135,253 | 104,137,617 | 277,357,327 | 16.96 |
| 4 | 12 | 34 | 77,471,870 | 86,902,468 | 107,770,109 | 385,127,436 | 23.56 |
| 5 | 12 | 24 | 89,413,646 | 103,020,694 | 139,510,165 | 524,637,601 | 32.09 |
| 6 | 12 | 14 | 69,430,201 | 353,707,028 | 704,031,946 | 1,228,669,547 | 75.15 |
| 7 | 12 | 12 | 13,741,195 | 184,253,114 | 170,511,919 | | |

Table B.6: Dataset R10. Number of $k$-mers saved to processed varying $d_1$, for $step = 10$. Experiment 2.

| i | k | d | $g'(i)$: number of unique $dk$-mers | $g(i)$: total number of $dk$-mers | $p(i)$: skipped $k$-mers | $P(i)$: accumulated skipped $k$-mers | $P(i)/N$ (%) |
|---|---|---|---|---|---|---|---|
| $d_1 = 52$ | | | | | | | |
| 1 | 15 | 52 | 6,916,996 | 11,446,784 | 137,177,430 | 137,177,430 | 38.66 |
| 2 | 15 | 42 | 9,988,660 | 10,607,351 | 7,199,626 | 144,377,056 | 40.69 |
| 3 | 15 | 32 | 15,182,519 | 16,849,394 | 20,413,430 | 164,790,486 | 46.44 |
| 4 | 15 | 22 | 27,574,544 | 33,630,930 | 34,387,970 | 199,178,456 | 56.13 |
| 5 | 15 | 15 | 59,101,897 | 155,671,848 | 96,569,951 | | |
| $d_1 = 55$ | | | | | | | |
| 1 | 15 | 55 | 6,892,657 | 11,446,784 | 136,005,687 | 136,005,687 | 38.33 |
| 2 | 15 | 45 | 9,951,063 | 10,597,598 | 7,566,401 | 143,572,088 | 40.46 |
| 3 | 15 | 35 | 12,174,757 | 13,603,573 | 19,447,985 | 163,020,073 | 45.94 |
| 4 | 15 | 25 | 16,663,215 | 18,545,908 | 19,065,148 | 182,085,221 | 51.31 |
| 5 | 15 | 15 | 59,101,897 | 172,765,083 | 113,663,186 | | |
| $d_1 = 58$ | | | | | | | |
| 1 | 15 | 58 | 6,873,150 | 11,446,784 | 134,439,838 | 134,439,838 | 37.89 |
| 2 | 15 | 48 | 9,918,351 | 10,592,141 | 7,936,076 | 142,375,914 | 40.12 |
| 3 | 15 | 38 | 12,524,874 | 13,587,418 | 11,524,072 | 153,899,986 | 43.37 |
| 4 | 15 | 28 | 17,490,108 | 19,283,912 | 19,260,440 | 173,160,426 | 48.80 |
| 5 | 15 | 18 | 36,752,813 | 52,666,202 | 58,136,274 | 231,296,700 | 65.18 |
| 6 | 15 | 15 | 59,101,897 | 123,553,604 | 64,451,707 | | |
| $d_1 = 61$ | | | | | | | |
| 1 | 15 | 61 | 6,856,642 | 11,446,784 | 132,506,466 | 132,506,466 | 37.34 |
| 2 | 15 | 51 | 9,889,146 | 10,590,836 | 8,334,716 | 140,841,182 | 39.69 |
| 3 | 15 | 41 | 12,474,151 | 13,574,532 | 11,981,837 | 152,823,019 | 43.07 |
| 4 | 15 | 31 | 14,881,465 | 16,102,006 | 12,720,337 | 165,543,356 | 46.65 |
| 5 | 15 | 21 | 29,291,765 | 36,439,672 | 41,791,971 | 207,335,327 | 58.43 |
| 6 | 15 | 15 | 59,101,897 | 147,514,977 | 88,413,080 | | |
| $d_1 = 64$ | | | | | | | |
| 1 | 15 | 64 | 6,837,966 | 11,446,784 | 130,365,620 | 130,365,620 | 36.74 |
| 2 | 15 | 54 | 9,856,355 | 10,586,100 | 8,736,280 | 139,101,900 | 39.20 |
| 3 | 15 | 44 | 12,414,972 | 13,556,528 | 12,484,480 | 151,586,380 | 42.72 |
| 4 | 15 | 34 | 14,779,373 | 16,061,699 | 13,522,740 | 165,109,120 | 46.53 |
| 5 | 15 | 24 | 17,768,889 | 21,445,984 | 35,789,046 | 200,898,166 | 56.61 |
| 6 | 15 | 15 | 59,101,897 | 153,952,138 | 94,850,241 | | |

Table B.7: Dataset H1. Number of $k$-mers saved to processed for constant $d_1$, varying *step*. Experiment 3.

| i | k | d | $g'(i)$: number of unique $dk$-mers | $g(i)$: total number of $dk$-mers | $p(i)$: skipped $k$-mers | $P(i)$: accumulated skipped $k$-mers | $P(i)/N$ (%) |
|---|---|---|---|---|---|---|---|
| | | | | $step = 15$ | | | |
| 1 | 11 | 64 | 29,904,698 | 33,268,422 | 154,397,787 | 154,397,787 | 9.34 |
| 2 | 11 | 49 | 43,097,386 | 44,372,297 | 25,606,013 | 180,003,800 | 10.89 |
| 3 | 11 | 34 | 63,563,605 | 70,285,182 | 123,791,109 | 303,794,909 | 18.38 |
| 4 | 11 | 19 | 105,203,153 | 149,484,225 | 352,617,234 | 656,412,143 | 39.71 |
| 5 | 11 | 11 | 4,044,294 | 772,309,218 | 768,264,924 | | |
| | | | | $step = 10$ | | | |
| 1 | 11 | 64 | 29,904,698 | 33,268,422 | 154,397,787 | 154,397,787 | 9.34 |
| 2 | 11 | 54 | 42,620,314 | 44,160,106 | 21,764,874 | 176,162,661 | 10.66 |
| 3 | 11 | 44 | 59,461,773 | 67,798,834 | 109,338,519 | 285,501,180 | 17.27 |
| 4 | 11 | 34 | 75,122,632 | 84,799,938 | 110,444,028 | 395,945,208 | 23.95 |
| 5 | 11 | 24 | 86,794,358 | 100,614,882 | 142,503,116 | 538,448,324 | 32.57 |
| 6 | 11 | 14 | 72,512,202 | 264,031,178 | 633,243,453 | 1171691777 | 70.88 |
| 7 | 11 | 11 | 4,044,294 | 257,029,584 | 252,985,290 | | |
| | | | | $step = 8$ | | | |
| 1 | 11 | 64 | 29,904,698 | 33,268,422 | 154,397,787 | 154,397,787 | 9.34 |
| 2 | 11 | 56 | 42,386,145 | 44,046,407 | 18,871,493 | 173,269,280 | 10.48 |
| 3 | 11 | 48 | 52,862,548 | 56,753,112 | 36,016,801 | 209,286,081 | 12.66 |
| 4 | 11 | 40 | 70,047,472 | 78,771,992 | 117,503,492 | 326,789,573 | 19.77 |
| 5 | 11 | 32 | 83,062,123 | 95,159,125 | 108,565,675 | 435,355,248 | 26.33 |
| 6 | 11 | 24 | 92,690,341 | 108,345,522 | 131,921,925 | 567,277,173 | 34.31 |
| 7 | 11 | 16 | 82,209,160 | 205,605,696 | 429,990,719 | 997,267,892 | 60.32 |
| 8 | 11 | 11 | 4,044,294 | 431,453,469 | 427,409,175 | | |
| | | | | $step = 5$ | | | |
| 1 | 11 | 64 | 29,904,698 | 33,268,422 | 154,397,787 | 154,397,787 | 9.34 |
| 2 | 11 | 59 | 41,627,120 | 43,916,396 | 15,751,159 | 170,148,946 | 10.29 |
| 3 | 11 | 54 | 50,930,181 | 55,780,883 | 27,962,264 | 198,111,210 | 11.98 |
| 4 | 11 | 49 | 58,563,142 | 65,196,799 | 36,506,193 | 234,617,403 | 14.19 |
| 5 | 11 | 44 | 70,961,904 | 83,638,585 | 110,739,595 | 345,356,998 | 20.89 |
| 6 | 11 | 39 | 79,893,175 | 95,353,060 | 89,449,175 | 434,806,173 | 26.30 |
| 7 | 11 | 34 | 86,568,205 | 104,681,902 | 100,915,153 | 535,721,326 | 32.41 |
| 8 | 11 | 29 | 91,486,025 | 111,480,035 | 108,581,045 | 644,302,371 | 38.97 |
| 9 | 11 | 24 | 95,040,778 | 116,306,280 | 113,110,868 | 757,413,239 | 45.82 |
| 10 | 11 | 19 | 97,421,806 | 119,482,690 | 114,121,784 | 871,535,023 | 52.72 |
| 11 | 11 | 14 | 65,512,073 | 207,789,082 | 317,337,824 | 1,188,872,847 | 71.91 |
| 12 | 11 | 11 | 4,044,294 | 239,848,514 | 235,804,220 | | |