University of Padua

Department of
Information Engineering

Master's degree in Ingegneria Informatica

Methods for compressing k-mers set with counters

Author

Enrico Rossignolo

Supervisor

Prof. Matteo Comin

Academic Year 2022/2023 – April 3rd

A Davide,
che è mancato a 17 anni mentre scrivevo questa tesi.

Acknowledgements

I'm so sorry for non-Italian speaking people but I need to write this section in my native language to better explain how I feel and to make it easier to read.

Sono diverse le persone che vorrei ringraziare al termine di questo percorso di studi. Per cominciare, vorrei ringraziare mia mamma che mi ha sostenuto e ha avuto fiducia in me. Poi vorrei ringraziare mio papà per i suoi incoraggiamenti e i miei fratelli che in qualche modo mi sono stati vicini. Ringrazio tutti i miei amici, in particolare i gruppi dei "Giorgi" e "Acqua, Luce e Caffè" per avermi fatto ricordare che non esiste solo lo studio e avermi ridato energia con la loro compagnia. Vorrei ringraziare anche il prof. Matteo Comin per la sua incredibile disponibilità e Marco B. per aver reso disponibile il suo codice e aver confermato il bug di NumPy che mi ha portato via tanto tempo. Infine vorrei ringraziare don Matteo per essermi sempre stato vicino anche spiritualmente e per aver creato quel gruppetto di terza e quarta superiore che è esempio di amicizia per tutti.

Abstract

With this work we want to find an efficient way to compress k-mers sets with counters since they take up a lot of disk space but their use brings several advantages over genomes or sets of genomes. Here some strategies are proposed to explore the cdBGs in order to produce smaller files than UST and the counts encoding has been revised. A new application has been presented to implement the above strategies and fix a bug in UST which caused wrong counts ordering. It has been shown that it is possible to improve the compression with respect to UST based on the density of the graph. Finally, a small value of k leads to denser graphs and therefore better results.

Con questo lavoro si vuole trovare un modo efficiente per comprimere i k-mers set con contatori poiché occupano molto spazio su disco ma il loro uso porta diversi vantaggi rispetto ai genomi o insiemi di genomi. Qui vengono proposte alcune strategie per esplorare i cdBG in modo da produrre file più piccoli rispetto a UST e si è rivista la codifica dei conteggi. È stata presentata una nuova applicazione per implementare le suddette strategie e correggere un bug di UST che sbagliava l'ordinamento dei conteggi. Si è dimostrato che è possibile migliorare la compressione rispetto a UST in base alla densità del grafo. Infine, un valore di k piccolo porta a grafi più densi e quindi risultati migliori.

Contents

1 | Introduction

1.1 DNA

The *DNA* (DeoxyriboNucleic Acid) is a macro-molecule that is found in the nucleus of a cell. It has a characteristic double helix shape formed by two chains of nucleotides. A nucleotide is formed by a sugar-phosphate molecule bound to a nitrogenous base. We have four possible bases: adenine, guanine, thymine and cytosine (A, C, T, G). They can make only two hydrogen bonds $A - T$ or $C - G$ between the two chains, see figure 1.1.

The particular order in which the bases are arranged defines the information stored in the DNA. Part of this information is transcribed by the *RNA-polymerase* to the mRNA (Messenger RiboNucleic Acid), that is a single-stranded copy[1] of a portion of the DNA called gene (figure 1.2). Then it is sent out of the nucleus to the ribosome, a special protein that builds other proteins using amino-acids in the cytoplasm following instructions given by the mRNA (figure 1.3).

Proteins make cells that build tissues that make organs and all forms of life.

[1]In this copy, thymine is replaced by uracil.

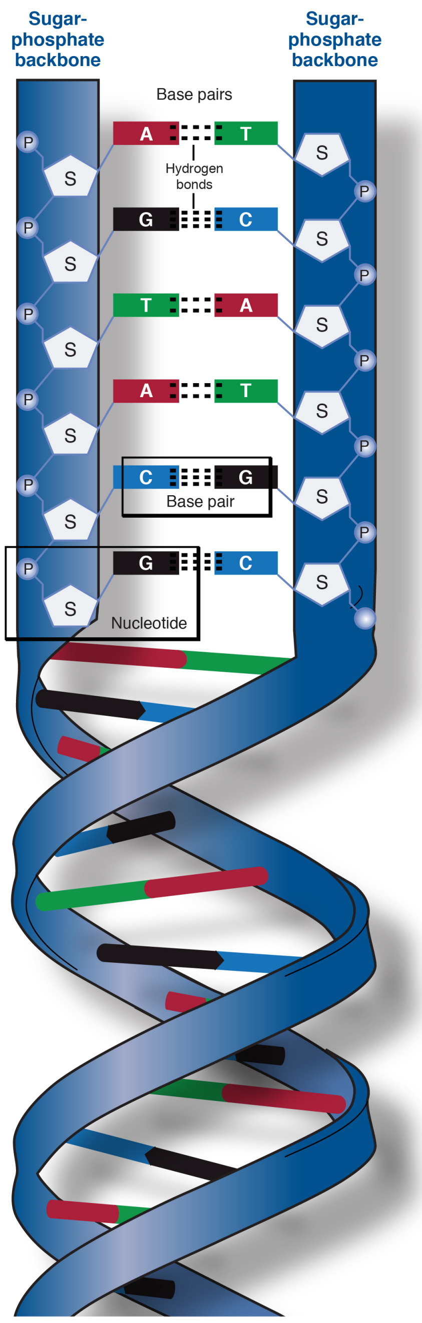


Figure 1.1: *DNA double helix structure. One strand is made by a chains of nucleotides; each nucleotide has a bond with its complementary in the other strand [12].*

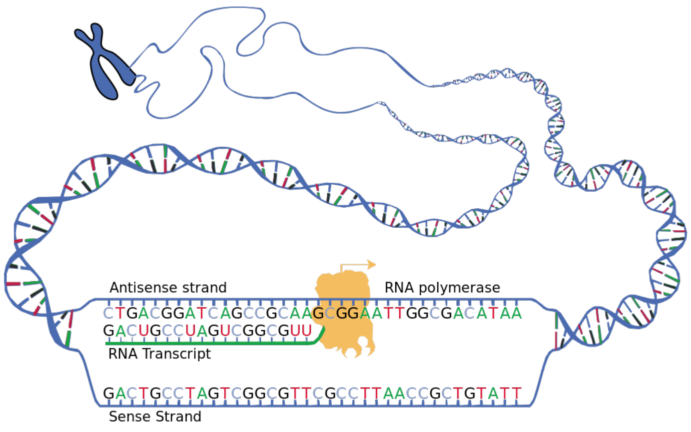


Figure 1.2: *Gene transcription by RNA-polymerase [9].*

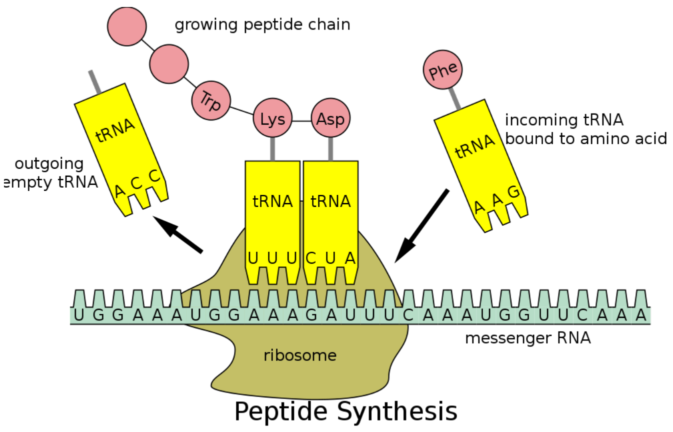


Figure 1.3: *Protein synthesis starts by assembling a small chain of amino-acids called peptide [11].*

1.2 DNA sequencing

Being able to read the DNA can be helpful in order to understand how gene are expressed, how mutations interact with the cell and how living being are related to each other. Therefore DNA sequencing is a key technology in many areas of biology, medicine, anthropology and forensics too.

We call genome all genetic material of a living being, that's what we aim to read.

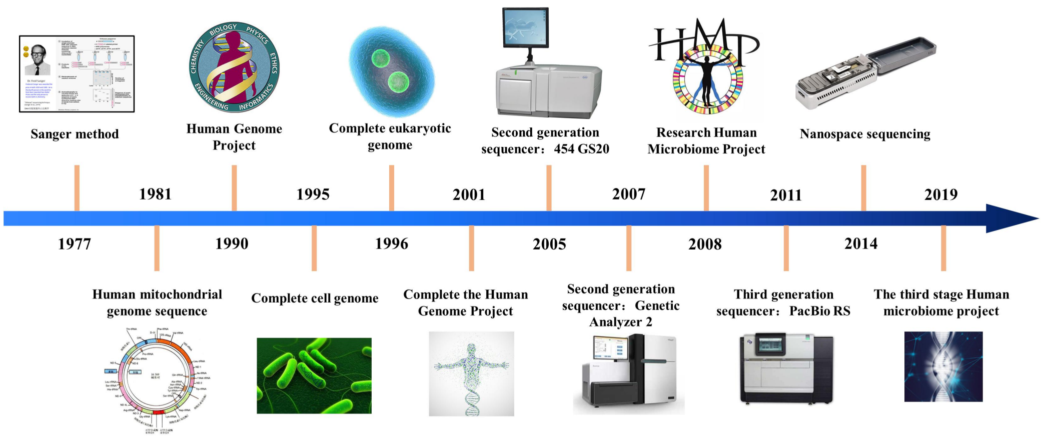

Sanger method
Human Genome Project
Complete eukaryotic genome
Second generation sequencer: 454 GS20
Research Human Microbiome Project
Nanospace sequencing
1981
1995
2001
2007
2011
2019
1977
1990
1996
2005
2008
2014
Human mitochondrial genome sequence
Complete cell genome
Complete the Human Genome Project
Second generation sequencer: Genetic Analyzer 2
Third generation sequencer: PacBio RS
The third stage Human microbiome project

Figure 1.4: *The history of sequencing technology [10].*

The Sanger method [28] paved the way to sequencing technology in 1977 (figure 1.4). In 2005 Next Generation Sequencers (NGS) were born: they are characterized by the ability of reading an high number of sample in short time thanks to task parallelization. There are four main steps in order to read the DNA:

1. *cut*: cut the sample into small pieces of fixed length;
2. *copy*: replicate them many times;
3. *read*: fragments - called reads - are read base by base by sensing light (figure 1.5) emitted from reactions with enzymes. The information obtained is saved in a *FASTQ* file with a quality score string for each read (listing 1.1);

4. *assemble*: finally reads are assembled in a single (or not) DNA sequence. There are two type of assembly:
    - *comparative assembly*: needs a reference genome in which reads can be mapped. It works well even if there are errors;
    - *de novo assembly*: needs high coverage[2] and it's computationally harder than the previous.

```
@SRR001665.1 071112_SLXA-EAS1_s_4:1:1:672:654 length=36
GCTACGGAATAAAACCAGGAACAACAGACCCAGCAC
+SRR001665.1 071112_SLXA-EAS1_s_4:1:1:672:654 length=36
IIIIIIIIIIIIIIIIIIIIIIIIIEII9IIIEIIII
@SRR001665.2 071112_SLXA-EAS1_s_4:1:1:657:649 length=36
GCAGAAAATGGGAGTGAAAATCTCCGATGAGCAGCT
+SRR001665.2 071112_SLXA-EAS1_s_4:1:1:657:649 length=36
IIIIIIIIIIIIIIIIIIIIIIIIII8II=II;III
@SRR001665.3 071112_SLXA-EAS1_s_4:1:1:708:653 length=36
GAGAGAGCAGTGGGCGAGGTTGGGACATGTCATGAT
```

Listing 1.1: *FASTQ file for sequence SRR001665_1. The format implies blocks formed by a definition line starting with* `@`*, the sequence itself, a separator line starting with* `+` *and ASCII quality scores from* `!` *(lowest) to* `~` *(highest).*

With the technology improvement we get lower and lower cost per base pair leading to a huge growth in DNA sequence data (figure 1.6). In figure 1.7, the National Health Institute compares the cost per genome with the Moore's Law: there is a huge demand of storage that actual technology struggles to satisfy.

[2] It's the number of overlapping sequences we have during alignment.

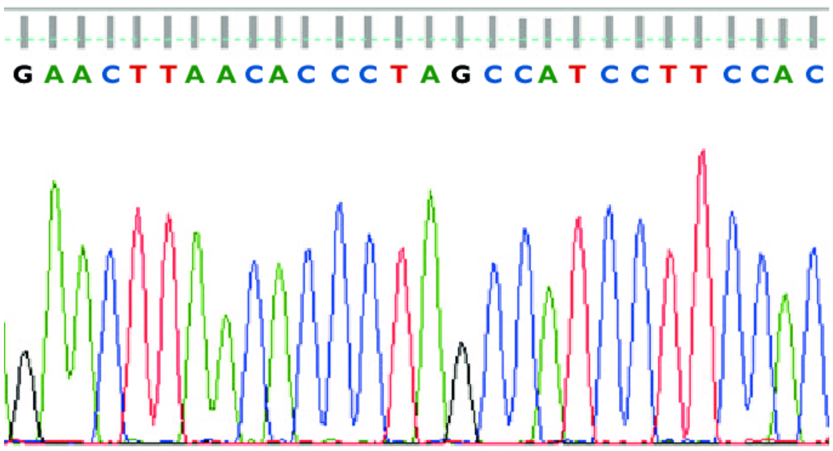


Figure 1.5: *An example of chromatogram used in a Sanger sequencer machine [8].*

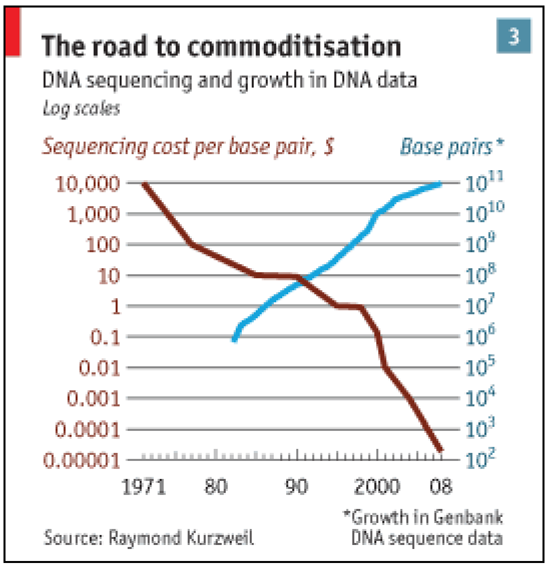


Figure 1.6: *The cost drop is followed by a growth in DNA sequence data. From The Economist.*

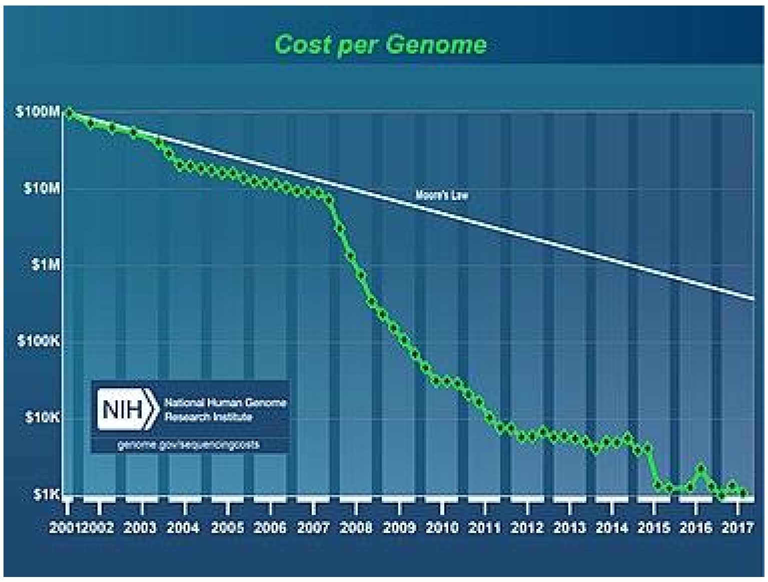


Figure 1.7: *The cost per genome has become much lower than Moore's Law: the exponential growth of genomic data is not followed by an exponential growth of storage.*

1.3 K-mers

In the bioinformatics literature a *k-mer* is a sequence of k bases represented with the characters *A, C, T, G* (Adenine, Guanine, Thymine, Cytosine).

The majority of bioinformatics analysis are performed by k-mer based tools. These tools operate primarily by transforming the input sequence data, which may be of various lengths depending on the technology, into a k-mer set. For example (see listings 1.2 and 1.3), an input sequence can contain many reads of length 36 and its 31-mers set contains only the substrings of length 31 with their multiplicity, called *counts*. With sufficiently large k, we can guess that, if a k-mer count is low, it is likely to be a read error while, if it is high, it is a repeat[3].

```
>SRR001665.1 071112_SLXA-EAS1_s_4:1:1:672:654 length=36
GCTACGGAATAAAACCAGGAACAACAGACCCAGCAC
>SRR001665.2 071112_SLXA-EAS1_s_4:1:1:657:649 length=36
GCAGAAAATGGGAGTGAAAATCTCCGATGAGCAGCT
>SRR001665.3 071112_SLXA-EAS1_s_4:1:1:708:653 length=36
GAGAGAGCAGTGGGCGAGGTTGGGACATGTCATGAT
>SRR001665.4 071112_SLXA-EAS1_s_4:1:1:675:644 length=36
GAACATTTATTATAATCCTATTCAATTATAATAATC
>SRR001665.5 071112_SLXA-EAS1_s_4:1:1:721:668 length=36
GCTGTAGATCTGGAAATCGCAACGGAGGAAGAAAGA
```

Listing 1.2: *First ten lines of the sequence SRR001665_1 in FASTA format. FASTA files have no quality scores.*

```
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA 3208
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAC 32
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAT 3
AAAAAAAAAAAAAAAAAAAAAAAAAAAAACA 28
AAAAAAAAAAAAAAAAAAAAAAAAAAAAACC 2
AAAAAAAAAAAAAAAAAAAAAAAAAAAAATA 3
AAAAAAAAAAAAAAAAAAAAAAAAAAAACAA 27
AAAAAAAAAAAAAAAAAAAAAAAAAAAACAC 3
AAAAAAAAAAAAAAAAAAAAAAAAAAAACCA 2
```

[3] A *repeat* is a k-mer that is repeated more than one time in the genome.

```
AAAAAAAAAAAAAAAAAAAAAAAAAAAATAA 1
```

Listing 1.3: *First ten canonical 31-mers of SRR001665_1 with their multiplicity, sorted in lexicographic order. Canonicalization and sorting are necessary to prove in practice that two k-mer sets are equal.*

K-mer based methods are used in a wide range of applications, including genome assembly[1], metagenomics[4][34], genotyping[5][15, 32], variant calling[6][31], phylogenomics[7][21] and database searching[17, 29, 30, 33, 16, 3, 2, 22, 14, 20].

Although they have been around for some time, k-mer based methods have only recently begun to be applied to terabyte-sized datasets. For example, the dataset used to test the BIGSI database search index, which is made up of 31-mer of 450,000 microbial genomes, takes approximately 12 TB to be stored in compressed form[25].

Reducing the size of such large k-mer set is desirable because it takes non-negligible time to be created and takes a lot of space to be stored (therefore time to be moved across the internet).

In practice, many people encode their k-mers in a text file and then compress it with a file compressor, like *gzip*[26]. Is there a more space-efficient way to store a k-mer set?

1.3.1 Advantages of using k-mers

Using a k-mers set can lead to important advantages with respect to using a genome or a set of genomes and they're not limited to computing resources and time spent on alignment.

[4]Metagenomics is the study of genetic material recovered directly from environmental sample. This imply that there is more than one genome to consider.

[5]Genotyping analyzes variations in genomes between individual organisms.

[6]Variant calling is the process that identifies single nucleotide polymorphism and small insertion and deletion in an individual genome.

[7]Phylogenomics is the analysis that involves genome data and evolutionary reconstructions.

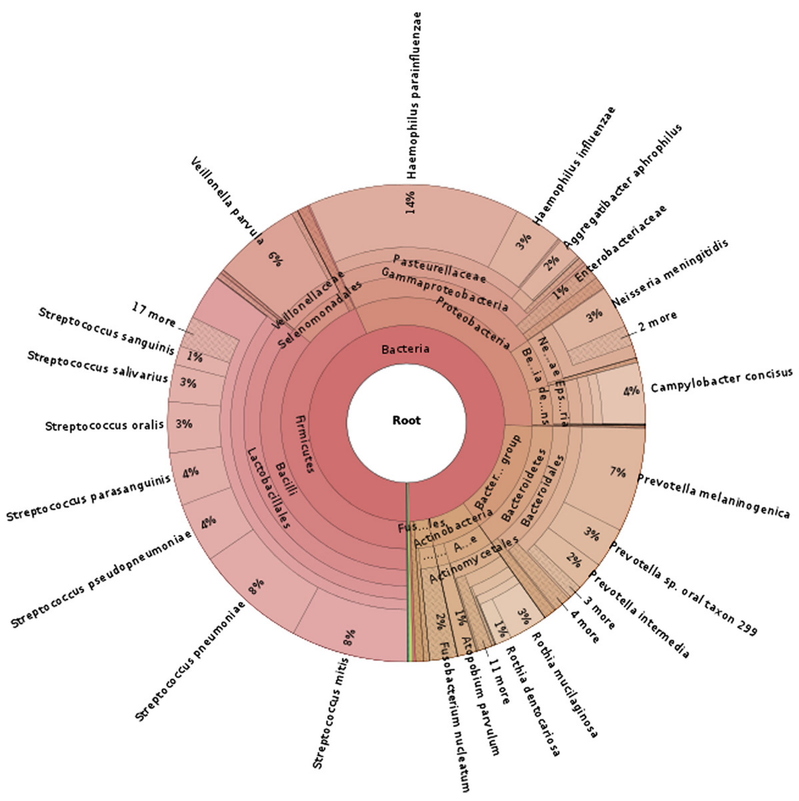
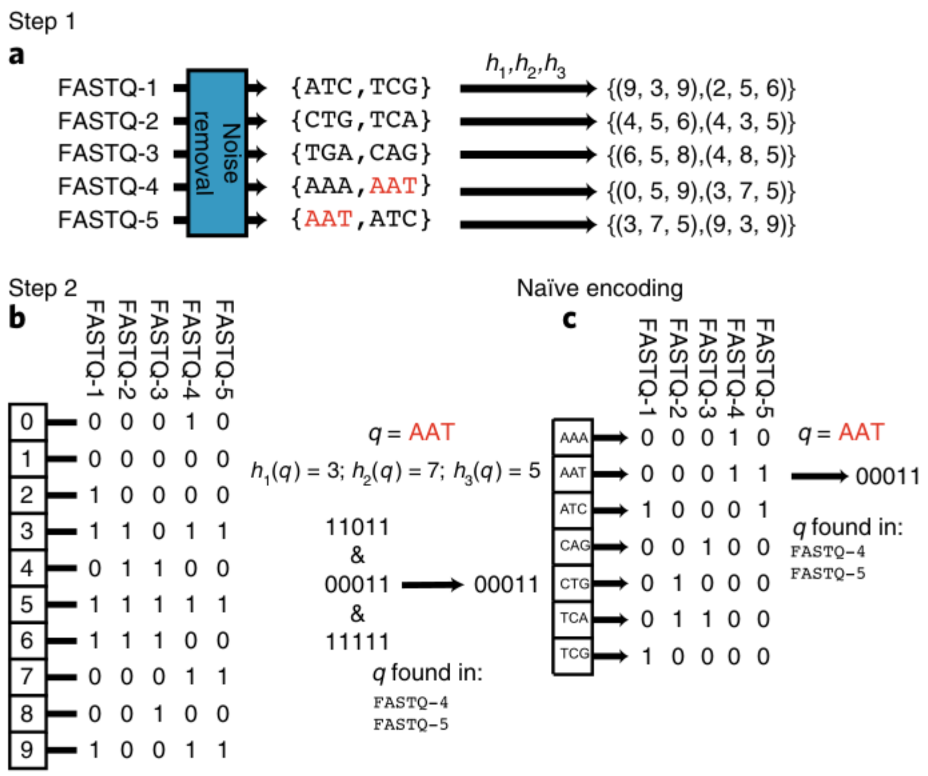


Figure 1.8: *Taxonomic distribution of saliva microbiome reads classified by Kraken[34].*

For example, assembling a metagenome can be really hard because of read errors and contamination. Kraken[34] is a k-mers based tool that assigns taxonomic labels (figure 1.8) to short DNA sequences. This is over 900 times faster than the state of the art MegaBLAST; the only downside is the negligible lower accuracy.

The aforementioned BIGSI[3] (BItsliced Genomic Signature Index) database can be used to search through the huge number of sequences stored on public archives like SRA (Sequence Read Archive) or ENA (European Nucleotide Archive). Compared to the BLAST database, it is able to easily scale (by adding columns to the matrix in figure 1.9b) and does not require assembled genomes as input which are fundamentally lossy[8] unlike k-mers sets.

[8]An assembled genome is lossy because sequencing errors need to be corrected or removed. Furthermore we can have more than one possible alignments.



	FASTQ-1	FASTQ-2	FASTQ-3	FASTQ-4	FASTQ-5
0	0	0	0	1	0
1	0	0	0	0	0
2	1	0	0	0	0
3	1	1	0	1	1
4	0	1	1	0	0
5	1	1	1	1	1
6	1	1	1	0	0
7	0	0	0	1	1
8	0	0	1	0	0
9	1	0	0	1	1

	FASTQ-1	FASTQ-2	FASTQ-3	FASTQ-4	FASTQ-5
AAA	0	0	0	1	0
AAT	0	0	0	1	1
ATC	1	0	0	0	1
CAG	0	0	1	0	0
CTG	0	1	0	0	0
TCA	0	1	1	0	0
TCG	1	0	0	0	0

Figure 1.9: *BIGSI use of k-mers compared to naive encoding. a) Datasets are converted to k-mers sets, optionally denoised and hashed with a set of hashing functions. b) Datasets are stored as bits columns in which there is 1 on positions given by the hash functions, known as Bloom filters. The query q is also hashed, selecting specific rows. The positions of* 1*s on the resulting* $\wedge$ *give the datasets in which q is found.*

2 | K-mers representation

In this chapter we will see some terminology, in particular the definition of a de Bruijn graph in order to understand how we can explore a k-mer set, some known methods for compressing a k-mers set and two problems found by the author.

2.1 Terminology

As already stated, a **k-mer** (from ancient Greek *méros*, "part") is a substring of a sequence of nucleobases. That means that the string alphabet is

$$\Sigma = \{A, C, T, G\}$$

and a k-mer $m \in \Sigma^k$.

We call **reverse complement** $rc(\cdot)$ of a k-mer its reverse sequence in which each nucleobase is replaced with its complement, that is: $A \mapsto T$, $C \mapsto G$, $T \mapsto A$, $G \mapsto C$.

We will consider one k-mer and its reverse-complement as the same since we consider both DNA strands.

We call **canonical** a k-mer that is the lexicographically[1] smaller between itself and its reverse complement.

Given a string $t = \langle t_1, \ldots, t_l \rangle$, we define **i-prefix** as the first i character of t

[1]Some tools, like the k-mer counter DSK, use the order $A < C < T < G$ instead [27].

and **i-suffix** as the last $l - i$ characters of t:

$$pref_i(t) = \langle t_1, \dots, t_i \rangle$$

$$suff_i(t) = \langle t_i, \dots, t_l \rangle$$

Given a k-mers set $K = \{m_1, \dots m_{|K|}\}$, a **de Bruijn graph** (figure 2.1) is a directed graph $dBG(K) = (V, A)$ in which:

- $V = \{v_1, \dots, v_{|K|}\}$
- each node $v \in V$ has a label $lab(v_i) = m_i$
- each node $v \in V$ has two different sides $s_v \in \{0, 1\}$
- a node side (v, s_v) is spelled as

$$spell(v, s_v) = \begin{cases} lab(v) & s_v = 0 \\ rc(lab(v)) & s_v = 1 \end{cases} \tag{2.1}$$

- there is an arc between two node sides (v, s_v) and (u, s_u) if and only if their spellings share a $(k-1)$-mer, in particular it must be

$$suff_{l-(k-1)}\left(spell\left(v, 1 - s_v\right)\right) = pref_{k-1}\left(spell\left(u, s_u\right)\right)$$

A **path** $p = \langle (v_{i_1}, s_{i_1}), \dots, (v_{i_l}, s_{i_l}) \rangle$ in dBG(K) is a sequence of node sides in which there exist an arc between a node and its successor and both sides of internal nodes are used (see figure 2.2).

The **spelling** of a path is the spelling of each node side "glued" together:

$$spell(p) = spell(v_{i_1}, s_{i_1}) \cdot suff_k(spell(v_{i_2}, s_{i_2})) \cdot \dots \cdot suff_k(spell(v_{i_l}, s_{i_l}))$$

A dBG(K) can be **compacted** (noted as cdBG(K), see figure 2.3) by considering non-branching paths as single nodes. A cdBG(K) consumes less memory than non-compacted dBG(K).

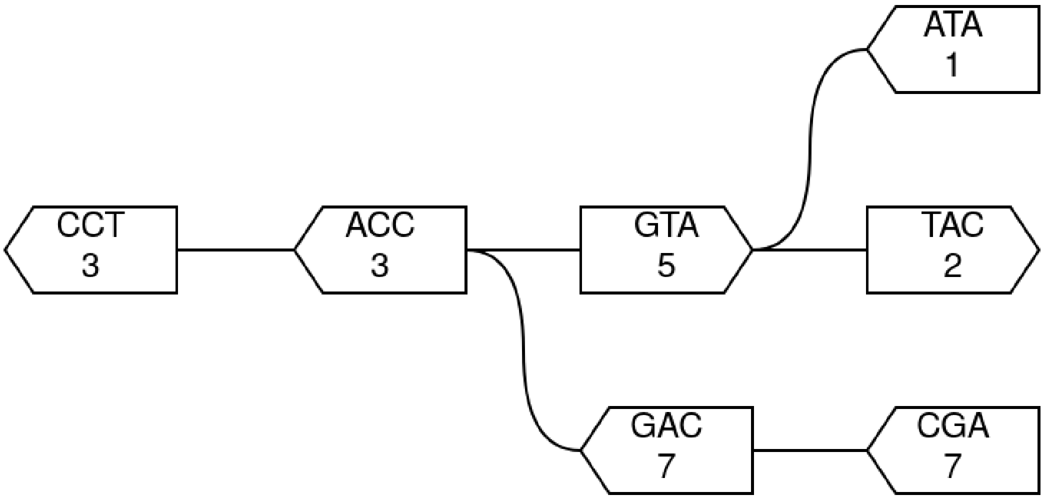
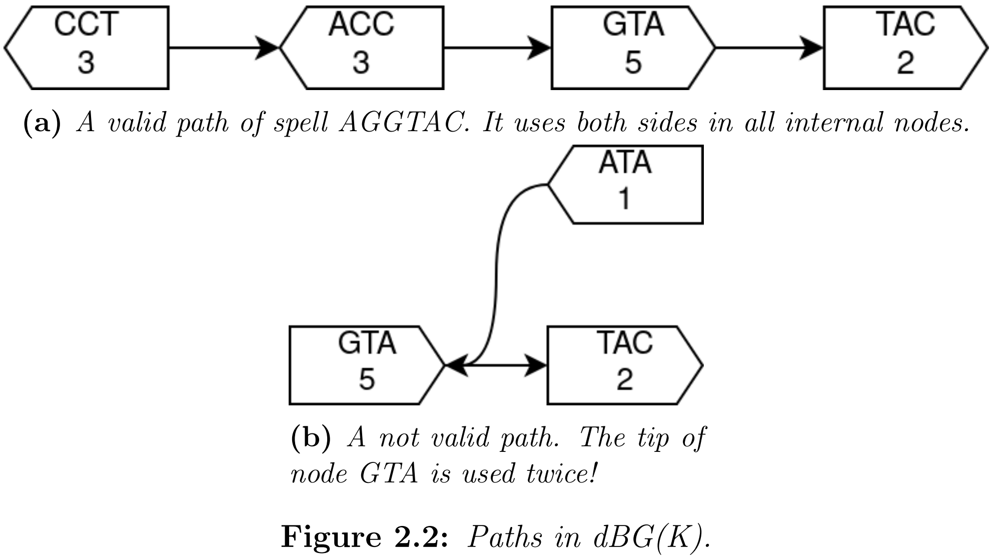


Figure 2.1: *An example of dBG. Double directed arcs are merged into undirected edges. Nodes are labelled with 3-mers and their multiplicity. Sides 1 are pointy.*



(a) *A valid path of spell AGGTAC. It uses both sides in all internal nodes.*

(b) *A not valid path. The tip of node GTA is used twice!*

Figure 2.2: *Paths in dBG(K).*

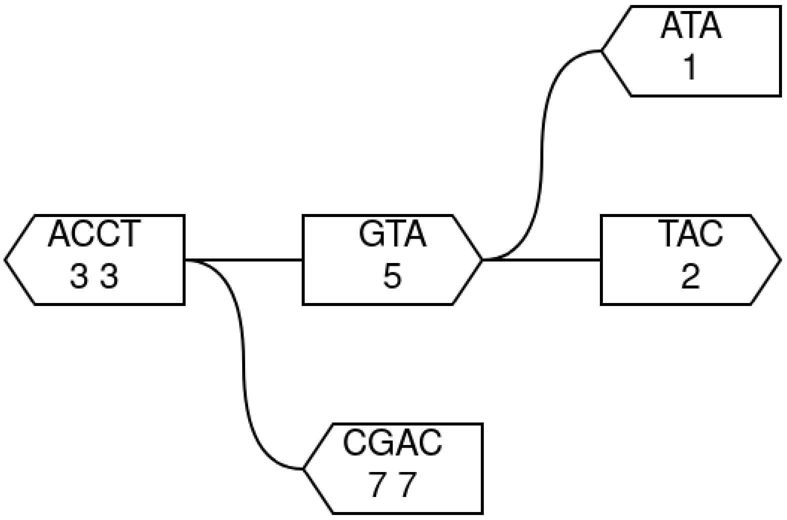


Figure 2.3: *A compacted dBG. Non-branching paths are merged into single nodes and labelled with their spellings and concatenated counts.*

The spelling of any path is called **contig**. The spelling of a non-branching path is called **unitig**.

Spellings of a vertex-disjoint path cover of dBG(K) are called **simplitigs**[4].

Given a string set S, we define its **weight**[2] as the sum of all string lengths:

$$weight(S) = \sum_{s \in S} |s|$$

2.2 K-mers set compression

In principle, we can glue k-mers together in order to gain space without loss of information: in this way we can save $k-1$ characters per gluing. Counts can be added progressively in a separate list.

For example, given the following 3-mer set K

```
AAA 4
AAT 2
CTG 3
```

[2]Also known as Cumulative Length (CL)

```
TCT 2
```

Listing 2.1: *This k-mer set has weight equal to* 12.

we can glue the first two 3-mers and the second two, obtaining the following sequences

```
AAAT
TCTG
```

Listing 2.2: *This string set has weight* 8.

and counts

```
4
2
2
3
```

Since we can derive the original 3-mers from these sequences, we can say that they **represents** K.

Observe that the sequences above are contigs in dBG(K).

Among the tools that are analyzed in [5] there are Squeakr, KMC, DSK, BCALM2, ProphAsm, ESS and UST. The first three are *k-mer counters*, i.e. programs that extract and count unique k-mers given a FASTA/FASTQ file. BCALM2 internally use DSK to extract k-mers and then creates a cdBG. The last three are tools that explore the cdBG and glue unitigs. Since ProphAsm and ESS do not consider counts, they are not taken into account in this work. UST gives the best compression ratio, therefore we will see the latter more in detail.

2.2.1 Unitig Stitch (UST)

The authors of UST, Rahman and Medvedev, claim that, given a string set S representing a k-mer set K, in order to minimize $weigth(S)$, we need to minimize the number of sequences that it contains, in particular it holds

$$weight(S) = |K| + (k-1) \cdot |S| \tag{2.2}$$

Furthermore, they claimed that there exist a non-tight lower bound on 2.2 that can be computed using cdBG(K) topology (see [26] for details).

In order to find a small string set made up of contigs, they wanted to solve the vertex-disjoint path cover problem on cdBG(K) but, since it is known to be NP-Hard[3] for general graphs, they designed a greedy algorithm (named UST) focusing more into speed than optimality.

However, based on their datasets of 31-mers, UST worked well having a gap with the lower bound within 3%. What about different datasets? What about different k-mer lengths?

They used *BCALM2* [6] in order to compute all the unique k-mers and counts from the dataset and to create cdBG(K). Their algorithm can be abstracted with algorithm 1: it arbitrarily chooses a node, called seed, it extends the seed in one direction until there is no unvisited node and then restart. When there is no seed, it try to merge paths. Finally, it extracts spellings and counts from paths.

Algorithm 1 UST

```
Require: $G = (V, A) = \text{dBG}(K)$; $lab(\cdot)$; $count(\cdot)$
Ensure: $S$ represents $K$
 1: $P \leftarrow \varnothing$
 2: while $V \neq \varnothing$ do
 3:     extract $v$ from $V$                                  ▷ choose a seed
 4:     $p \leftarrow \langle (v, 0) \rangle$
 5:     while $p$ has a successor $(u, s_u)$ not visited do       ▷ extend p
 6:         $p \leftarrow p \cdot (u, s_u)$                          ▷ forward
 7:     end while
 8:     $P \leftarrow P \cup \{p\}$
 9: end while
10: while $\exists p_1, p_2 \in P \mid p_1$ is linked to $p_2$ do          ▷ merge paths
11:     replace $p_1$ and $p_2$ with $p_1^{rev} \cdot p_2$
12: end while
13: $S \leftarrow \{spell(p) \mid p \in P\}$
```

[3]Actually, we don't know if the restriction of the vertex-disjoint path cover problem to dBGs can make the problem polynomial.

What if we choose the seeds and the successors more wisely?

Contigs and counts are saved on different files and then compressed separately with MFCompress[24] and LZMA respectively.

2.2.2 Counts encoding

Another author[5] claims that we can improve the compression ratio of counts by choosing the right encoding. In particular, he obtained an improvement from 55% to 60% with respect to the original counts file sizes.

The technique consists on ordering contigs and counts based on contigs average counts and then applying a modified run length encoding (RLE) in which 1-runs are implicit. We call RLE* this modified RLE. For example, given the cdBG in figure 2.3, running UST we get

```
>
ATACCT
>
TAC
>
CGAC
```

```
1
5
3
3
2
7
7
```

Then sorting counts vectors per average:

```
2
1
5
3
3
7
7
```

and applying RLE* it becomes

```
2
1
5
3-2
7-2
```

The ordering step can potentially help on nearing equal counts that will be compacted with RLE.

2.2.3 Problems found

Wrong counts

The author found that UST does not list counts correctly. It can be easily shown with a counterexample[4].

Consider the following input (visually, cdBG in figure 2.4)

```
>0 LN:i:4 ab:Z:2 3 L:-:1:+ L:-:2:-
ACCT
>1 LN:i:3 ab:Z:5 L:-:0:+
GTA
>2 LN:i:3 ab:Z:7 L:+:0:+
GAC
```

Listing 2.3: *UST input as cdBG(K) with BCALM2 headers that encode counts and unitigs links.*

This outputs the following files

```
>
AGGTA
>
GAC
```

Listing 2.4: *UST contigs output.*

[4]Aside from this ad-hoc counterexample, it is proved evenly wrong in real sequences using a k-mer counter.

```
2
3
5
7
```

Listing 2.5: *UST counts output.*

From there we can derive the k-mer set

```
AGG 2
GGT 3
GTA 5
GAC 7
```

Listing 2.6: *Derived k-mer set.*

and, by applying the reverse-complement to the first two k-mers, we obtain

```
CCT 2
ACC 3
GTA 5
GAC 7
```

Listing 2.7: *Final k-mer set.*

where clearly the first two counts should be swapped.

This happens because UST does not reverse the counts vector when it takes a node with side 1, entering the node from the tip.

Fixing this behaviour may make counts more compressible.

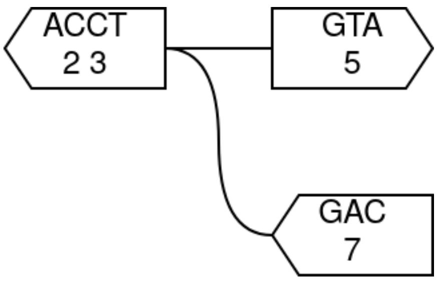


Figure 2.4: *This cdBG forces UST to output wrong counts. It arbitrarily takes the first node AACT, it computes the reverse-complement, AGGT, and it glues it with GTA. But it does not reverse the counts associated to the first node, leading to incorrect results!*

Bugged encoding

There is another problem regarding how the encoding was applied in [5]. The idea was implemented in Python using a NumPy function to read the counts file. The function, named `loadtxt()`, had a known bug[13] that changes numbers under certain conditions. In particular, it reads chunks of 50000 strings and it establishes the maximum string length on the first chunk. The bug arises when following chunks contain strings longer than the maximum string length previously found.

Therefore, it can happen that, when reading numbers like $10, 11, \ldots, 20, 21, \ldots, 30, 31, \ldots$, it reads instead $1, 1, \ldots, 2, 2, \ldots, 3, 3, \ldots$; then counts clearly becomes very compressible using RLE*.

This idea looks promising but needs to be tested with a correct implementation.

3 | Methods

After founding the problem described on subsection 2.2.3, UST was completely rewritten and changed its name to *USTAR*. It can be found on the author repository `https://github.com/enricorox/USTAR`.

3.1 The pipeline

We are going to stick to the following pipeline (figure 3.1): we build cdBG(K) with BCALM2, we run USTAR choosing the strategy to pick the seed and its successors, we sort in some way counts and sequences, we apply RLE* to the counts and finally we compress sequences with *MFCompress* and counts with *bzip3*[1].

The algorithm 2 differs a bit since it extends the seed forward and backward instead of merging paths later as in algorithm 1. This is more similar to what ProphAsm does and may help find longer contigs. In the next section we will see how we can explore the cdBG.

[1]We use bzip3 instead of LZMA as in [26, 5] because it provides better compression ratios.

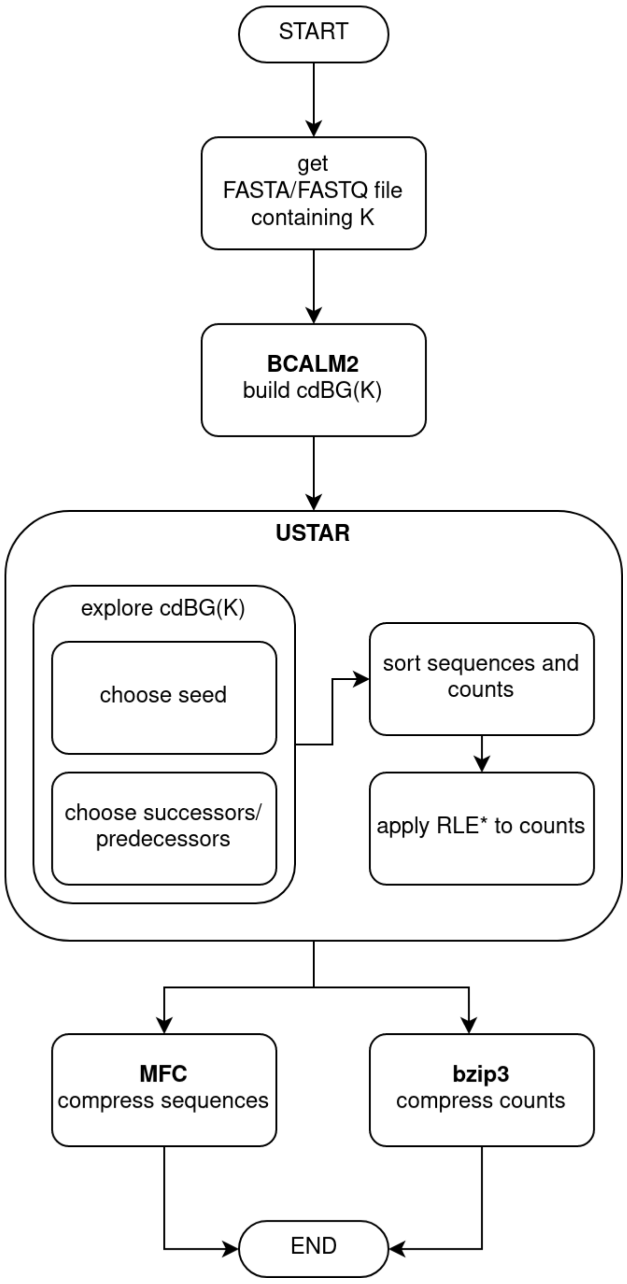


Figure 3.1: *K-mers set compression using USTAR.*

Algorithm 2 USTAR

```
Require: $G = (V, A) = \mathrm{dBG}(K)$; $lab(\cdot)$; $count(\cdot)$
Ensure: $S$ represents $K$
 1: $P \leftarrow \varnothing$
 2: while $V \neq \varnothing$ do
 3:     extract $v$ from $V$                                  ▷ wisely choose a seed
 4:     $p \leftarrow \langle (v, 0) \rangle$
 5:     while $p$ has a successor $(u, s_u)$ not visited do   ▷ wisely extend p
 6:         $p \leftarrow p \cdot (u, s_u)$                   ▷ forward
 7:     end while
 8:     while $p$ has a predecessor $(u, s_u)$ not visited do ▷ wisely extend p
 9:         $p \leftarrow (u, 1 - s_u) \cdot p$               ▷ backward
10:     end while
11:     $P \leftarrow P \cup \{p\}$
12: end while
13: $S \leftarrow \{spell(p) \mid p \in P\}$
```

3.2 Strategies

3.2.1 Exploring cdBG

There are different strategies to choose the seed and to extend the path.

Starting with choosing the seed, we can choose the unitig that is

- more (or less) long
- more (or less) connected
- with the higher (or lower) average (or median) abundance

When choosing successors (or predecessors), we can do exactly the same things as before plus choosing the unitig u with the nearest abundance computed as

- shorter distance between averages (or medians) vectors elements

code	**description**
`s`	choose the Seed
`x`	eXtend the path
`+`	more/higher
`-`	less/lower
`=`	closer
`f`	choose the First node available
`c`	Connections
`l`	unitig Length
`a`	Abundance
`aa`	Average Abundance
`ma`	Median Abundance

Table 3.1: *Mnemonic codes used to create methods short names.*

- or shorter distance between first or last vectors elements, based on nodes sides:

d	**(v, 0)**	**(v, 1)**
(u, 0)	$\mid counts(v).first - counts(u).first \mid$	$\mid counts(v).last - counts(u).first \mid$
(u, 1)	$\mid counts(v).first - counts(u).last \mid$	$\mid counts(v).last - counts(u).last \mid$

Strategies on table 3.2 have mnemonic short names and use codes on table 3.1.

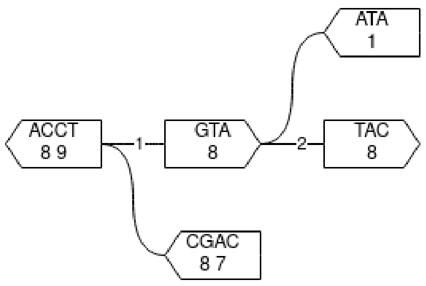
We will see that there are two strategies that work better than the others:

- `s+aa x=a`: choose the seed with highest average abundance and its successors with nearest average abundance. With this method we aim to obtain consecutive counts as similar as possible in order to full exploit RLE* later (see figure 3.2a).

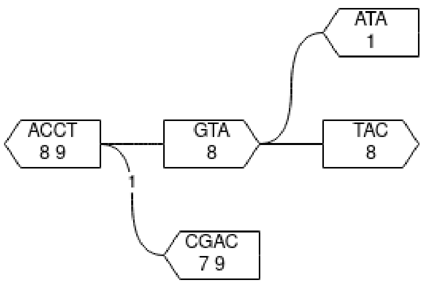
- `s+aa x-c`: choose the seed with highest average abundance and its successors with less connections. Here we aim to optimize counts as before, starting ordering seeds by average abundance, but we want to avoid congested nodes leaving them as a last resort (see figure 3.2b).

methods	description
`sf xf`	choose the first available node as seed and extend the path with the first available neighbour
`s-ma x=a`	choose the node with lower median abundance as seed and extend the path with the node with the nearest k-mer abundance
`s-aa x=a`	choose the node with lower average abundance as seed and extend the path with the node with the nearest k-mer abundance
`s+l x-l`	choose the node with higher unitig length as seed and extend the path with the node with the lower unitig length
`s-l x+l`	choose the node with lower unitig length as seed and extend the path with the node with the higher unitig length
`s+aa x=ma`	choose the node with higher average abundance as seed and extend the path with the node with the nearest median abundance
`s-ma x-ma`	choose the node with lower median abundance as seed and extend the path with the node with the lower median abundance
`s+aa x=a`	choose the node with higher average abundance as seed and extend the path with the node with the nearest k-mer abundance
`s+aa x-c`	choose the node with higher average abundance as seed and extend the path with the node with less connection

Table 3.2: *Methods used to explore cdBGs. All methods have their counterpart with counts encoding.*



(a) *Method `s+aa x=a`.*



(b) *Method `s+aa x-c`.*

Figure 3.2: *Methods for exploring dBGs. a) It starts from ACCT, then choose GTA ($|8-8| < |8-7|$) and TAC ($|8-8| < |8-1|$), the unitigs with nearest abundance. b) It start from ACCT, then choose GCAC that has less connections.*

3.2.2 Counts encoding

We need to sort count vectors (and then sequences!) in order to fully exploit RLE*. A good idea is, as done in [5], sorting vectors by average so that similar-average vectors, that likely have similar counts, are close to each other. We can do better: instead of computing the whole vector average, we can compute the average of first and last element that are the ones that are adjacent to other vectors. Another step is to flip vectors and reverse-complement sequences if we see that there are equal extremes that can be adjacent.

Let's explain it with an example. Consider the cdBG(K) in figure 3.2a and apply `s+aa x=a`. We get

```
>
AGGTAC
>
CGAC
>
ATA
```

and

```
9
8
8
8

7
8

1
```

Listing 3.1: *Blank lines between contigs are added for clarity.*

They are already sorted by average; now we can flip the second contig in order to bring the eights closer:

```
>
AGGTAC
```

```
>
GTCG
>
ATA
```

and

```
9
8
8
8

8
7

1
```

Listing 3.2: *Blank lines between contigs are added for clarity.*

Finally we can compact eights with RLE* saving $2 \cdot 3 - 2 = 4$ characters (new lines included):

```
9
8-4
7
1
```

This encoding is called `avg_flip_rle`.

3.3 Validation

At the end of each USTAR run, the output files are validated, using a k-mers counter like in [25], to be sure that there is no implementation errors but, unlike [25], we also need to verify counts. Thus, the validation requires, for each k-mer extracted from simplitigs, to: compute their canonical and append the counts on the same line, sort the lines and compare them to the ones generated from a well known k-mers counter like Jellyfish[19] (see algorithm 3).

Algorithm 3 Output validation.

```
Require: dataset
Ensure: USTAR output is correct
 1: cdBG ← *output of BCALM given the dataset*
 2: simplitigs, counts ← *output of USTAR given cdBG*
 3: line ← ∅
 4: extracted_kmers ← *new file*
 5: while line ≠ EOF do
 6:     line ← *read simplitigs line*
 7:     for i ← 1; i = |line| - k + 1; i++ do
 8:         kmer = line[i:i+k-1]            ▷ extract a substring of length k
 9:         count = *next count from counts file*
10:         *write canonical(kmer) and count to extracted_kmers*
11:     end for
12: end while
13: extracted_kmers ← *sort extracted_kmers*
14: kmers ← *outputs of Jellyfish given the dataset*
15: kmers ← *sorted kmers*
16: if kmers ≠ extracted_kmers then
17:     return FAILURE
18: end if
```

4 | Experimental setup

In this chapter we will see the experiment setup as the tools used, the chosen datasets, the metrics we considered and the machine that did the computations.

4.1 Tools

All the tools used are summarized in table 4.1. Let's take an example about how to use them for the sequence `SRR001665`. We need to: download it from the NCBI's server[1], convert it to FASTA format splitting if it is paired-end[2], feed it to BCALM2 that output the cdBG, run UST/USTAR and finally compress the sequences and counts. See listings 4.1, 4.2 and 4.3 for the specific commands.

```
$ prefetch --progress SRR001665
$ fasterq-dump --progress --threads 8 --fasta \
    --split-files SRR001665
$ bcalm -max-memory 16000 -nb-cores 8 -kmer-size 31 \
    -abundance-min 2 -in SRR001665_1.fasta \
    -all-abundance-counts -out SRR001665_1.a2.k31
```

Listing 4.1: *Download, convert and build cdBG(K). SRR001665_1 is the first strand of the sequence SRR001665.*

[1]The National Center for Biotechnology Information provides public access to biomedical and genomic information as the SRA archive.

[2]A FASTA/FASTQ file is paired-end if it contains labelled sequences from both DNA strands.

name	description	version	multi-thread
prefetch	download SRA files from NCBI	v3.0.1	no
fasterq-dump	convert SRA files to FASTA/FASTQ	v3.0.1	yes
BCALM2	build cdBG in low memory	v2.2.3	yes
UST	compress k-mers sets	v1.0	only unitigs sorting
USTAR	compress k-mers sets	v1.0	no
MFCompress	compress FASTA files	v1.01	yes
bzip3	file compressor	v1.2.1	yes

Table 4.1: *Summary of the tools used.*

```
$ ust -k 31 -i SRR001665_1.a2.k31.unitigs.fa -a 1
$ MFCompressC -t 8 -3 -o \
    SRR001665_1.a2.k31.unitigs.fa.ust.fa.mfc \
    SRR001665_1.a2.k31.unitigs.fa.ust.fa
$ bzip3 SRR001665_1.a2.k31.unitigs.fa.ust.counts
```

Listing 4.2: *Run UST and compress output files.*

```
$ ustar -k 31 -i SRR001665_1.a2.k31.unitigs.fa -s+aa -x=a \
    -e avg_flip_rle
$ MFCompressC -t 8 -3 -o SRR001665_1.a2.k31.ustar.fa.mfc \
    SRR001665_1.a2.k31.unitigs.fa.ust.fa
$ bzip3 SRR001665_1.a2.k31.ustar.counts
```

Listing 4.3: *Run USTAR and compress output files.*

4.2 Datasets

The datasets consist of sequences downloaded from NCBI's server listed on table 4.2. Since there is a huge variability between sequences, the choice is not easy. However, they are chosen from some papers ([23, 27, 18, 7, 4, 26]) paying attention on sequences variability with respect to read length and number of reads (therefore dataset size). Furthermore, we choose other four metagenomes to see if there are differences between genome and metagenome compression.

Beside the read length and the number of reads, when the k-mer size is fixed and the cdBG(K) (need to fix the k-mer size) is computed, some of

name	description	notes	read length	#reads	size [GB]
SRR001665	Escherichia coli	genome, paired	36	20,816,448	9.304
SRR061958	Human Microbiome 1	metagenome, paired	101	53,588,068	3.007
SRR062379	Human Microbiome 2	metagenome, paired	100	64,491,564	2.348
SRR10260779	Musa balbisiana RNA-Seq	genome, paired	101	44,227,112	2.363
SRR11458718	Soybean RNA-seq	genome, paired	125	83,594,116	3.565
SRR13605073	Broiler chicken DNA	genome	92	14,763,228	0.230
SRR14005143	Foodborne pathogens	genome, paired	211	1,713,786	0.261
SRR332538	Drosophila ananassae	genome, paired	75	18,365,926	0.683
SRR341725	Gut microbiota	metagenome, paired	90	25,479,128	1.254
SRR5853087	Danio rerio RNA-Seq	genome	101	119,482,078	3.194
SRR957915	Human RNA-seq	genome, paired	101	49,459,840	3.671

Table 4.2: *Downloaded sequences in SRA format. In the column "note" it is specified if it is a genome or a metagenome and if it is a collection of paired-end sequences that can be splitted in two files.*

the sequences intrinsic properties are: number of k-mers and nodes (unitigs), number of arcs (links between unitigs), graph density, number of isolated nodes, the GC content[3] and counts variance. In particular, graph density is computed as the number of arcs divided by the maximum number of arcs (8 times the number of nodes, 4 arcs for each side):

$$density = \frac{\#arcs}{8 \cdot \#nodes}$$

Let's fix $k = 31$ and remove all the 31-mers with abundance less than 2 assuming they're errors as the majority of k-mers based tools do. Properties of cdBGs of the above datasets are on table 4.3. We have cdBGs of 54K-7M nodes and 12%-28% density. Counts variance is even more diverse, spanning from 36 to 8M.

We will see that graph density is a key property that significantly changes USTAR improvement over UST. In order to increase it, we will use also k-mers with multiplicity 1 (see table 4.4): as a side effect, it also decreases the number of isolated nodes and counts variance.

[3]The GC content is the percentage of Gs and Cs in a DNA sequence.

name	#31-mers	#unitigs	isolated nodes [%]	density[%]	GC content [%]	counts variance
SRR001665_1	4,842,286	161,403	23.55	18.98	51.03	37.74
SRR001665_2	4,953,108	238,840	22.34	19.27	51.00	36.62
SRR061958_1	75,557,725	3,456,486	24.21	14.67	40.20	6,106.83
SRR061958_2	76,041,929	3,725,295	17.05	23.46	40.62	1,824.95
SRR062379_1	46,108,438	1,604,652	22.12	21.51	41.56	6,972.12
SRR062379_2	45,569,938	1,465,409	24.89	21.57	41.55	3,219.37
SRR10260779_1	52,952,684	1,509,151	16.54	25.09	45.91	92,422.70
SRR10260779_2	53,639,306	1,635,926	16.39	25.56	46.06	88,979.70
SRR11458718_1	72,180,320	2,052,425	14.18	26.99	42.87	8,103,010.00
SRR11458718_2	74,367,059	2,134,438	14.36	27.79	43.28	8,670,680.00
SRR13605073_1	31,639,418	1,222,560	22.53	20.35	53.85	4,260,880.00
SRR14005143_1	5,247,490	54,246	10.51	24.54	51.22	404.285
SRR14005143_2	5,494,520	102,283	12.75	23.42	51.51	356.679
SRR332538_1	5,292,049	261,333	29.39	24.45	48.22	7,478,630.00
SRR332538_2	6,648,069	726,475	19.72	26.73	48.44	4,084,150.00
SRR341725_1	94,897,526	1,526,427	55.31	12.09	43.43	120.059
SRR341725_2	94,264,606	1,514,990	56.18	11.90	43.38	119.054
SRR5853087_1	124,740,993	7,775,719	12.55	28.02	42.09	6,123,300.00
SRR957915_1	63,418,496	3,338,082	12.97	28.25	46.80	82,318.00
SRR957915_2	63,293,138	3,745,996	17.63	26.88	46.86	54,692.70

Table 4.3: *Properties of cdBG(K) where K is the specified dataset with 31-mers with multiplicity* 1 *removed.*

name	#31-mers	#unitigs	isolated nodes [%]	density[%]	GC content [%]	counts variance
SRR001665_1	10,343,472	1,671,602	37.75	15.68	52.52	46.20
SRR001665_2	12,058,109	2,239,012	35.00	16.45	52.49	39.75
SRR061958_1	404,149,685	20,669,968	9.58	26.44	43.97	1,172.84
SRR061958_2	495,804,915	24,188,615	11.44	25.94	44.34	301.76
SRR062379_1	160,692,477	9,429,828	3.64	27.17	45.61	2,079.89
SRR062379_2	159,905,793	8,774,002	5.42	27.31	45.65	998.27
SRR10260779_1	123,624,245	7,700,639	2.93	29.17	46.25	39,770.60
SRR10260779_2	139,633,894	8,839,865	3.08	29.35	46.87	34,348.90
SRR11458718_1	137,995,280	6,523,514	3.94	30.85	43.66	4,239,080.00
SRR11458718_2	150,549,990	7,152,027	3.62	32.37	44.74	4,283,690.00
SRR13605073_1	55,764,573	3,018,512	6.51	26.46	54.87	2,417,560.00
SRR14005143_1	15,005,192	658,011	0.09	31.26	53.32	375.67
SRR14005143_2	31,850,681	1,353,300	0.96	30.77	53.90	170.25
SRR332538_1	11,382,816	687,166	11.01	26.71	48.81	3,478,370.00
SRR332538_2	28,880,136	2,240,104	11.67	29.32	49.41	940,744.00
SRR341725_1	185,618,107	4,511,197	18.17	21.64	44.75	70.67
SRR341725_2	192,133,588	4,891,154	18.39	21.52	44.85	67.62
SRR5853087_1	382,773,071	25,382,771	2.80	32.27	40.67	1,995,640.00
SRR957915_1	239,200,400	17,651,052	2.10	31.45	47.88	21,936.10
SRR957915_2	364,597,018	22,045,191	7.08	29.58	47.94	9,552.82

Table 4.4: *Properties of cdBG(K) where K is the specified dataset with all 31-mers.*

4.3 Metrics

Since we are interested in obtaining a good compression ratio and UST alone reaches good results (even if counts are wrong), we will compute the improvement over UST. In this way we can also understand the error entity that wrong counts have introduced.

Given the output set of sequences S, we will consider the following metrics:

- *weight*: the weight of S as defined in section 2.1;
- $fasta$: the compressed sequence file size;
- *counts*: the compressed counts file size;
- *overall*: the sums of compressed file sizes of sequences and counts.

Therefore improvements are computed as percentage differences so that positive values imply smaller USTAR files:

$$\Delta M = \frac{M_{UST} - M_{USTAR}}{M_{UST}}$$

where M is the metric considered.

In particular we will take care to separate the encoding from the exploring strategies for two main reasons:

- since encoding changes sequence ordering, $fasta$ can vary;
- since encoding changes counts ordering and format, *counts* can vary.

4.4 Machine

These tools are executed on the Department's Blade cluster.

The cluster consists of many nodes equipped with

- *runner-[01-03]* Three nodes with 48 CPUs (4x Intel(R) Xeon(R) Gold 5118 CPU @ 2.30/3.20GHz), 1.5TB RAM
- *runner-[04-06]* Three nodes with 72 CPUs (4x Intel(R) Xeon(R) Gold 5220 CPU @ 2.20/3.90GHz), 2TB RAM, one Nvidia Quadro P2000 GPU
- *runner-[07-09]* Three nodes with 96 CPUs (4x Intel(R) Xeon(R) Gold 6252N CPU @ 2.30/3.60GHz, 3TB RAM
- *gpu-1* 24 CPUs (2x Intel(R) Xeon(R) Gold 5118 CPU @ 2.30/3.20GHz), 1TB RAM, six Nvidia Titan RTX GPUs
- *gpu-[2-3]* Two nodes with 32 CPUs (2x Intel(R) Xeon(R) Gold 5218 CPU @ 2.30/3.90GHz), 1.5TB RAM, eight Nvidia RTX 3090 GPUs

Slurm is used to submit each job and to request resources to the cluster.

We used 16GB of RAM for nearly all cdBGs, but up to 100GB for particularly big ones when k-mers are not filtered.

5 | Results

In this chapter we will see the results presented step by step, as done in the experiments. There are so many data that it is impossible to show them all here: file sizes are in the author repository (`https://github.com/enricorox/USTAR`) while improvement tables are in appendix A.

5.1 Filtered k-mers

First, k-mers was filtered using BCALM2 so that the ones with only one count are removed.

Eighteen different methods (the ones on table 3.2 with and without counts encoding) was tested: the two methods that worked the best[1] overall are `s+aa x=a` (choose the seed with highest average abundance and choose the next node with closest abundance) and `s+aa x-c` (choose the seed as before and choose the next node with less connection), average results are on table 5.1. Surprisingly, counts encoding worked well only for counts compression but it can increase the compressed sequence size because of different ordering or reverse-complementing. It's worth noting that all methods have improved the sequence weight but this does not imply an improved compressed sequence size.

Since from tables A.1,A.2 and A.3 we can observe that improvements vary depending on the dataset, we want to find some properties that lead to better

[1]Here we are counting the number of times a particular method scores the best. Then we take the one with the highest value.

improvements	`s+aa x=a`	`s+aa x=a avg_flip_rle`	`s+aa x-c`
$\Delta weight$ [%]	0.54	0.54	0.54
$\Delta fasta$ [%]	-1.57	-1.89	0.03
$\Delta counts$ [%]	10.13	10.62	5.48
$\Delta overall$ [%]	1.72	1.59	1.47

Table 5.1: *Average improvement over UST, three methods that works the best.*

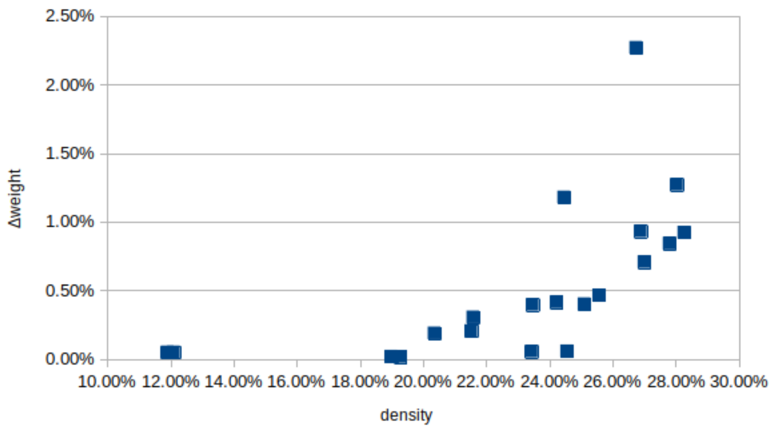


Figure 5.1: *Improvement on weight using method* `s+aa x=a`*. The denser the graph, the better the weight improvement.*

compression, if they exist. We find that, for method `s+aa x=a`, some correlation between graph density and improvements may exists, see figure 5.1. Furthermore, it seems that counts improvement depends on counts variance (figure 5.2).

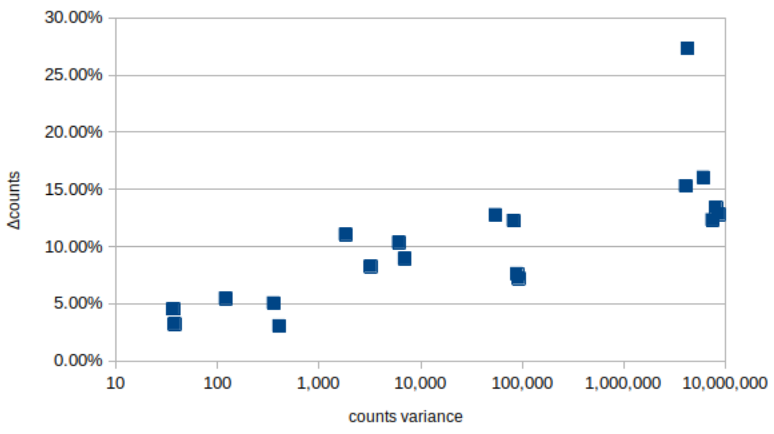


Figure 5.2: *Improvement on counts using method* `s+aa x=a` *with filtered k-mers. Higher variance corresponds to higher counts improvement.*

5.2 Unfiltered k-mers

In order to confirm the density trend, denser graphs must give better results. Therefore in this experiment, in which it was used the previous winning methods, all k-mers are kept. It turns out that counts encoding greatly helped counts compression but also can worsen sequences compression, see table 5.2 and appendix A. Method `s+aa x-c` gave better overall improvement on average. We can see that the link between graph density and weight improvement is even stronger and there is a negative correlation between isolated nodes and weight (figures 5.3 and 5.4).

improvements	`s+aa x=a`	`s+aa x=a avg_flip_rle`	`s+aa x-c`
$\Delta weight$ [%]	0.97	0.97	0.97
$\Delta fasta$ [%]	-4.31	-4.13	0.24
$\Delta counts$ [%]	31.06	32.97	12.70
$\Delta overall$ [%]	1.45	1.92	2.30

Table 5.2: *Average improvement over UST, three winning methods are tested with all k-mers in the datasets.*

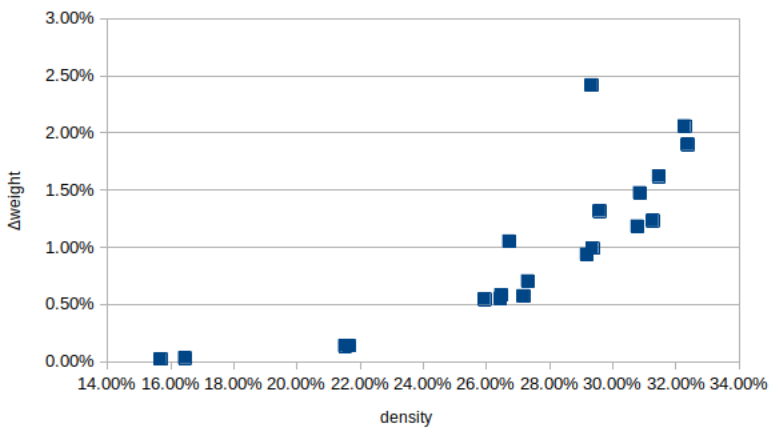


Figure 5.3: *Improvement on weight using `s+aa x-c` with unfiltered k-mers.*

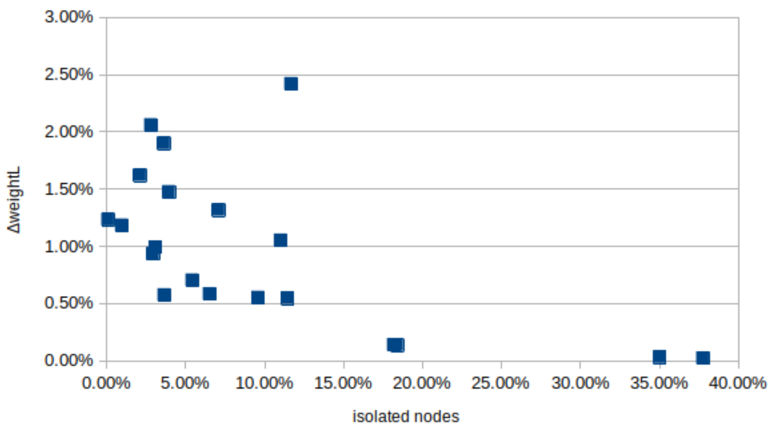


Figure 5.4: *Improvement on weight using* `s+aa x-c` *with unfiltered k-mers.*

5.3 Different k-mer size

Changing k-mer size modifies the number of k-mers and the number of arcs between them, thus changing the density.

The following plots (figures 5.5, 5.6, 5.7, 5.8) describe how metrics changes with respect to density, isolated nodes, counts variance and number of nodes, applying `s+aa x-c`, the best method found, on the unfiltered datasets with $k \in \{15, 21, 31, 41\}$.

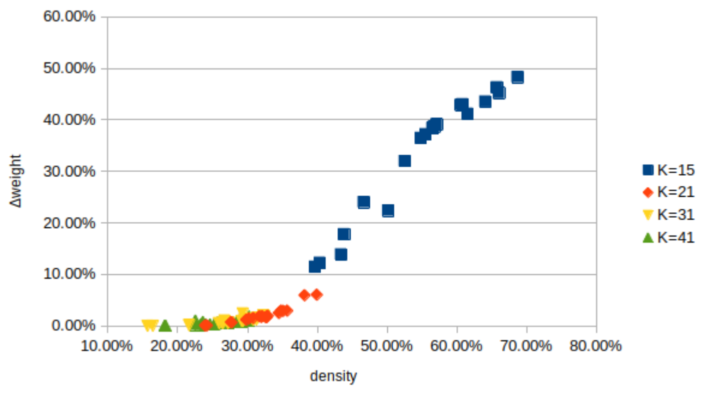
Plot 5.5a shows that improvement on weight definitely depends on graph density and we can guess why: density is a scaled version of the average number of arcs per node. The greater the average number of arcs, the harder is for UST to make a good arbitrary choice letting USTAR to make a good one. With this result, we proved that a good choice is the node with less connections.

Plot 5.6a shows that we get higher improvements on weight when we have less isolated nodes. The reason may be that we can only choose isolated nodes as seeds and not as path extension, thus we have less opportunities to use them in such a way that helps compression.

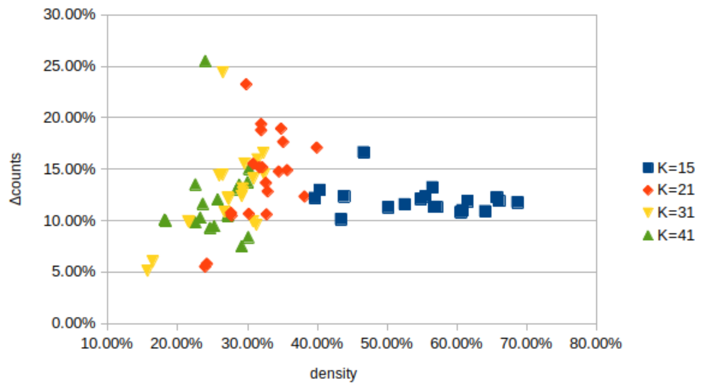
Plots 5.7 show that counts variance does not influence improvements on the metrics considered.

Finally, figure 5.8a shows that the improvements on weight is correlated to the number of unitigs but only for $k = 15$, probably hiding something in an higher dimension that we cannot see in the plot.

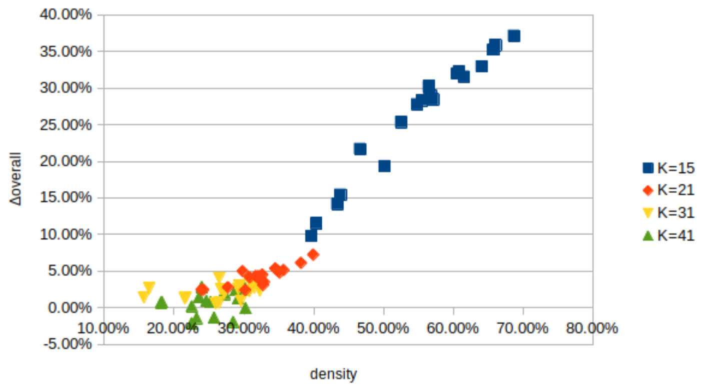
On figure 5.9 we can see that improvements on all metrics except *counts* increase decreasing the k-mer size, while *counts* remains almost constant.



(a) *Weight improvement.*

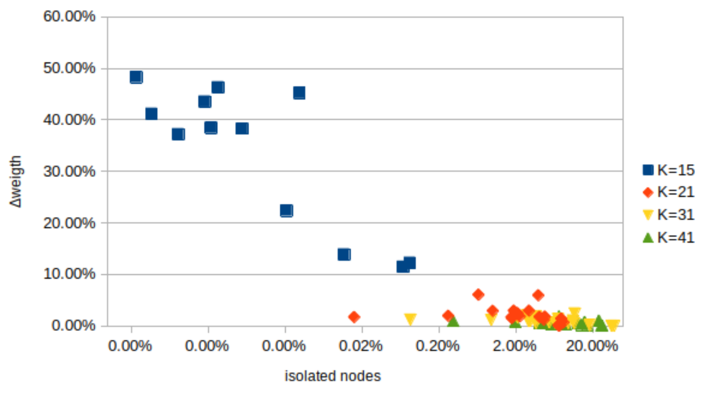


(b) *Counts improvement.*

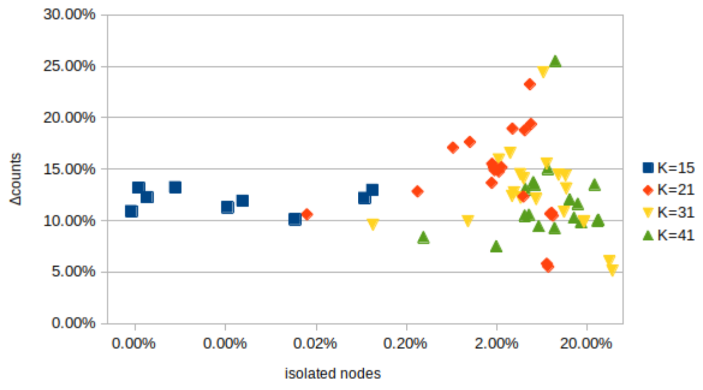


(c) *Overall improvement.*

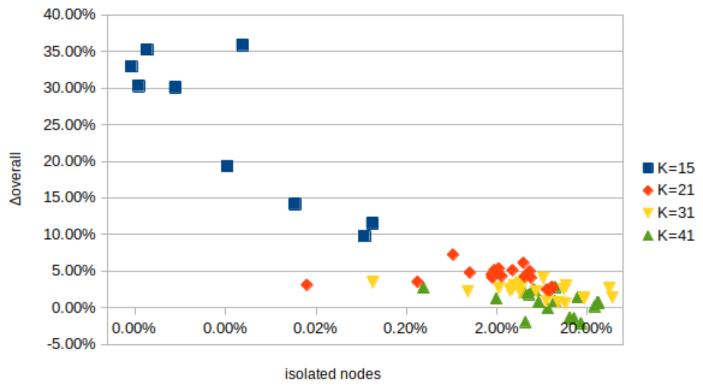
Figure 5.5: *Improvements varying density and using* `s+aa x-c`*. In 5.5a we can see a very definite curve that highlights the dependence on graph density.*



(a) *Weight improvement.*

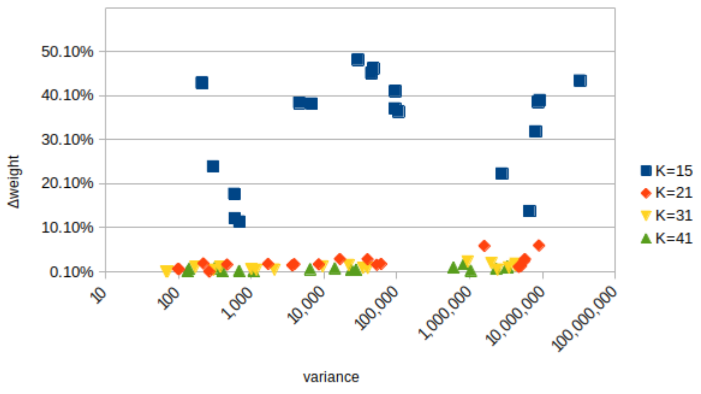


(b) *Counts improvement.*

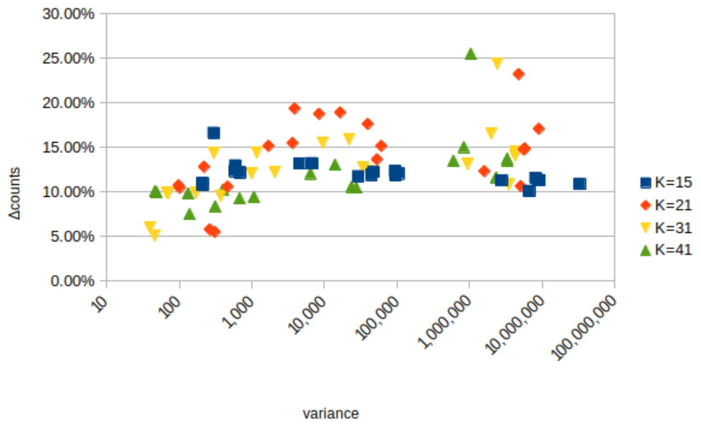


(c) *Overall improvement.*

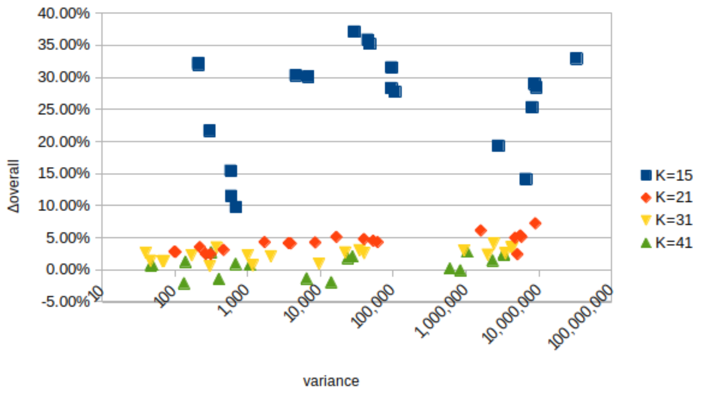
Figure 5.6: *Improvements varying isolated nodes and using* `s+aa x-c`. *It's clear that the fewer isolated nodes, the better.*



(a) *Weight improvement.*

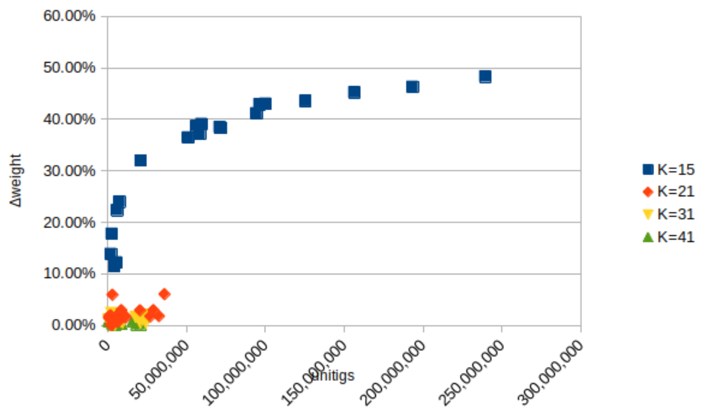


(b) *Counts improvement.*

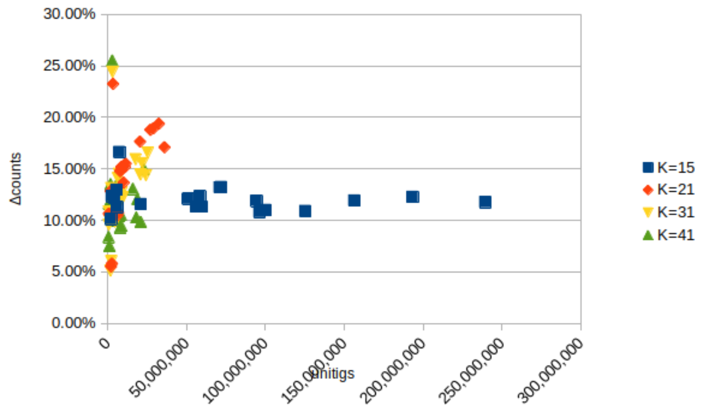


(c) *Overall improvement.*

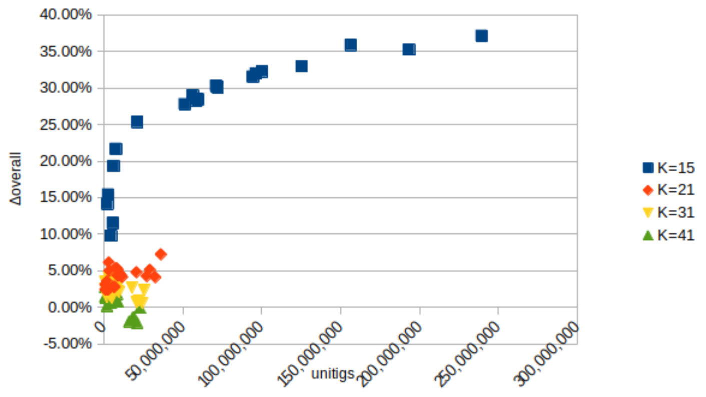
Figure 5.7: *Improvements varying counts variance and using* `s+aa x-c`.



(a) *Weight improvement.*



(b) *Counts improvement.*



(c) *Overall improvement.*

Figure 5.8: *Improvements varying #unitigs and using* `s+aa x-c`*.*

k-mer size	Δweight[%]	Δfasta[%]	Δcounts[%]	Δoverall[%]
15	33.64	31.16	12.07	26.40
21	2.10	1.95	14.17	4.20
31	0.97	0.24	12.70	2.30
41	0.66	-1.23	11.90	0.75

Table 5.3: *Average improvements using* `s+aa x-c`*, unfiltered k-mers.*

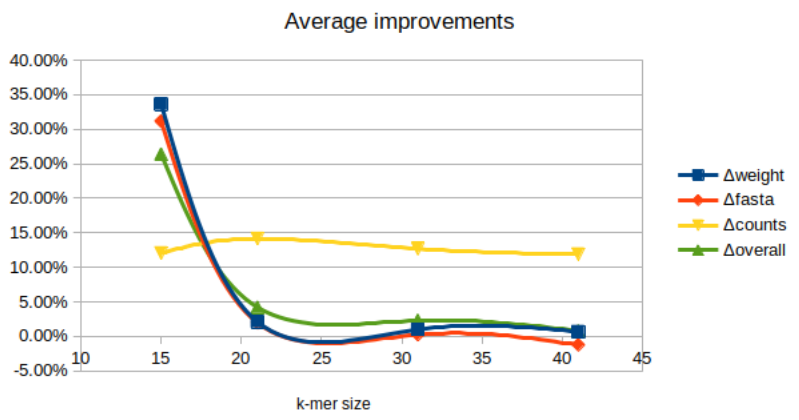


Figure 5.9: *Average improvements using* `s+aa x-c` *with unfiltered k-mers, with $k \in \{15, 21, 31, 41\}$. Since Δcounts is almost constant, Δoverall is driven by Δweight.*

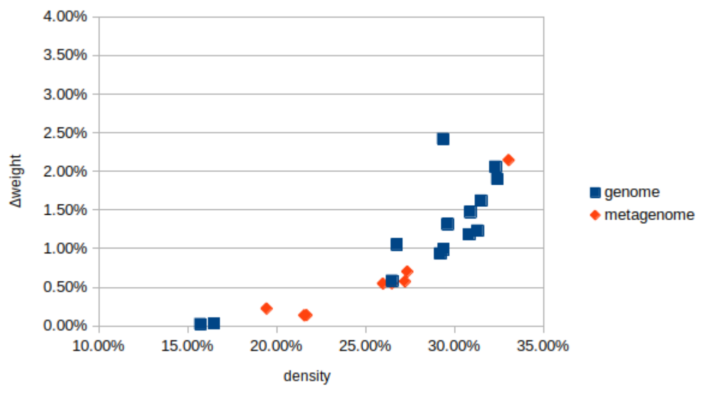
5.4 Metagenomes

Since we got good improvements with cdBGs with a lot of unitigs (figure 5.8), we want to see whether with a big metagenome we achieve denser cdBGs (with $k = 31$) and thus good results. Therefore other four metagenomes were downloaded from the NCBI's server and they are listed on table 5.4.

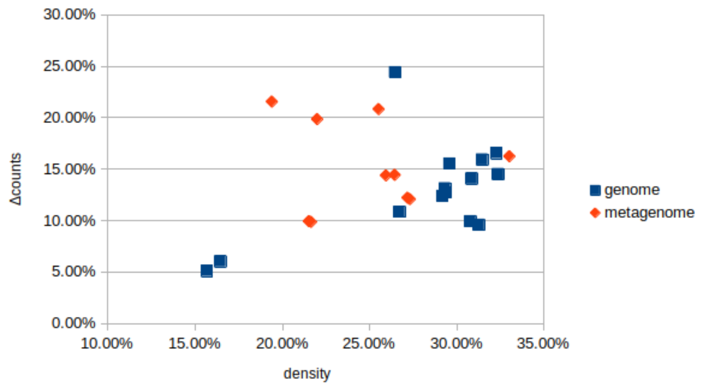
name	**description**	**notes**	**#bases**	**size [GB]**
SRR10849012	Switchgrass phillosphere	metagenome, paired	25G	6.9
SRR14556465	Goat digestive tract	metagenome, paired	6G	1.7
SRR6869040	Marine biofilm	metagenome, paired	26G	9.3
SRR13608728	River microbiome	metagenome, paired	26G	7.8

Table 5.4: *Additional metagenome datasets of greater size.*

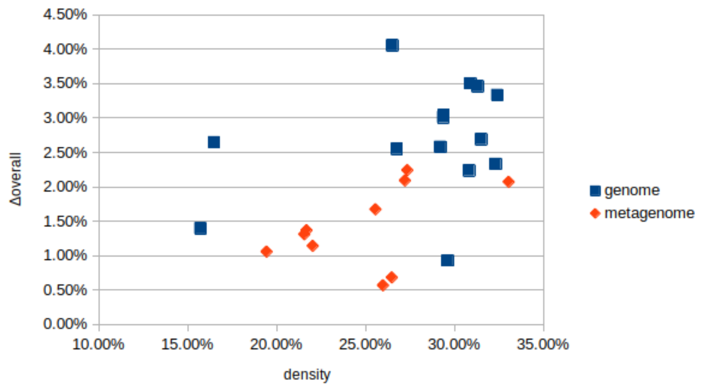
From plots on figures 5.10, we can say that metagenomes do not enjoy special benefits but, on the contrary, they're the ones with the worst overall improvement.



(a) *Weight improvement.*



(b) *Counts improvement.*



(c) *Overall improvement.*

Figure 5.10: *Differences between genomes and metagenomes using* `s+aa x-c`*.*

6 | Conclusions

In order to find a way to save storage for the use of k-mers set and counters, we review UST, a tool that uses compacted de Bruijn graphs to remove redundancies basically gluing k-mers.

We presented some new strategies to explore cdBGs and found wrong counts produced by UST. Then a new tool named USTAR was created to solve the bug and to implement the exploring strategies: we have shown that it is possible to improve UST making smaller sequences and counts file. In particular the denser the graph, the smaller the files; we have a similar trend with the fewer isolated nodes, the better. Counts encoding can be used to get smaller counts files but it can enlarge sequences files leading to suboptimal results. Smaller k-mer sizes lead to denser graphs and then better results.

Specifically, we found that with filtered k-mers sets and $k = 31$ there is an average overall improvement of 1.72% with method `s+aa x=a` and an improvement on counts of 10.13%. Using unfiltered k-mers we can do generally better: the best method becomes `s+aa x-c` with an overall improvement of 2.30% and with counts improvement of 12.70%. If we focus on counts, we can get much better counts compression (32.97%) using `s+aa x=a avg_flip_rle` instead. We got the best compression with $k = 15$ with an overall improvement of 26.40%. With greater k-mer size, the overall improvement drop to 0.75% but counts remains at 11.90%.

Appendices

A | Additional tables

dataset	**Δweight[%]**	**Δfasta[%]**	**Δcounts[%]**	**Δoverall[%]**
SRR001665_1	0.02	-0.04	3.22	1.57
SRR001665_2	0.02	-0.85	4.55	1.62
SRR061958_1	0.42	-3.06	10.35	0.88
SRR061958_2	0.40	-3.86	11.05	0.32
SRR062379_1	0.21	-2.30	8.94	1.07
SRR062379_2	0.31	-2.76	8.27	0.61
SRR10260779_1	0.40	-0.43	7.23	2.17
SRR10260779_2	0.47	-0.36	7.62	2.31
SRR11458718_1	0.71	-1.31	13.42	3.10
SRR11458718_2	0.84	-1.14	12.83	3.12
SRR13605073_1	0.19	-2.25	27.31	3.32
SRR14005143_1	0.06	-0.24	3.06	1.04
SRR14005143_2	0.06	-0.96	5.05	1.25
SRR332538_1	1.18	-0.38	12.32	2.42
SRR332538_2	2.27	-1.83	15.30	1.99
SRR341725_1	0.05	-1.59	5.46	0.12
SRR341725_2	0.05	-0.73	5.49	0.77
SRR5853087_1	1.27	-3.02	16.04	2.35
SRR957915_1	0.92	-2.18	12.25	2.18
SRR957915_2	0.93	-2.02	12.77	2.12
Average	0.54	-1.57	10.13	1.72

Table A.1: *Method `s+aa x=a` applied to filtered k-mers.*

dataset	**Δweight[%]**	**Δfasta[%]**	**Δcounts[%]**	**Δoverall[%]**
SRR001665_1	0.02	-0.67	1.91	0.60
SRR001665_2	0.02	-1.34	2.94	0.62
SRR061958_1	0.42	-3.60	7.41	-0.36
SRR061958_2	0.40	-4.22	8.13	-0.76
SRR062379_1	0.21	-2.66	7.25	0.32
SRR062379_2	0.31	-3.14	6.72	-0.13
SRR10260779_1	0.40	-1.18	4.90	0.89
SRR10260779_2	0.47	-0.96	5.20	1.10
SRR11458718_1	0.71	-1.84	11.67	2.20
SRR11458718_2	0.84	-2.14	11.20	1.92
SRR13605073_1	0.19	-2.05	27.03	3.42
SRR14005143_1	0.06	-0.35	2.95	0.93
SRR14005143_2	0.06	-1.48	4.63	0.77
SRR332538_1	1.18	-0.14	9.09	1.90
SRR332538_2	2.27	-0.20	9.72	2.02
SRR341725_1	0.05	-2.23	5.40	-0.38
SRR341725_2	0.05	-1.31	5.43	0.31
SRR5853087_1	1.27	-2.21	12.08	1.82
SRR957915_1	0.92	-1.78	9.05	1.49
SRR957915_2	0.93	-1.20	9.34	1.75
Average	0.54	-1.73	8.10	1.02

Table A.2: *Method `s+aa x=a avg_flip_rle` applied to filtered k-mers.*

dataset	**Δweight[%]**	**Δfasta[%]**	**Δcounts[%]**	**Δoverall[%]**
SRR001665_1	0.02	0.81	1.32	1.06
SRR001665_2	0.02	-1.16	2.65	0.58
SRR061958_1	0.42	-3.26	5.00	-0.83
SRR061958_2	0.40	-4.02	5.54	-1.34
SRR062379_1	0.21	-2.38	4.57	-0.29
SRR062379_2	0.31	-2.82	4.19	-0.68
SRR10260779_1	0.40	-1.27	3.30	0.28
SRR10260779_2	0.47	-0.26	3.62	1.03
SRR11458718_1	0.71	-1.55	6.34	0.81
SRR11458718_2	0.84	-1.87	6.43	0.66
SRR13605073_1	0.19	-1.48	15.42	1.70
SRR14005143_1	0.06	-0.07	1.97	0.72
SRR14005143_2	0.06	-1.04	3.47	0.62
SRR332538_1	1.18	-0.11	3.26	0.63
SRR332538_2	2.27	-0.49	4.80	0.69
SRR341725_1	0.05	-1.83	1.72	-0.97
SRR341725_2	0.05	-0.94	1.82	-0.28
SRR5853087_1	1.27	-1.95	8.31	0.94
SRR957915_1	0.92	-1.15	6.89	1.27
SRR957915_2	0.93	-0.83	7.06	1.38
Average	0.54	-1.38	4.88	0.40

Table A.3: *Method `s+aa x-c` applied to filtered k-mers.*

dataset	**Δweight[%]**	**Δfasta[%]**	**Δcounts[%]**	**Δoverall[%]**
SRR001665_1	0.02	-7.53	16.14	-3.48
SRR001665_2	0.03	-7.80	19.91	-3.73
SRR061958_1	0.55	-8.07	38.51	-2.58
SRR061958_2	0.55	-7.82	40.58	-2.91
SRR062379_1	0.57	-6.14	34.62	0.20
SRR062379_2	0.70	-6.53	33.71	-0.24
SRR10260779_1	0.94	-1.81	29.64	4.93
SRR10260779_2	0.99	-2.25	31.66	4.53
SRR11458718_1	1.48	-3.04	29.05	4.02
SRR11458718_2	1.90	-2.62	29.33	4.43
SRR13605073_1	0.58	-0.14	38.37	5.77
SRR14005143_1	1.23	-1.45	30.30	5.43
SRR14005143_2	1.18	-2.57	42.52	3.98
SRR332538_1	1.05	-1.26	24.45	3.13
SRR332538_2	2.42	-1.72	29.44	2.35
SRR341725_1	0.14	-3.30	17.58	0.47
SRR341725_2	0.14	-3.94	18.41	-0.06
SRR5853087_1	2.06	-6.07	34.30	1.21
SRR957915_1	1.62	-5.22	40.24	2.30
SRR957915_2	1.32	-6.84	42.47	-0.75
Average	0.97	-4.31	31.06	1.45

Table A.4: *Method `s+aa x=a` applied to unfiltered k-mers.*

dataset	**Δweight[%]**	**Δfasta[%]**	**Δcounts[%]**	**Δoverall[%]**
SRR001665_1	0.02	-7.05	15.87	-3.12
SRR001665_2	0.03	-7.48	19.67	-3.50
SRR061958_1	0.55	-8.00	39.65	-2.39
SRR061958_2	0.55	-7.72	41.90	-2.69
SRR062379_1	0.57	-5.30	36.16	1.15
SRR062379_2	0.70	-5.94	35.40	0.52
SRR10260779_1	0.94	-1.87	30.31	5.02
SRR10260779_2	0.99	-2.09	32.51	4.83
SRR11458718_1	1.48	-2.43	31.48	5.03
SRR11458718_2	1.90	-1.92	32.02	5.57
SRR13605073_1	0.58	-0.66	45.36	6.40
SRR14005143_1	1.23	-1.55	31.38	5.58
SRR14005143_2	1.18	-2.49	43.89	4.24
SRR332538_1	1.05	-1.16	28.07	3.83
SRR332538_2	2.42	-1.82	32.53	2.67
SRR341725_1	0.14	-3.80	20.71	0.63
SRR341725_2	0.14	-4.31	21.52	0.17
SRR5853087_1	2.06	-5.67	36.10	1.86
SRR957915_1	1.62	-4.59	41.11	2.97
SRR957915_2	1.32	-6.67	43.71	-0.45
Average	0.97	-4.13	32.97	1.92

Table A.5: *Method `s+aa x=a avg_flip_rle` applied to unfiltered k-mers.*

dataset	**Δweight[%]**	**Δfasta[%]**	**Δcounts[%]**	**Δoverall[%]**
SRR001665_1	0.02	0.62	5.13	1.39
SRR001665_2	0.03	2.06	6.03	2.65
SRR061958_1	0.55	-1.16	14.45	0.68
SRR061958_2	0.55	-0.99	14.40	0.57
SRR062379_1	0.57	0.23	12.22	2.09
SRR062379_2	0.70	0.42	12.09	2.24
SRR10260779_1	0.94	-0.10	12.40	2.58
SRR10260779_2	0.99	0.61	12.75	3.04
SRR11458718_1	1.48	0.52	14.09	3.50
SRR11458718_2	1.90	0.17	14.52	3.33
SRR13605073_1	0.58	0.37	24.42	4.06
SRR14005143_1	1.23	1.76	9.59	3.46
SRR14005143_2	1.18	0.93	9.94	2.24
SRR332538_1	1.05	0.84	10.86	2.55
SRR332538_2	2.42	1.48	13.14	3.01
SRR341725_1	0.14	-0.51	9.88	1.37
SRR341725_2	0.14	-0.50	9.94	1.31
SRR5853087_1	2.06	-0.80	16.58	2.33
SRR957915_1	1.62	0.07	15.94	2.69
SRR957915_2	1.32	-1.13	15.54	0.93
Average	0.97	0.24	12.70	2.30

Table A.6: *Method `s+aa x-c` applied to unfiltered k-mers.*

dataset	**Δweight[%]**	**Δfasta[%]**	**Δcounts[%]**	**Δoverall[%]**
SRR001665_1	11.50	8.89	12.20	9.76
SRR001665_2	12.26	11.03	12.98	11.51
SRR061958_1	46.32	41.36	12.27	35.23
SRR061958_2	48.31	43.51	11.77	37.12
SRR062379_1	38.31	35.27	13.24	30.08
SRR062379_2	38.48	35.58	13.20	30.29
SRR10260779_1	36.50	33.98	12.10	27.76
SRR10260779_2	37.22	34.32	12.34	28.31
SRR11458718_1	38.70	36.45	11.34	28.99
SRR11458718_2	39.12	35.57	11.33	28.43
SRR13605073_1	32.00	30.67	11.58	25.34
SRR14005143_1	17.80	16.55	12.37	15.43
SRR14005143_2	24.04	22.91	16.64	21.66
SRR332538_1	13.87	15.32	10.13	14.14
SRR332538_2	22.41	21.41	11.29	19.32
SRR341725_1	42.91	39.48	10.82	31.98
SRR341725_2	43.04	39.63	11.01	32.24
SRR5853087_1	43.55	40.92	10.91	32.94
SRR957915_1	41.19	38.04	11.90	31.54
SRR957915_2	45.23	42.38	11.92	35.87
Average	33.64	31.16	12.07	26.40

Table A.7: *Method `s+aa x-c` applied to unfiltered k-mers with $k = 15$.*

dataset	**Δweight[%]**	**Δfasta[%]**	**Δcounts[%]**	**Δoverall[%]**
SRR001665_1	0.09	1.64	5.53	2.45
SRR001665_2	0.12	1.72	5.81	2.47
SRR061958_1	1.77	1.78	18.78	4.26
SRR061958_2	1.82	1.81	19.38	4.11
SRR062379_1	1.55	1.70	15.52	4.15
SRR062379_2	1.87	1.94	15.18	4.31
SRR10260779_1	1.87	1.94	15.18	4.31
SRR10260779_2	1.68	1.75	13.67	4.52
SRR11458718_1	2.51	2.43	14.77	5.35
SRR11458718_2	2.97	2.11	14.91	5.14
SRR13605073_1	1.20	1.32	23.24	4.98
SRR14005143_1	1.73	0.93	10.60	3.10
SRR14005143_2	1.97	1.75	12.84	3.53
SRR332538_1	1.43	0.54	10.66	2.42
SRR332538_2	5.96	4.92	12.34	6.12
SRR341725_1	0.71	0.85	10.48	2.79
SRR341725_2	0.72	0.89	10.75	2.82
SRR5853087_1	6.07	4.63	17.10	7.24
SRR957915_1	2.94	1.80	17.64	4.79
SRR957915_2	2.98	2.63	18.94	5.12
Average	2.10	1.95	14.17	4.20

Table A.8: *Method `s+aa x-c` applied to unfiltered k-mers with $k = 21$.*

dataset	**Δweight[%]**	**Δfasta[%]**	**Δcounts[%]**	**Δoverall[%]**
SRR001665_1	0.23	-2.75	10.28	-1.43
SRR001665_2	0.22	-3.23	9.86	-2.11
SRR061958_1	0.29	-0.66	9.46	0.78
SRR061958_2	0.30	-0.47	9.29	0.92
SRR062379_1	0.56	-0.05	10.50	2.07
SRR062379_2	0.58	-0.16	10.58	1.84
SRR10260779_1	1.05	-0.57	13.50	2.39
SRR10260779_2	1.33	-0.49	13.75	2.51
SRR11458718_1	0.35	-0.93	25.52	2.83
SRR11458718_2	0.93	1.23	8.40	2.74
SRR13605073_1	0.75	0.28	7.53	1.25
SRR14005143_1	0.79	-0.42	11.65	1.45
SRR14005143_2	1.05	-1.38	13.52	0.19
SRR332538_1	0.05	-1.08	10.00	0.76
SRR332538_2	0.05	-1.06	10.10	0.71
SRR341725_1	1.85	-2.93	15.05	-0.08
SRR341725_2	0.88	-4.55	13.07	-1.94
SRR5853087_1	0.68	-2.91	12.07	-1.36
SRR957915_1	2.94	1.80	17.64	4.79
SRR957915_2	2.98	2.63	18.94	5.12
Average	0.66	-1.23	11.90	0.75

Table A.9: *Method `s+aa x-c` applied to unfiltered k-mers with $k = 41$.*

dataset	**Δweight[%]**	**Δfasta[%]**	**Δcounts[%]**	**Δoverall[%]**
SRR10849012_1	2.15	-1.41	16.26	2.07
SRR14556465_1	0.22	-0.84	21.56	1.06
SRR6869040_1	485.10†	-0.23	20.84	1.67
SRR13608728	-663.63†	-0.69	19.86	1.14

Table A.10: *Method `s+aa x-c` applied to unfiltered k-mers, additional metagenome datasets. † UST integer overflow error.*

References

[1] Anton Bankevich et al. “SPAdes: a new genome assembly algorithm and its applications to single-cell sequencing”. In: *Journal of computational biology* 19.5 (2012), pp. 455–477.

[2] Timo Bingmann et al. “COBS: a compact bit-sliced signature index”. In: *String Processing and Information Retrieval: 26th International Symposium, SPIRE 2019, Segovia, Spain, October 7–9, 2019, Proceedings 26*. Springer. 2019, pp. 285–303.

[3] Phelim Bradley et al. “Ultrafast search of all deposited bacterial and viral genomic data”. In: *Nature biotechnology* 37.2 (2019), pp. 152–159.

[4] Karel Břinda, Michael Baym, and Gregory Kucherov. “Simplitigs as an efficient and scalable representation of de Bruijn graphs”. In: *Genome biology* 22.1 (2021), pp. 1–24.

[5] Marco Brunato. “Rappresentazione efficiente di k-mer sets su grafo di overlaps”. MA thesis. Università degli Studi di Padova, 2021.

[6] Rayan Chikhi, Antoine Limasset, and Paul Medvedev. “Compacting de Bruijn graphs from sequencing data quickly and in low memory”. In: *Bioinformatics* 32.12 (June 2016), pp. i201–i208. ISSN: 1367-4803. DOI: `10.1093/bioinformatics/btw279`. eprint: `https://academic.oup.com/bioinformatics/article-pdf/32/12/i201/17130531/btw279.pdf`. URL: `https://doi.org/10.1093/bioinformatics/btw279`.

[7] Rayan Chikhi, Antoine Limasset, and Paul Medvedev. “Compacting de Bruijn graphs from sequencing data quickly and in low memory”. In: *Bioinformatics* 32.12 (2016), pp. i201–i208.

[8] J Martin Collinson. “CSI: Birding–DNA-based identification”. In: *British Birds* 110 (2017), pp. 8–26.

[9] Wikimedia Commons. *File:DNA transcription.svg — Wikimedia Commons, the free media repository.* [Online; accessed 3-March-2023]. 2022. URL: `https://commons.wikimedia.org/w/index.php?title=File:DNA_transcription.svg&oldid=645230141`.

[10] Wikimedia Commons. *File:History of sequencing technology.jpg — Wikimedia Commons, the free media repository.* [Online; accessed 7-March-2023]. 2022. URL: `https://commons.wikimedia.org/w/index.php?title=File:History_of_sequencing_technology.jpg&oldid=708003683`.

[11] Wikimedia Commons. *File:Peptide syn.svg — Wikimedia Commons, the free media repository.* [Online; accessed 3-March-2023]. 2021. URL: `https://commons.wikimedia.org/w/index.php?title=File:Peptide_syn.svg&oldid=542460417`.

[12] Wikimedia Commons. *File:Phosphate backbone.jpg — Wikimedia Commons, the free media repository.* [Online; accessed 2-March-2023]. 2022. URL: `https://commons.wikimedia.org/w/index.php?title=File:Phosphate_backbone.jpg&oldid=702852310`.

[13] GitHub Community. *loadtxt() changes numbers if integers are read as strings.* [Online; accessed 27-December-2022]. 2020. URL: `https://github.com/numpy/numpy/issues/17277`.

[14] Temesgen Hailemariam Dadi et al. “DREAM-Yara: an exact read mapper for very large databases with short update time”. In: *Bioinformatics* 34.17 (2018), pp. i766–i772.

[15] Luca Denti et al. “MALVA: genotyping by Mapping-free ALlele detection of known VAriants”. In: *Iscience* 18 (2019), pp. 20–27.

[16] Robert S Harris and Paul Medvedev. “Improved representation of sequence bloom trees”. In: *Bioinformatics* 36.3 (2020), pp. 721–727.

[17] Robert S Harris and Paul Medvedev. *Improved representation of sequence Bloom trees. bioRxiv.* 2018.

[18] Marek Kokot, Maciej Długosz, and Sebastian Deorowicz. “KMC 3: counting and manipulating k-mer statistics”. In: *Bioinformatics* 33.17 (2017), pp. 2759–2761.

[19] Guillaume Marçais and Carl Kingsford. “A fast, lock-free approach for efficient parallel counting of occurrences of k-mers”. In: *Bioinformatics* 27.6 (2011), pp. 764–770.

[20] Camille Marchet et al. “REINDEER: efficient indexing of k-mer presence and abundance in sequencing datasets”. In: *Bioinformatics* 36.Supplement_1 (2020), pp. i177–i185.

[21] Brian D Ondov et al. “Mash: fast genome and metagenome distance estimation using MinHash”. In: *Genome biology* 17.1 (2016), pp. 1–14.

[22] Prashant Pandey et al. “Mantis: a fast, small, and exact large-scale sequence-search index”. In: *Cell systems* 7.2 (2018), pp. 201–207.

[23] Prashant Pandey et al. “Squeakr: an exact and approximate k-mer counting system”. In: *Bioinformatics* 34.4 (2018), pp. 568–575.

[24] Armando J Pinho and Diogo Pratas. “MFCompress: a compression tool for FASTA and multi-FASTA data”. In: *Bioinformatics* 30.1 (2014), pp. 117–118.

[25] Amatur Rahman, Rayan Chikhi, and Paul Medvedev. “Disk compression of k-mer sets”. In: *Algorithms for Molecular Biology* 16.1 (2021), pp. 1–14.

[26] Amatur Rahman and Paul Medvedev. “Representation of k-mer sets using spectrum-preserving string sets”. In: *International Conference on Research in Computational Molecular Biology*. Springer. 2020, pp. 152–168.

[27] Guillaume Rizk, Dominique Lavenier, and Rayan Chikhi. “DSK: k-mer counting with very low memory usage”. In: *Bioinformatics* 29.5 (2013), pp. 652–653.

[28] F Nicklen Sanger. “S. and Coulson, AR (1977) Proc”. In: *Natl. Acad. Sci. USA*. Vol. 74, pp. 5463–5467.

[29] Brad Solomon and Carl Kingsford. "Fast search of thousands of short-read sequencing experiments". In: *Nature biotechnology* 34.3 (2016), pp. 300–302.

[30] Brad Solomon and Carl Kingsford. "Improved search of large transcriptomic sequencing databases using split sequence bloom trees". In: *Journal of Computational Biology* 25.7 (2018), pp. 755–765.

[31] DS Standage, CT Brown, and F Hormozdiari. *Kevlar: A mapping-free framework for accurate discovery of de novo variants. iScience, 18: 28–36.* 2019.

[32] Chen Sun and Paul Medvedev. "Toward fast and accurate SNP genotyping from whole genome sequencing data for bedside diagnostics". In: *Bioinformatics* 35.3 (2019), pp. 415–420.

[33] Chen Sun et al. "Allsome sequence bloom trees". In: *Journal of Computational Biology* 25.5 (2018), pp. 467–479.

[34] Derrick E Wood and Steven L Salzberg. "Kraken: ultrafast metagenomic sequence classification using exact alignments". In: *Genome biology* 15.3 (2014), pp. 1–12.