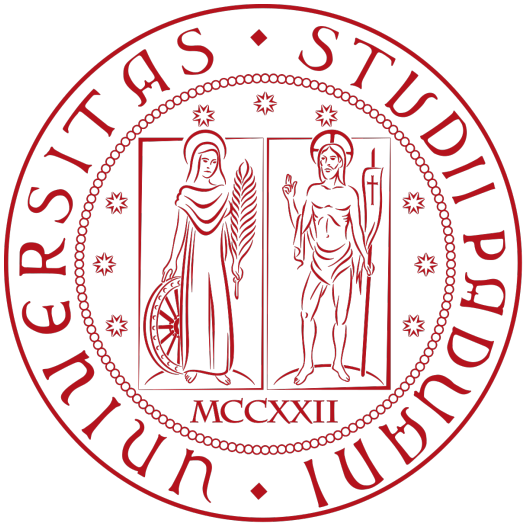


Compression of k-mers set with counters

Bioinformatics Course

Supervisor: prof. Matteo Comin

Enrico Rossignolo *(1218951)*

Leonardo Gemin *(2023860)*

Master's degree in Computer Engineering

Department of Information Engineering

University of Padua

November 3, 2022

Abstract

In this paper we want to show if it is possible to obtain a good compression ratio for kmer sets using standard compressors after using BCALM, UST, ProphAsm and Metagraph on k-mer sets we have as input. We found that preprocessing the input with any of the previous tools still perform better than using a direct compression. The best tool found is UST both for representation with counts and without counts.

Contents

1 | Introduction

Most of bioinformatics analysis are performed by k-mer-based tools. These tools operate primarily by reducing the input sequence data, which can be of various lengths, to a set of fixed-length strings called k-mer. K-mer-based methods are used in a wide range of applications, including genome assembly, metagenomics, genotyping, phylogenomics and database searching.

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 comprised of 31-mer of 450,000 microbial genomes, takes approximately 12 TB to be stored in compressed form[8].

The storage of datasets of this size is representative of the bottleneck due to the data structure defined by the k-mer. A k-mer, in fact, is defined as a substring of length k contained in a biological sequence, so the list of all possible k-mer is really very large: 4^k, where 4 is the number of nucleotide bases (A, C, T, G).

Compression of k-mer, associated with the development of software capable of handling compressed data, proves to be a crucial step.

1.1 Related works

Rahman and Medvedev translated the problem of finding the more space-efficient representation in finding the minimum spectrum preserving string set[1]. Indeed, they showed that this is equivalent to finding the smallest path cover in a de Bruijn graph (an NP-Hard problem for standard graphs). So they designed a greedy algorithm named UST that can encode kmers with their counts. One of the dataset they compressed, *Zebrafish RNA-seq*, with

[1]See section 2.3

$K = 31$, achieved an average bit per k-mer of 5.4 after running MFCompress on the output of UST, with counts. This implies a compressed size of 673.6 MB and a compression ratio of $\simeq$ 11.5. Furthermore, they managed to compress 19,000 de-noised microbial DNA data, obtaining a compression ratio of 35, starting from gzipped[2] fasta files[9].

Břinda, Baym, and Kucherov proposed *simplitigs* - equivalent to an SPSS - as a way to store de Bruijn graphs. They created prophAsm that use a similar greedy algorithm to UST and compressed its output with *xz*. They managed to compress genomes and pan-genomes with compression ratio between 4 and 5[3] and achieving an average bit per k-mer between 2 and 3.

However the two tools are not directly comparable because of different datasets: *ProphAsm* researchers uses a dataset from genome assemblies and applied frequency-based k-mer filtering. This may provide a better compression ratio than if they used a more raw k-mer set.

Finally, Karasikov et al. used *Metagraph* to compress losslessly[4] Virus PacBio HiFi read sets and obtained a compression ratio of 14.7 without further compression with general purpose compressors. In this way, this representation remains searchable.

[2]This means that the compression ratio using the uncompressed file is much higher.
[3]They computed the ratio differently considering simplitigs instead of the original file.
[4]Adding kmer coordinates as annotation allows them to rebuild the original sequences.

2 | Representations of a k-mers set

Sequences of k nucleotides[1] (A, C, T, G: Adenine, Cytosine, Thymine and Guanine) are known as *k-mers*, from ancient Greek *méros*, "part". We know k-mers set can be represented in different ways among which de Bruijn graphs which in turn are often encoded as unitigs[2].

In practice, many people encode their k-mers in a text file and then compress it with a file compressor, like *gzip*, because the overall process is faster but perhaps there are more space-efficient ways to store them[9].

2.1 Counting de Bruijn graphs

Recall that the i-prefix of a string $s = \langle s_1, \ldots, s_l \rangle$ of length l is defined as

$$pref_i(s) = \langle s_1, \ldots, s_i \rangle$$

while the j-suffix of s is

$$suf_j(s) = \langle s_{l-j+1}, \ldots, s_l \rangle$$

Definition Given two string s and t of length at least k, there exist a **binary connectivity relation** $s \rightarrow_k t$ if and only if $suf_k(s) = pref_k(t)$. If $s \rightarrow_k t$, then the **k-merging operation** is defined as the string

$$s \odot^k t = \langle s, suf_{|t|-k}(t) \rangle$$

Definition Given a set of k-mers K, the **de Bruijn graph** of K if defined as the directed graph $dBG(K) = (V, A)$ with

[1] The same thing can be adapted to amino acids.

[2] See section 2.2

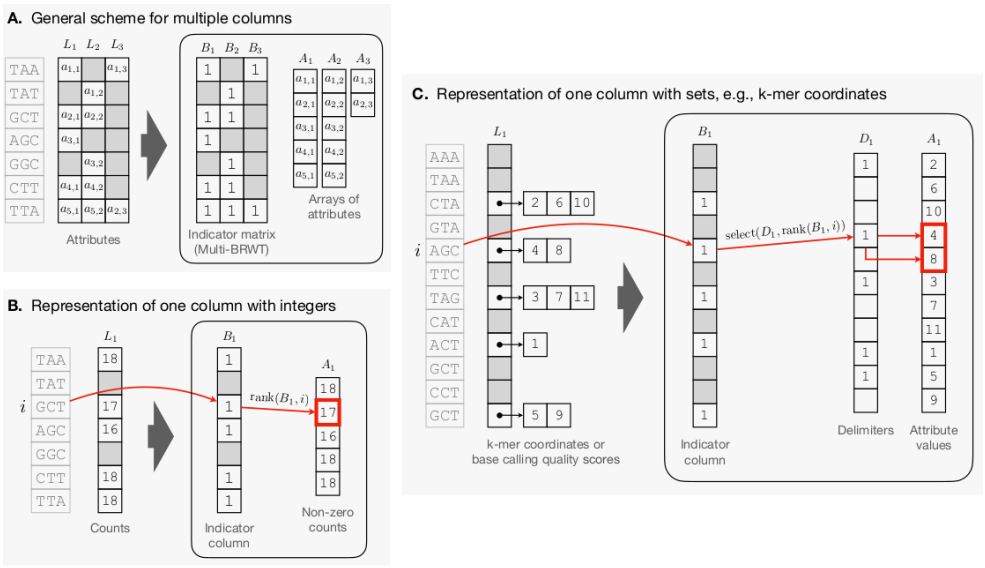


Figure 2.1: External data structures for Counting De Bruijn graphs.

1. $V = K$ and
2. $A = \{(u, v) \in K \times K \mid u \rightarrow_{k-1} v\}$

Definition Given a path $p = \langle v_1, \ldots, v_l \rangle$ (called **contig**) in a de Bruijn graph, the **spelling** of p is defined as the sequence of k-merging operations

$$spell(p) = v_1 \odot^k v_2 \odot^k \cdots \odot^k v_l$$

Nodes in a de Bruijn graph can be annotated to store additional information. Karasikov et al. found a way to compactly annotate nodes in an external data structure (see figure 2.1) focusing in particular on k-mers counts and k-mers coordinates - that is, k-mers positions in a list or in a genome.

In general, every k-mer can be associated to one or more list of attributes (figure 2.1 A) represented by a matrix that can be decomposed to a binary matrix and arrays of attributes equal to the number of columns. Assuming the binary matrix sparse, it can be compressed using RRR (Raman Raman Rao [10]) bit vector representation that can answer rank queries while the attributes arrays can be compressed using DAC (Direct Addressable Codes [3]) that support direct access.

The arrays can be additionally compressed using the *row diff transform* [4]. We will consider only the case of k-mer counts, see the paper [5] for k-mer coordinates. This transformation assumes a very likely similarity between

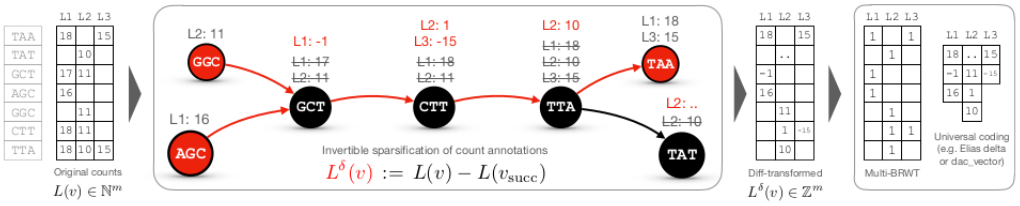


Figure 2.2: The row-diff transform using counts as annotations. Red nodes as anchors and they have original values. There are three different labels already transformed.

adjacent nodes, that is, k-mers that have a binary connectivity relation may have a similar or equal count. In the following way labels can be zeroed or, at least, they can be represented with less bits. Furthermore, the binary matrix will be sparsified. The transformed count of a node v is given by

$$L^\delta(v) \triangleq L(v) - L(v_{succ}) \;\; \forall v \in V \setminus \mathcal{A}$$

where v_{succ} is the marked successor of v. Successors of all $v \in K$ are marked arbitrarily. Nodes that terminates every path are called **anchors** (in the set $\mathcal{A}$), they are leaved unchanged as they serve to break the recursion (see figure 2.2).

The binary matrix can be compressed again exploiting column correlations using a Multi-BRWT (Multi Binary Relation Wavelet Tree) [6]. At the same time this will improve query speed.

2.2 Simplitigs

Definition Given a de Bruijn graph, the literature refers to a **unitig** as a path of nodes $p = \langle n_1, n_2, \ldots, n_l \rangle$ in which $n_1, \ldots, n_{l-1}$ have out-degree *exactly* 1 and $n_2, \ldots, n_l$ have in-degree *exactly* 1. In other words, a path with non-branching internal nodes.

Břinda, Baym, and Kucherov define simplitigs as follows.

Definition Given a set K of k-mers, consider its corresponding de Bruijn graph $G = (K, A)$. A **simplitig graph** $G' = (K, A')$ is a spanning subgraph of G that has no cycles and in-degree and out-degree *at most* one for every node. Spellings of paths in this particular graph are called **simplitigs**.

Note that it follows from the definition that a simplitig graph is a union of

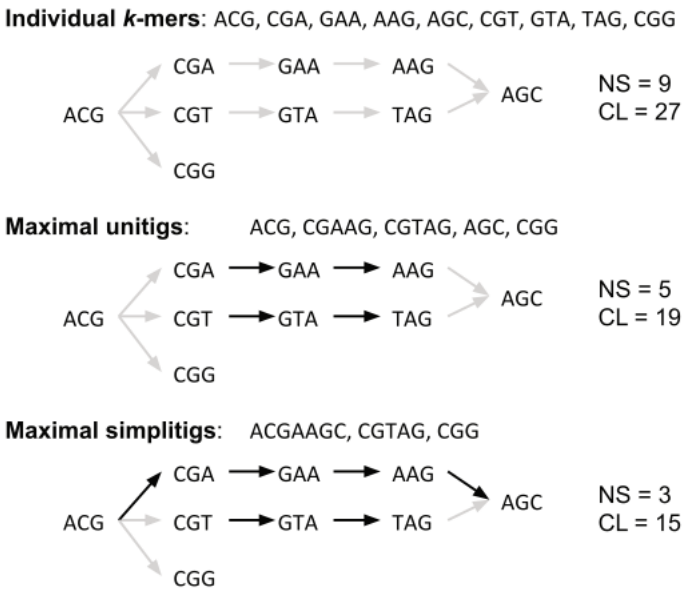


Figure 2.3: Simplitigs are spellings of disjoint paths from a de Bruijn graph. Unitigs are a particular case of simplitigs where paths consist of non-branching nodes. NS and CL are the number of sequences and cumulative length respectively.

vertex-disjoint paths (see figure 2.3) and that unitigs are a particular case of simplitigs.

Given a k-mer set K with $|K| = n_k$ and its simplitigs representation S_K, we can define two quantities:

1. the **cumulative length** (CL): it's the sum of the simplitigs lengths

$$CL = \sum_{s \in S_K} |s|$$

2. the **number of sequences** (NS): it's the number of simplitigs

$$NS = |S_K|$$

The two quantities are related by the following formula

$$CL = n_k + (k-1) \cdot NS \tag{2.1}$$

Minimizing 2.1 (assuming $NS \geq 1$) we obtain the following lower bound

$$CL \geq n_k + k - 1 \tag{2.2}$$

Indeed, $NS = 1$ represent the maximum compactness with a single simplitigs containing all the k-mers. But is this bound theoretically tight? That is, given a set of k-mers K, does *always* exist a set of simplitigs such that $CL = n_k + k - 1$? The answer is no, and it's not difficult to prove it with a counter-example. Let's consider

$$K = \{AAAA, AAAB, CCCC\}$$

in which the best CL is achieved with

$$S_K = \{AAAAB, CCCC\}$$

but $CL = 9 > n_k + k - 1 = 6$, therefore the bound is not tight.

Since we need to minimize NS in order to minimize the file size, we are interested in computing maximal simplitigs[3]. A maximal simplitig is a simplitig that, if extended forward or backward, it is no more a simplitig, that is, it is no more vertex-disjoint.

Finding an optimal simplitigs representation translates to the *vertex-disjoint path cover* problem (for general graph) that is NP-Hard since can be reduced from Hamilton. However it is not known if there exist a polynomial algorithm for the restriction of de Bruijn graphs[2][9]. Anyway Břinda, Baym, and Kucherov designed a greedy algorithm with speed in mind, not optimality.

The Prophasm tool 3.1.2 compute simplitigs using a simple greedy algorithm described in Listing 2.1. As we can see, the algorithm start drawing a random k-mer from the k-mer set K and try to extend it forward and then backward. Note that, if we have branching nodes, the choice is made following the k-mer lexicographically smaller (row 6).

```
1  def extend_simplitig_forward(K, simplitig):
2    extending = True
3    while extending:
4      extending = False
5      q = simplitig[-k+1:] # (k-1)-suffix
6      for x in ["A","C","G","T"]: # lexicographically smaller
7        kmer = q + x
8        if kmer in K:
9          extending = True
10         simplitig = simplitig + x
11         K.remove(kmer)
12         K.remove(reverse_complement(kmer))
```

[3]In the best case it's a single simplitigs like an assembled genome. This is confirmed by the fact that Břinda, Baym, and Kucherov found that assemblies are more space-efficient than (non-maximum) simplitigs in genome datasets.

```
        break
  return K, simplitig

def get_maximal_simplitig(K, initial_kmer):
  simplitig = initial_kmer
  K.remove(initial_kmer)
  K.remove(reverse_complement(initial_kmer))
  # forward extension
  K, simplitig = extend_simplitig_forward(K, simplitig)
  # backward extension
  simplitig = reverse_complement(simplitig)
  K, simplitig = extend_simplitig_forward(K, simplitig)
  return K, simplitig

def compute_simplitigs(kmers):
  K = set() # remove k-mers multiplicity
  for kmer in kmers:
    K.add(kmer)
    K.add(reverse_complement(kmer))
  simplitigs = set()
  while |K|>0:
    initial_kmer = K.random()
    K, simplitig = get_maximal_simplitig(K, initial_kmer)
    simplitigs.add(simplitig)
  return simplitigs
```

Listing 2.1: The function *compute_simplitigs* used by ProphAsm to compute simplitigs. Given a list of k-mers, it return a set of simplitigs.

An important consideration is that the algorithm 2.1 does not take into account k-mers multiplicity but it collapse everything in a set. In this way we lost k-mers counts.

2.3 Spectrum Preserving String Set (SPSS)

Curiously, simplitigs was introduced independently by Rahman and Medvedev with the name of Spectrum Preserving String Set[2][9] but with a more complex theory behind it.

In this section we only consider **bidirected dBGs**, that is dBG with directed nodes. A **directed** node v has two sides: $(v, 0)$ and $(v, 1)$ in which its associated kmer is the forwarded or the reversed-complement one respectively.

Definition A de Brujin graph of K is said **compact**, noted with **cdBG(K)**,

if the spell of each node is a maximal unitig.

We observe that all dBGs can be transformed in cdBGs simply by collapsing maximal unitigs in single nodes.

Definition Given a nucleotide sequence s, Rahman and Medvedev define it's **k-spectrum** $sp^k(s)$ as the *multi-set* of all *canonized* k-mers of s.

$$sp^k(s) = \left\{\left\{w \in \{A,C,T,G\}^k \,\middle|\, \text{w is a canonized substring of s}\right\}\right\}$$

The k-spectrum of a set S is the union of the k-spectrum of its elements

$$sp^k(S) = \bigcup_{s \in S} sp^k(s)$$

Note that, if $|s| < k$ then $sp^k(s) = \varnothing$.

Definition Given a set of canonical k-mers K, and a set of strings S, S is said to be a **k-spectrum preserving string set (SPSS) representation** of K if and only if

1. their k-spectrum are equal:

$$sp^k(K) = sp^k(S)$$

2. and each strings of S have length at least k:

$$\forall s \in S, |s| \geq k$$

For brevity, we will say that S represents K.

Definition Given a set of sequences S, Rahman and Medvedev define the **weight** of S as

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

Note that this corresponds to the cumulative length (CL) defined for simplitigs.

It can be shown that a path cover of $cdBG(K)$ represents K, then we have the following claim (similar to 2.1):

Theorem[4] Let S^{opt} be a minimum weight SPSS representation of K. Let W^{opt} be the smallest path cover on cdBG(K). Then

$$weight(S^{opt}) = |K| + |W^{opt}| \cdot (k-1) \tag{2.3}$$

[4]See [9] page 3 for the formal proof.

In this way we got a link between an optimal SPSS and an optimal path cover in a cdBG.

Definition Given a de Bruijn graph G and a node side (u, su), they define a **special neighbourhood** $B_{u,su}$ as all the node sides that have their opposite linked to (u, su):

$$B_{u,su} = \left\{ (v, sv) \,\middle|\, \big((u, su), (v, sv - 1) \big) \in E \right\}$$

A node side $(v, sv) \in B_{u,su}$ is called **special side**.

Rahman and Medvedev came up with a tighter lower bound with respect to Břinda, Baym, and Kucherov:

Theorem Given a path cover W over a cdBG(K) we have

$$|W| \geq \left\lceil \frac{n_{dead} + n_{sp}}{2} \right\rceil + n_{iso} \tag{2.4}$$

where n_{dead} is the number of nodes that are dead ends,

$$n_{sp} = \sum_{(u,su) \in (V,\{0,1\})} max\{0, |B_{u,su}| - 1\}$$

and n_{iso} is the number of isolated nodes.

Using the relationship 2.3 we finally get the lower bound

$$weight(S^{opt}) \geq |K| + \left(\left\lceil \frac{n_{dead} + n_{sp}}{2} \right\rceil + n_{iso} \right) \cdot (k - 1) \tag{2.5}$$

But is this bound theoretically tight? Again, the answer is no. Consider the counter-example on figure 2.4 in which we have a lower bound of 2 but an optimal path cover of size 4. This means that, by equation 2.3 the bound is not tight.

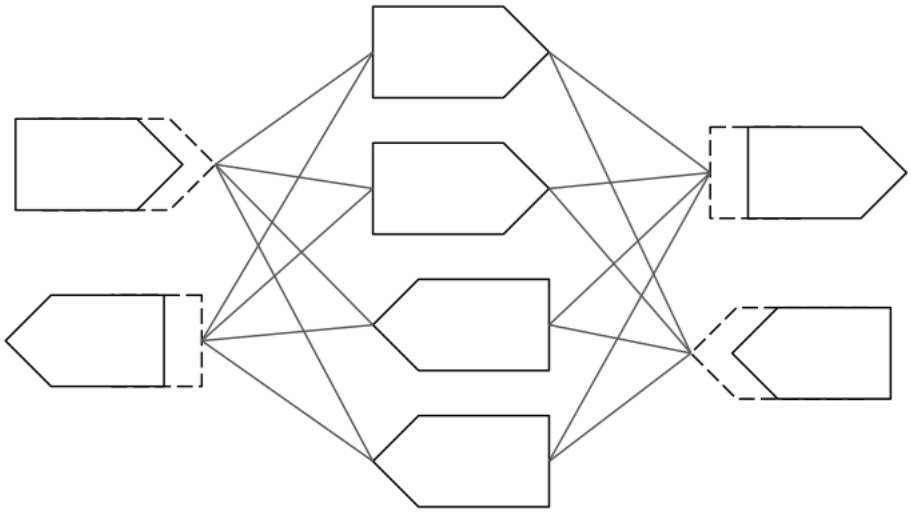


Figure 2.4: In this oriented cdBG we have $n_{dead} = 4$, $n_{sp} = 0$, $n_{iso} = 0$ giving a lower bound of 2 but the optimal path cover has cardinality 4, giving an optimal size greater than 4.

3 | Experimental setup

3.1 Tools

The first step was to prepare the work environment. For convenience, on the cluster, we have used *Bioconda*, a distribution of bioinformatics software realized as a channel for the versatile *Conda* package manager.

3.1.1 Metagraph

MetaGraph is a tool for scalable construction of annotated genome graphs and sequence-to-graph alignment.

It takes one or more `fasta`/`fastq` files as input (may be gzipped) in order to build a De Bruijn graph. After a series of annotations and subsequent modifications, the results are saved in several files including:

- `.dbg`, where the resulting graph is saved;
- `.annodbg`, where the annotations binary matrix is stored;
- `.counts`, where the actual counts are saved.

The tool is updated to version 0.3.4.

3.1.2 ProphAsm

ProphAsm is a tool for computing *simplitigs* from k-mer sets. Upon execution, it firstly loads all specified datasets from `fasta` files, then computes its simplitigs, a new and more compact way to store information in order to create a De Bruijn graph. Output is also saved in a `fasta` file.

Unfortunately the tool does not support multithreading, so it turns out to be slow compared to other tools like Metagraph or UST. Furthermore, since it encodes k-mers with 64-bit integers, it is limited by a k-mer-length of 32.

Version 0.1.1 of Prophasm was installed.

3.1.3 BCALM

BCALM (version 2) is a bioinformatics tool for constructing the compacted De Bruijn graph from sequencing data.

File input format can be `fasta`/`fastq`, either gzipped or not. BCALM 2 does not care about paired-end information, all given reads contribute to k-mers in the graph. BCALM 2 outputs the set of unitigs of the de Bruijn graph. The tool generates a special header for each sequence, described in Appendix A.2.2.

The version used is v2.2.3.

3.1.4 UST (Unitig STitch)

UST is a bioinformatics tool for constructing a spectrum-preserving string set representation from sets of k-mers. It requires that the input dataset has received a preprocessing from bcalm. Those output files are used as input to UST, which outputs a `fasta` file with the SPSS representation.

UST is not included in the Bioconda library, so it must be installed as described in its repository.

The version we use is 1.0.

3.1.5 MFCompress

MFCompress is a dedicated compression tool used in [9] to squeeze FASTA files. Later it was merged with *UST* in a tool named *ESSCompress*[1].

3.2 Datasets

The sequences were downloaded from the NCBI's server with a tool in the *SRA-toolkit* named *prefetch*. Then we used *fasterq-dump* in order to convert

from the *SRA* format to the *FASTA* one. In this process only technical reads are rejected while all the biological reads are kept even with possible reading errors[1]. The commands used are:

```
$ prefetch --progress "$S"
$ fasterq-dump --threads $NTHREAD --progress
--skip-technical --fasta-unsorted --outfile "$S.fasta" "$S.sra"
```

where `$S` and `$NTHREAD` are the accession name and the number of threads to use respectively.

The SRRs (Sequence Read Run) we choose with their sizes are on table 3.1.

Accession	Description	FASTA file size [MB]
SRR000001	Human haplotype map	148.6
SRR21394969	E. Coli	314.7
SRR21394970	Coronavirus 2	705.7
SRR21284212	Salmonella	727.1
SRR21073883	N. Gonorrhea	3982.6

Table 3.1: Datasets downloaded from NCBI

For the sake of curiosity we added a completely random sequence named `RND1664714109` (of about 1GB, with reads of length 31) generated with our tool in the repository. This will not be considered in our average case since it is not a biological sequence.

3.3 Machine

These tools are executed on the Department's Blade cluster and on our personal computer.

The PC specifications are

- Processor: Intel(R) Core(TM) i7-3517U CPU @ 1.90GHz, dual-core with 2 thread per core
- RAM: 16 GB (32 GB of swap)

The cluster consists of many nodes equipped with

[1]A k-mer with abundance less than 2 is often considered a reading error and it is rejected from the k-mer set.

- 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
- gpu1 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 to the cluster.

4 | Methods

We build a bash script in order to systematically execute the same commands for each sequence varying the k-mer length. It automatically produce a nice *CSV* file with all the file sizes needed. Then this file is parsed by a python script that compute ratios and plot figures. These scripts can be found on GitHub (`https://github.com/enricorox/kmers-compression`).

Tools that encode kmer counts and tools that ignore them are considered separately.

We compressed files outputted by each tool with three different compressor: *gZip*, *lzma* and *MFCompress* with their maximum compression ratio (i.e. using `--best` option). Then we keep the file with the minimum size.

In order to see if there is some benefit reorganizing the sequences with these tools, we compared the results with the compressed original file.

We want to minimize the compression ratio computed as

$$r = \frac{Original\ input\,[Bytes]}{Compressed\ output\,[Bytes]}$$

so that an high ratio implies a reduction in space.

4.1 Without counts

Since *prophAsm* do not encode counts, we compared it with tools that do not (necessarily) produce counts. In this section we considered *ProphAsm*, *BCALM*, *UST* and *Metagraph*. The commands used in each method are reported in table 4.1.

All the tools produce a *FASTA* file (compressed or not) except for *Metagraph* that gives a custom *dbg* file (see table 4.1). Compressed files are decompressed in order to compress them again with different compressors.

ProphAsm	`prophasm -s "$outfile_stat" -k "$K" -i "$infile" -o "$outfile"`
BCALM	`bcalm -nb-cores $NTHREAD -kmer-size "$K" -abundance-min 1 -minimizer-size $bca_min_size -in "$infile" -out "$outfile_base"`
UST	`ust -k "$K" -i "$infile" -a 0`
Metagraph	`metagraph build --parallel "$NTHREAD" --kmer-length "$K" --count-kmers -o "$graph" "$infile"`
contigs	`metagraph clean -p $NTHREAD --to-fasta -o "$outfile_base.contigs" "$graph"`

Table 4.1: Commands used in each method without counts.

Method	Filetype
ProphAsm	*.fasta
BCALM	*.fasta
UST	*.fasta
Metagraph	*.dbg
contigs	*.fasta.gz

Table 4.2: Output file from methods without counts.

4.2 With counts

In this section we consider *BCALM*, *UST* and *Metagraph*. The commands used are in table 4.4.

Here *Metagraph* used the outputs given by *contigs* in order to annotate the graph.

Relevant files for each method are in table 4.3.

BCALM	`bcalm -nb-cores $NTHREAD -kmer-size "$K" -abundance-min 1 -minimizer-size $bca_min_size -all-abundance-counts -in "$infile" -out "$outfile_base"`
UST	`ust -k "$K" -i "$infile" -a 1`
Metagraph	*# Build the main graph*

	`metagraph build -p "$NTHREAD" -k "$K"` `--count-kmers -o "$graph" "$infile"`

	# Annotate the graph
	`metagraph annotate --anno-filename --count-kmers`
	`-p "$NTHREAD" -i "$graph" -o "$outfile_base"`
	`"$outfile_base.contigs.fasta.gz"`

	# Row diff transform (stage 0)
	`metagraph transform_anno --count-kmers`
	`--anno-type row_diff --row-diff-stage 0`
	`--disk-swap swap -p "$NTHREAD" -i "$graph"`
	`-o "$outfile_base.row_count" "$annotations_binary"`

	# Row diff transform (stage 1)
	`metagraph transform_anno --count-kmers`
	`--anno-type row_diff --row-diff-stage 1`
	`--disk-swap swap -p "$NTHREAD" -i "$graph"`
	`-o "$outfile_base.row_reduction"`

	# Row diff transform (stage 2)
	`metagraph transform_anno --count-kmers`
	`--anno-type row_diff --row-diff-stage 2`
	`--disk-swap swap -p "$NTHREAD" -i "$graph"`
	`-o "$outfile_base.row_diff" "$annotations_binary"`

	# row_diff_int_brwt transform
	`metagraph transform_anno --anno-type row_diff_int_brwt`
	`--greedy --fast -p "$NTHREAD" -i "$graph"`
	`-o "$outfile_base" "$annotations_binary"`

	# Relax annotation brwt
	`metagraph relax_brwt -p "$NTHREAD"`
	`-i "$graph" -o "$outfile_base"`
	`"$outfile_base.row_diff_int_brwt.annodbg"`

contigs	`metagraph clean -p $NTHREAD --to-fasta` `-o "$outfile_base.contigs" "$graph"`

Table 4.4: Commands used in each method with counts.

Method	Filetype
BCALM	*.fasta
UST	*.fasta, *.counts
Metagraph	*.dbg, *.column.annodbg, *.counts
contigs	*.fasta.gz, *.kmer_counts.gz

Table 4.3: Output file from methods with counts.

5 | Results

All the file sizes can be found in `results.csv` in our repository.

About *Metagraph*, since in all our cases the *brwt* transformation gives a bigger file than the *row diff* one, we kept the last one.

We found that the best compression tool is *lzma* (see table 5.1).

compressor	number of files with minimum size
lzma	155
MFCompress	25
gZip	0

Table 5.1: Overall frequency of compressed file with minimum size

5.1 Without counts

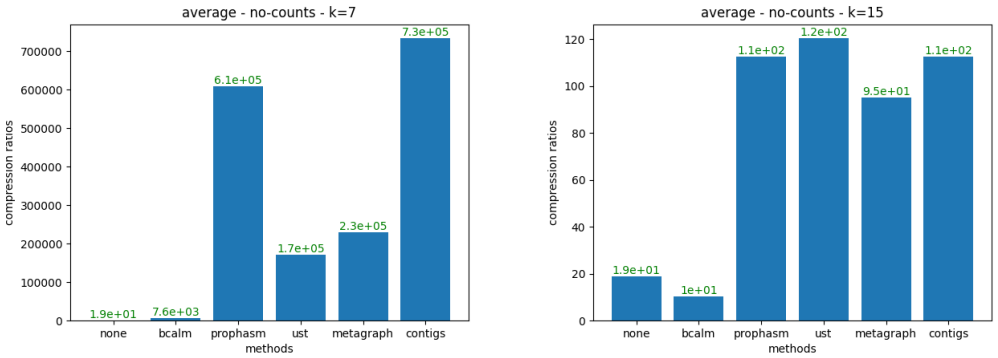
We found that with small k-mer length the method that works better is *contigs* while for medium and big k-mer length *UST* is the best.

We can see the average cases in figure 5.1.

A particular case is `SRR21073883` in which we obtain very high ratios. It is on table 5.2.

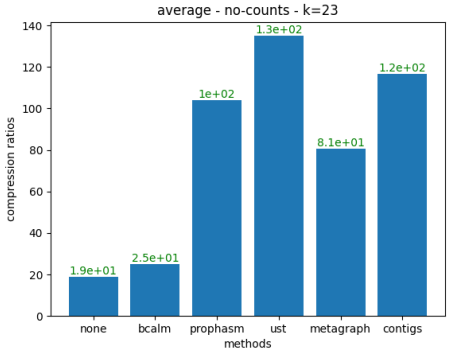
The random sequence give us comparable results for all the methods with $k \geq 15$ (see table 5.3).

Using the stat files, we have checked the lower bounds 2.2 and 2.5 for which there are gaps from 10.7% to 97.6% for ProphAsm and from 8.8% to 71.9% for UST (see tables 5.4 and 5.5). Note that it is considered the whole file size (with the overhead of the deflines and new lines).

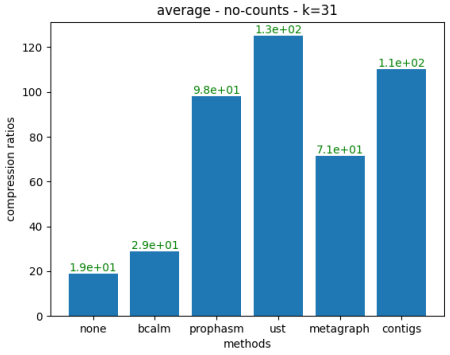


(a) $k = 7$

(b) $k = 15$



(c) $k = 23$



(d) $k = 31$

Figure 5.1: Compression ratios. Average cases with different k-mer lengths, without counts

k-mer length	none	BCALM	ProphAsm	UST	Metagraph	contigs
7	21.54	25526.08	2056108.31	583201.31	777867.53	2495414.66
15	21.54	34.37	352.40	393.78	308.85	366.26
23	21.54	80.034	323.00	442.52	256.55	369.77
31	21.54	90.89	304.78	409.91	224.03	346.93

Table 5.2: Compression ratios for `SRR21073883` without counts

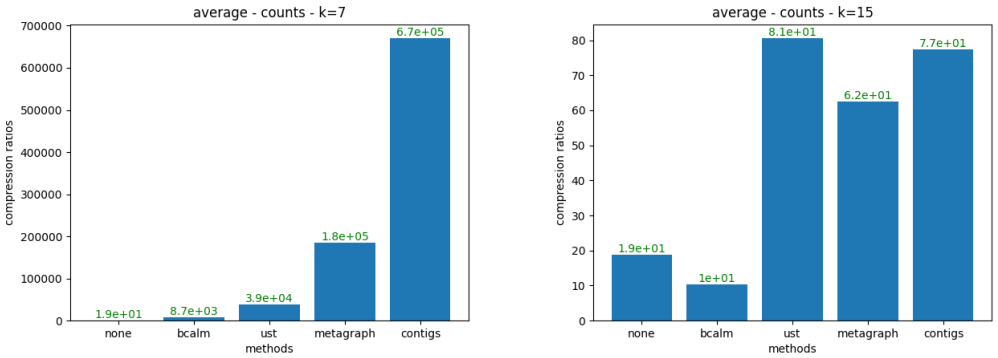
k-mer length	none	BCALM	ProphAsm	UST	Metagraph	contigs
7	3.97	8237.75	573626.66	155548.74	214213.71	870455.70
15	3.97	†	3.57	‡	4.23	3.03
23	3.97	3.89	3.89	3.91	4.18	5.28
31	3.97	3.96	3.96	4.19	4.18	5.33

Table 5.3: Compression ratios for `RND1664714109` without counts. (†) output file is too big for compression. (‡) input file is too big to keep in memory.

5.2 With counts

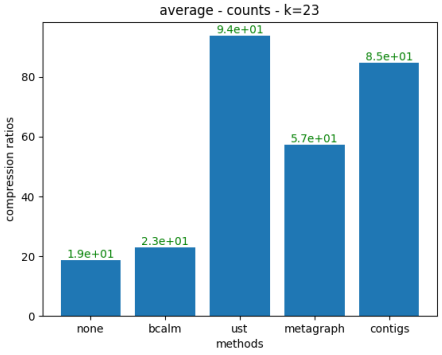
Considering k-mer counts gives pretty much the same results, only decreasing compression ratios because of counts overhead, see figure 5.2.

Even in this case, we got very good results for `SRR21073883` and poor results for `RND1664714109`, see tables 5.6 and 5.7.

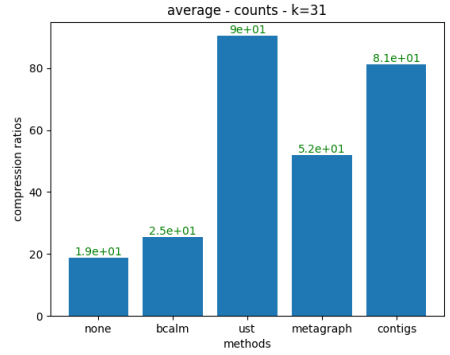


(a) $k = 7$

(b) $k = 15$



(c) $k = 23$



(d) $k = 31$

Figure 5.2: Compression ratios. Average cases with different kmer lengths, with counts

k-mer length	ProphAsm	LB	gap	UST	LB	gap
7	9.1K	8.2K	10.7 %	29.1K	8.2K	71.8%
15	47.1M	15.6M	66.9%	45.3M	31.2M	31.2%
23	57.5M	18.2M	68.3%	49.5M	44M	11.2%
31	69.3M	17.2M	75.2%	60.2M	54.8M	9%

Table 5.4: Lower bounds for SRR21073883 (without counts). All numbers are in byte

k-mer length	ProphAsm	LB	gap	UST	LB	gap
7	9.2K	8.2K	11.1%	29.1K	8.2K	71.9%
15	1348.5M	343.6M	74.5%	†	†	†
23	1376M	290.3M	78.9%	1096.8M	1000M	8.8%
31	1376M	32.3M	97.6%	1096.7M	1000M	8.8%

Table 5.5: Lower bounds for RND1664714109 (without counts). (†) input file is too big to keep in memory. All numbers are in byte

k-mer length	none	BCALM	UST	Metagraph	contigs
7	21.54	29266.99	126257.98	633478.89	2355222.82
15	21.54	34.12	260.98	201.39	252.74
23	21.54	75.41	308.95	181.29	274.31
31	21.54	83.90	300.16	162.52	262.04

Table 5.6: Compression ratios for `SRR21073883`, with counts

k-mer length	none	BCALM	UST	Metagraph	contigs
7	3.97	8915.63	54306.51	176301.91	809427.44
15	3.97	†	‡	3.05	2.61
23	3.97	3.88	3.84	3.79	5.27
31	3.97	3.96	4.19	3.79	5.33

Table 5.7: Compression ratios for `RND1664714109`, with counts. (†) output file is too big for compression. (‡) input file is too big to keep in memory.

6 | Conclusions

All the tools used gave us compression ratios that are higher than those with a direct compression (*none* method). That means that data are re-organized in such a way that is more compressible.

Clearly, we have got huge ratios with low k-mer sizes because it's more likely to have duplicates (that collapse in a single k-mer) with small length than with bigger ones.

From our results we get that *UST* is the method that works well on most cases both without and with counts.

ProphAsm, that has the same theory behind *UST*, has better performance only with low k-mer sizes.

As expected, *BCALM* does not provide any advantage to the compression ratio since it expands every sequence in subsequences of length k and its performance heavily depends on the repetitiveness of the dataset.

Metagraph is still better than nothing and than *BCALM* but does reach *UST* performance, may be because of the de Bruijn graph structure. Surprisingly, *contigs*, that is a different representation of the same de Bruijn graph, works really well with any k-mer size.

Finally, the fact that the random sequence has low compression ratios compared to biological sequences implies that the the correlation between reads is well exploited.

6.1 Future works

We still don't get how the compression ratio changes varying the input size due to its huge variability.

The next step would be checking the compression speed and memory usage

for each method and trying to do queries, if possible, without full decompression like in [5].

Forking *ProphAsm* in order to take into account k-mer multiplicity would be interesting in order to compare it with *UST* even with counts.

Finally, finding an optimal solution for the Minimum Path Cover would improve both *ProphAsm* and *UST* compression ratios.

Appendices

A | SRA, FASTA and FASTQ file format

A.1 SRA

SRA means Sequence Read Archive and it is both an online archive managed by the US National Center for Biotechnology Information (NCBI) and a file format.

There are two type of SRA format that can be downloaded form NCBI:

- normalized SRA format: it's the older one, it contains base calls, full base quality scores, and alignments. It has *.sra* extension.
- lite SRA format: it contains base calls, simplified quality scores, and alignments. This means that each reads is flagged with *pass* or *reject* instead of per base score so that it save space. It has *.sralite* extension.

This format can be easily converted in FASTA or FASTQ using `fasterq-dump` from the SRA toolkit.

A.2 FASTA and FASTQ

A.2.1 FASTA

FASTA is a file format derived by the homonym program and it is pronounced "FAST A" (i.e. FAST All, compared to FASTN for Nucleotides and FASTP for Proteins).

A FASTA file begin with a `>` character followed by some sequence information. This is the so called *defline*; the NCBI defines some rule and modifiers

in order to upload the file correctly in their archive but, in principle, it can be blank since all the tools we used ignore it. An example taken from[7] is (all in one line):

```
>Seq1 [organism=Streptomyces lavendulae] [strain=456A]
Streptomyces lavendulae strain 456A mitomycin radical oxidase
(mcrA) gene, complete cds.
```

After the defline we have the sequence itself that can be splitted in multiple lines.

Since in a FASTA file it is allowed only one `>` at the beginning, multiple FASTA file can be merged in a single one without problems. This file type is called *Multi-FASTA*.

A.2.2 BCALM defline

A remarkable case is the BCALM defline in its output FASTA file:

```
><id> LN:i:<length> KC:i:<abundance> km:f:<abundance>
L:<+/->:<other id>:<+/-> [...]
```

where

- `LN` is an integer field (`i`) representing the unitig length
- `KC` is an integer field representing total k-mers (that are in the unitig) abundance in the input file
- `km` is a float field (`f`) representing mean k-mers abundance. Since in our case the k-mer length is equal to unitig length, this is the same as KC
- `L` field encodes all possible combinations of forward/reverse edges whit this node.

A.2.3 FASTQ

FASTQ is an extension of FASTA that encode quality scores.

In this case the defline starts with an `@` character, followed by a string identifier. The second line contains the sequence. The third line contains a `+` character (optionally followed by the same sequence identifier) while the last line contains quality scores, one per nucleotide, encoded as an ASCII characters from `!` (worst) to `~` (best) giving 94 possible scores.

References

[1] Amatur Rahman. *ESSCompress v3.1*. Accessed Oct 7, 2022. `https://github.com/medvedevgroup/ESSCompress`.

[2] 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.

[3] Nieves R Brisaboa, Susana Ladra, and Gonzalo Navarro. "DACs: Bringing direct access to variable-length codes". In: *Information Processing & Management* 49.1 (2013), pp. 392–404.

[4] Daniel Danciu et al. "Topology-based sparsification of graph annotations". In: *Bioinformatics* 37.Supplement_1 (2021), pp. i169–i176.

[5] Mikhail Karasikov et al. "Lossless Indexing with Counting de Bruijn Graphs". In: *bioRxiv* (2021).

[6] Mikhail Karasikov et al. "Sparse binary relation representations for genome graph annotation". In: *Journal of Computational Biology* 27.4 (2020), pp. 626–639.

[7] NCBI. *FASTA Format for Nucleotide Sequences*. Accessed Oct 5, 2022. `https://www.ncbi.nlm.nih.gov/genbank/fastaformat/`.

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

[9] 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.

[10] Rajeev Raman, Venkatesh Raman, and Srinivasa Rao Satti. "Succinct indexable dictionaries with applications to encoding k-ary trees, prefix sums and multisets". In: *ACM Transactions on Algorithms (TALG)* 3.4 (2007), 43–es.