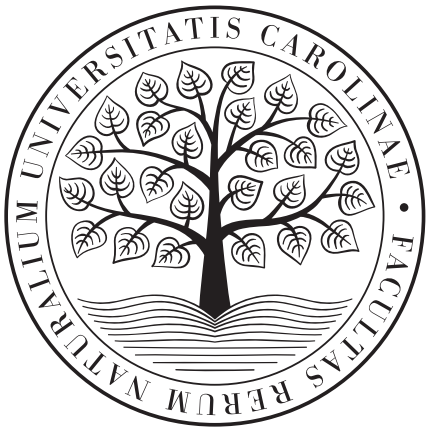
Charles University in Prague
Faculty of Science

BACHELOR THESIS



Ján Plachý

Targeted optimization of masked superstrings for *k*-mer sets

Department of Cell Biology

Supervisor of the bachelor thesis: Mgr. Pavel Veselý, Ph.D.

Study programme: Bioinformatics

Study branch: Bioinformatics

Prague 2025

I declare that I carried out this bachelor thesis on my own, and only with the cited sources, literature and other professional sources. I understand that my work relates to the rights and obligations under the Act No. 121/2000 Sb., the Copyright Act, as amended, in particular the fact that the Charles University has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 subsection 1 of the Copyright Act.

In ..... date ....

Author's signature

I am very grateful to Pavel Veselý, Ondřej Sladký and Karel Břinda for the knowledge and insights they shared with me during the work on this thesis, their comments, and the feedback they provided.

I am very grateful to Pavel for his kind approach and the time he spent guiding me towards the result.

I am especially grateful to Ondra for his careful and friendly support and for bringing me to this topic of turning a camel 🐫 into a dromedary 🐪.

Title: Targeted optimization of masked superstrings for *k*-mer sets

Author: Ján Plachý

Department: Department of Cell Biology

Supervisor: Mgr. Pavel Veselý, Ph.D., Computer Science Institute of Charles University

Abstract: The increase in genomic data collection over the last decades has created a challenge for efficient data storage and querying. Methods based on *k*-mers, substrings of a small fixed length, have proven to be particularly useful and outperform standard assembly-based methods in a variety of applications. The novel approach of masked superstrings generalized *k*-mer-based methods and provided state-of-the-art compression efficiency together with a simple and memory-efficient way to design data structures for *k*-mers. However, the optimization of masked superstrings, which is NP-hard, has only been done as a two-step process so far, leaving room for improvement.

In this thesis, we model the space complexity of masked superstrings with masks stored in run-length and Elias-Fano encodings. We propose a polynomial-time heuristic algorithm for joint optimization of masked superstrings, implement the algorithm, and test it on eukaryotic genomes and microbial pangenomes. We then experimentally prove that our implementation outperforms the best-performing methods used so far, both theoretically and practically, especially for pangenomic datasets.

Keywords: *k*-mer sets, bioinformatics, computational genomics, algorithms, shortest superstring problem,

Contents

Introduction

The amount of genomic data being collected worldwide has increased rapidly in the last two decades, thanks to next-generation sequencing (NGS) technologies. Efficient data storage and comparison techniques are therefore crucial for fields of interest depending on genomics, such as drug design, disease prevention and detection, environmental screening, and many more.

k-mers are substrings of genomic data of a small fixed length. They can be computed directly from sequenced reads, without the need for computationally demanding genome assembly. They can also be stored in a more efficient way than genomes. k-mer based methods are used for studying pangenomes and metagenomes, genomic data representing many individuals of possibly different species. In addition to storing the data, k-mers are used for fast searching by creating efficiently compressible k-mer data structures, such as indexes [Mar+21; SVB25; LZL19].

In recent years, k-mer-based methods have been used in a growing number of practical tasks such as large-scale data search [Kar+24; Bři+25], metagenomic classification [WS14; KD15; BBS18], prediction of transcription factor binding sites [Fle+13] and identification of disease risk factors in the genome [Cor+25].

The masked superstring framework is a recent generalization of textual representations of k-mer sets [SVB23]. It improves compressibility by introducing false positive k-mers, which are filtered out by a binary mask. The mask can be equipped with a number of interpretation functions to simplify set operations on k-mer sets [SVB25]. In addition, the combination of masked superstring with Burrows-Wheeler transform allows to create a memory efficient k-mer index [SVB24].

k-mers are defined in two different models, the uni-directional and the bi-directional model. The bi-directional model, which treats each k-mer as equivalent to the one with reverse complementary sequence, is used in most cases, as it has biological relevance. Computing the masked superstring of optimal (smallest) length is NP-complete in both models. This means that approximation algorithms and heuristics must be used for computation.

The practical compressibility of masked superstrings depends both on the superstring length and on the structure of the mask. So far, the best performing method for masked superstring optimization was to use the two-step optimization — a global greedy algorithm for superstring computation followed by mask optimization [SVB23].

However, the input of the second step and the overall quality of the result are highly dependent on the output of the first step. As the optimization in the first step does not consider the target of the second step, the results can be significantly worse for certain optimization criteria.

In this thesis, we model the compressibility of masked superstrings by a linear function of the superstring length and the count of maximal sequences of ones in the mask, called *runs*. We show that optimization of this objective is NP-complete by a

reduction from the shortest superstring problem.

We develop a greedy heuristic algorithm for the joint (single-step) optimization of a masked superstring with the aim of minimizing the defined objective function. The algorithm is based on the Aho-Corasick automaton and is parametrized by the amount of priority it gives to keeping the superstring short versus the number of runs small.

Since the Aho-Corasick automaton of a k-mer set would be impractically big, we only store the leaves of the tree and use the rest implicitly. The implementation is therefore called Leaf-Only Aho-Corasick automaton, LOAC.

We implement the algorithm inside a fork of an open-source tool KmerCamel 🐫 [SVB23].

We evaluate the performance of LOAC and the parameters of the resulting masked superstrings against the state-of-the-art implementation of two-step optimization from KmerCamel 🐫. We compare the result for eukaryotic genomes of model organisms, such as *S. cerevisiae* and *D. melanogaster*, and for microbial pangenomes from *Genomic datasets used for evaluation of k-mer representations and indexes* [VBS25].

For each dataset, we search for the optimal value of the input parameter called *run penalty*. It represents the ratio of the importance of the resulting superstring length to the number of runs of ones in the resulting mask considered by the optimization algorithm.

From the data gathered, we suggest the suitable values of the run penalty parameter to be used for different types of datasets.

We then use the optimal configuration for each dataset to compute the masked superstring and compare its length, the number of runs of ones, and the objective function with the results of the two-step optimization. We then compress the resulting masked superstrings with commonly used compression tools to demonstrate the superior compressibility of the masked superstrings produced by LOAC. We also demonstrate the practical usability of LOAC by measuring and comparing the running times and memory efficiency of the algorithms.

In Chapter 1, we briefly explain the biological and bioinformatical background relevant for the thesis. In Chapter 2, we formulate the prerequisites and give an overview of previous work related to the representation of k-mers and k-mer sets. In Chapter 3, we prove the NP-completeness of the joint optimization of the objective function studied and describe the algorithm for the joint optimization of masked superstrings. In Chapter 4, we describe the practical implementation details of the algorithm. In Chapter 5, we describe the experiments performed to evaluate the performance of our implementation and discuss the results.

The selected results are presented as tables and plots in Appendix A and Appendix B, respectively. The implementation can be found on GitHub: `https://github.com/Jajopi/Plachy-bc-thesis` or as a supplementary material submitted with this thesis.

Chapter 1

Background

This chapter contains a basic and simplified overview of the biological and bioinformatical background relevant to the work.

1.1 Genomic data in bioinformatics

1.1.1 Biology of informational macromolecules

Informational macromolecules play a crucial role in the existence of life. They are present in all known forms of life (except for some simple viruses, however, these are not always considered living organisms). There are two types of informational macromolecules:

- Nucleic acids, such as DNA (which encodes and stores genetic information) and RNA (with various functions ranging from storing and transferring information to catalyzing chemical reactions).
- Polypeptides that act as functional molecules by catalyzing chemical reactions or providing mechanical structures for the cell.

Other macromolecular compounds of cells, such as polysaccharides or polylipids, are not considered informational macromolecules, although their structures can be of high entropy.

DNA

Deoxyribonucleic acid (DNA) plays a key role in the storage of genetic information and serves as a template for transcription. The whole set of DNA contained in an organism is called a genome. The basic functional unit of the genome is called a gene. The definition of a gene is quite unspecific, as any piece of DNA can have various functions, depending on its context (the spatial arrangement of the DNA, the interaction with the DNA-binding proteins, etc.). The set of genomes of more individuals of the same species or even different species (for example, the bacterial population of the human intestines) is called a *pangenome* or a *metagenome*, respectively.

Although DNA is considered a very stable molecule, in eukaryotic cells, it is enclosed in the nucleus and protected by histones and other chromatin-forming proteins to avoid its interactions with factors that could cause mutations. DNA in the nucleus is present in the form of several linear molecules called chromosomes. The

number of chromosomes and the positions of different genes throughout the genome often vary even between closely related species.

Mutations are one of the most significant forces of evolution and are, in general, beneficial to species and life as a whole, as they allow adaptation to changing conditions. However, they are often accompanied by negative effects on individuals such as loss of functionality of fine-tuned cellular mechanisms, which creates strong evolutionary pressure for their suppression and correction.

In prokaryotic cells, DNA is present in several (often circular) molecules in the cytoplasm. Prokaryotes are single-cell organisms that lack cell compartmentation. This results in a higher mutation rate and faster evolution, which increases the flexibility of species but prevents the emergence of complexity of ”higher” living forms.

Eukaryotic cells also contain organelles such as mitochondria or chloroplasts that store their own DNA. According to endosymbiotic theory, these organelles originated from events of endosymbiosis, the absorption of a bacterial cell by an archaeal cell (or, in later events, an eukaryotic cell). The first eukaryotic cells appeared about two billion years ago. The DNA in organelles is not sufficient for their independent survival as it has lost most of its former genes.

DNA is composed of four (main) types of nucleotides, each of them contains saccharide deoxyribose, a phosphate group and a nucleotide base — either adenine, cytosine, guanine or thymine (denoted respectively as `A`, `C`, `G` and `T`). DNA is (most of the time) a double-stranded molecule, where each chain of nucleotides proceeds in an opposite direction. The canonical direction is the one in which transcription progresses and is indicated by $5' \to 3'$ (based on the chemical numbering of deoxyribose). For chains to pair correctly, each nucleotide base must be aligned with a complementary one, adenine with thymine, and cytosine with guanine (this is called a *canonical* pairing; although there are many spatial arrangements in which the nucleotides can pair non-canonically, they require specific conditions).

Central dogma of molecular biology

The flow of information between the molecules is described by the central dogma of molecular biology: The information moves only from a DNA molecule through replication to a new DNA molecule, through transcription to an RNA molecule, and through translation from an RNA molecule into the sequence of peptides that form proteins.

This idea has been proposed in 1958 [Cri58]. There have been several exceptions discovered since then — enzymes called reverse transcriptases are able to create DNA sequences based on an RNA template, there are mechanisms of RNA replication, and some proteins called prions can force their own conformation to other proteins of the same kind. However, most of the information that determines the development of living forms is contained in their DNA.

1.1.2 Sequencing methods

The key step in the study of informational macromolecules is to obtain information about their sequence. Data are collected from a source of physical signal. In the past, the single DNA molecule was insufficient for analysis. Therefore, most older sequencing methods rely on DNA amplification in the *polymerase chain reaction* (PCR) process.

Polymerase chain reaction [Erl89]

PCR uses thermally stable DNA polymerase enzymes to *amplify* (synthesize billions of copies of) a certain DNA fragment bounded by short sequences called *primers*. These are added into the reaction mixture in a large number of copies together with free nucleotides to be used to synthesize the DNA.

The amplification works in cycles consisting of three phases at different temperatures. In the *denaturation* phase (with a temperature $\approx$ 100°C), hydrogen bonds between the complementary bases break and the DNA strands unravel. In the *annealing* phase (with a temperature $\approx 50 - 65$°C), the primers bind (*hybridize*) to the DNA strands. In the *elongation* phase (with a temperature $\approx$ 75°C), the DNA polymerase synthesizes the rest of each sequence starting from the primer, using the strand to which the primer binds.

In ideal conditions, this would result in the number of DNA strands doubling in each iteration. Practically, about 30 iterations are sufficient to reach a number of copies that is sufficient for further analysis.

Sanger sequencing [Val+13]

The sequencing method developed by Frederick Sanger in 1977 is based on irreversible termination of DNA synthesis and separation of DNA fragments by electrophoresis.

The complement of an amplified DNA sample is synthesized in the mixture enhanced by labeled dideoxynucleotides in low concentrations. If one of those modified nucleotides becomes incorporated into the chain, the synthesis cannot continue, and the resulting fragment is shorter than it would be if a normal nucleotide was incorporated instead.

The fragments are then separated by length (molecular weight) using the gel electrophoresis method. The sequence can be obtained by reading the terminating nucleotides of fragments sorted by length, which can be fluorescently labeled.

Sanger sequencing was the most widely used method for almost 40 years, until the emergence of *next-generation sequencing* methods. It is still used in some cases because of its high precision and relatively large length of the sequences obtained (up to 1000 bases).

Next-generation sequencing [GMM16; SGA18; Hu+21]

Most sequencing data are currently produced by several methods developed after the year 2000, commonly referred to as the *next-generation sequencing* (NGS) or second-generation sequencing methods.

The most dominant method is Illumina sequencing; most of the reads produced yearly in the last decade were sequenced using this method. The method is often called *sequencing by synthesis*. Amplified DNA is cut into polynucleotides of length up to several hundreds of bases. They are then equipped with specific primers at both ends and randomly distributed across the sequencing chip, where they bind to complementary primers present on the chip in large numbers. The sequences are then amplified using PCR, but the primers are bound to the surface of the chip. This results in a huge number of copies of each sequence occupying small spots on the chip.

The complementary sequences are then synthesized in cycles. Each nucleotide carries fluorescent dye which prevents the synthesis from proceeding with more than

one nucleotide at a time. The fluorescent dye is then washed away from all nucleotides, allowing the next repetition to proceed. In each cycle, the optical signal from the dyes is detected for each spot on the chip.

Many similar approaches are also widely used [Liu+12]. NGS methods allow billions of sequences to be read at the same time. Therefore, they are also known as *massively parallel sequencing* and are very efficient in terms of the cost of the data produced.

The main disadvantage of NGS methods is the small length of individual reads, up to a few hundreds of bases. Therefore, in most cases, a lot of computational power is required for a further detailed analysis of the data obtained. In addition, some interesting features of genomes, such as the number of repetitions in highly repetitive regions, cannot be studied in this way.

Third-generation sequencing [Lee+16; STK10; Ama+20]

The *third-generation sequencing* methods are able to produce reads of length greater than 10,000 bases (possibly reaching several hundreds of thousands of bases). Although they are almost as old as NGS methods, they have not been used much in the past because of high error rates. However, thanks to the improvements in precision in recent years, these methods are currently on the rise.

In *Oxford Nanopore* sequencing, the single molecule of DNA travels through a special pore in a membrane, altering the electrical potential between the solutions on both sides of the membrane. The exact potential change depends on the geometry of the molecule (the specific bases passing through the pore) and can be measured precisely. The resulting potential is influenced by several bases present in the pore at the moment. Therefore, it is difficult to obtain the exact sequence from measurements, and machine learning principles are used. This technique outputs very long reads of lower quality and is also capable of detecting modified or non-standard bases. Sequencers are very small (often the size of a USB stick), but can use hundreds of thousands of pores [Goo+15; Sim+16].

The *Pacific Biosciencies* sequencing is another method that reads the sequence of a single molecule and is therefore called *single-molecule real-time* (SMRT) sequencing. It uses DNA polymerase placed close to a hole of depth and diameter ≈ 100 nanometers (called a *zero-mode waveguide)*, which effectively blocks light emitted from sources outside its small volume. The complementary strand of DNA is synthesized from nucleotides with attached fluorescent dyes, which are released during synthesis and detected. To increase error correction, the molecule is circularized and possibly read several tens of times during each sequencing run [RA15].

Despite the rise of third-generation sequencing methods, NGS methods are likely to remain widely used in the future, mainly because of their efficiency and ability to generate large amounts of data at once.

1.1.3 Challenges of genomic data management

As the amount of genomic data collected and used every day has been increasing yearly for several decades, new challenges arise for their storage and analysis [Ste+15].

There are various types of data produced by measurements and analyses, but the largest volume is obtained from the NGS sequencers in the form of FASTQ files. Those ”raw” data consume huge amounts of storage space. The simple solution to

this problem would be to use them for certain analyses right after the sequencing and discard them afterward. However, this is often expensive and inefficient as additional analyses would require repeating the sequencing, which is not always possible.

Another possible solution is to store as much raw data as possible. However, this approach is also very expensive because petabytes of sequencing data are generated daily. Various compression techniques have been developed to address this challenge [Her+19; DG13; Zhu+15].

Single genome sequencing data can be effectively stored by mapping reads to a reference genome [BM13], possibly storing only the differences, called *variants*. If the reference genome is not present, genome assembly can be performed [RG19; KM95]. However, this is quite impractical with pangenomic data (which come from a number of different individuals) and very challenging with metagenomic data (from different species).

Problems also arise with database storage and searching. As more genomes of different organisms and individuals are sequenced, storing the assembled genomes or individual genes in the databases requires more resources. In addition, searching databases for similar sequences (with tools such as BLAST [Alt+90]) becomes slower and more computationally demanding.

Many computational techniques have been developed to overcome the challenges faced by genomics [Bři16]. k-mer-based methods are especially promising.

1.2 k-mer based methods

k-mer sets are obtained by taking every substring of length k from given DNA sequences and discarding duplicates; the order of the substrings in the original sequences is not preserved (the mathematical perspective of the k-mer sets is explained in Chapter 2). The values of k usually range between 15 and 127, with 31 being the most used.

k-mer based methods find various usages in the field of comparative genomics. Probably the biggest advantage of storing genomic sequences in a form of k-mer sets over storing assembled genomes is the possibility of skipping the genome assembly phase. k-mers can be computed directly from sequenced reads (possibly filtering out those not present enough times, occurring due to sequencing errors).

In some cases, genome assembly or is too computationally demanding (for example, when studying pangenomes, or when we need to perform *de novo assembly* of a genome without known structure). Mapping reads (with mutations and errors) to a genome scaffold is a much harder task than creating a k-mer index. The efficient index also simplifies working with k-mer sets and can be optimized for different types of queries.

k-mer sets allow us to store only "biologically relevant" information — whole genomes often contain repetitive sequences like telomeric sequences at the ends of chromosomes or a significant amount of transposable elements at various places in the genome (for example, about 10% of the human genome is formed by about a million copies of *Alu* element [SNH98]). Those sequences are problematic for the genome assembly process, but are effectively compressed in the k-mer sets.

By encoding the genome as a k-mer set, we lose information about the absolute positions of genes, and even their respective sequences, which may seem like a problem. However, if we choose a sufficiently high k, the number of possible distinct k-mers is orders of magnitude higher than the number of k-mers we might get from the

genome. This makes it possible to recompute longer parts of the genome considering, for example, the paths in the de Bruijn graph (see Section 2.1.3 for more details). Moreover, when comparing genomes of (even closely related) species, the absolute positions of given parts of their genomes tend to differ dramatically (see, for example, Figure 4 of [McN+11]).

k-mer based techniques are increasingly used in a growing number of practical tasks. Notable applications include metagenomic classification [WS14; KD15; BBS18], large-scale data search [Kar+24; Bři+25], prediction of transcription factor binding sites [Fle+13], detection of structural variation and cancer detection [Abo+15], antibiotic resistance inference [Bři+20], viral infection detection [Ren+17; Bai+19], identification of disease risk factors in the genome [Cor+25], and many more [Moe+24].

k-mer sets can be effectively stored and used as a base for building indexes, allowing for a fast comparison of genomic data [Mar+21; SVB25; LZL19].

Chapter 2

k-mer set based techniques

2.1 Definitions

2.1.1 *k*-mers and complements

k-mer Q is defined as a string of length $k \in \mathbb{N}$ over alphabet $\Sigma_{DNA} = \{\mathtt{A}, \mathtt{C}, \mathtt{G}, \mathtt{T}\}$. There exist 4^k distinct k-mers of length k. Examples of k-mers of length 3 are `AAA`, `ATG`, `CAT`.

A *reverse complement* of k-mer Q, $\mathrm{RC}(Q)$ is the string obtained by substituting each letter of Q by the letter of its complementary base and reversing the order of the result. Complementary pairs of bases are $(\mathtt{A}, \mathtt{T})$ and $(\mathtt{C}, \mathtt{G})$. For example, $\mathrm{RC}(\mathtt{AAA}) = \mathtt{TTT}$, $\mathrm{RC}(\mathtt{ATG}) = \mathtt{CAT}$. We refer to the reverse complement of Q simply as a *complement* of Q.

Being a complement is a symmetrical relation — for k-mers Q and $Q' = \mathrm{RC}(Q)$, always $\mathrm{RC}(Q') = Q$. The lexicographically smaller k-mer from pair $(Q, \mathrm{RC}(Q))$ is called the *canonical k-mer* of Q.

Note that if k is even, there exist $4^{k/2}$ k-mers for which $Q = \mathrm{RC}(Q)$. For example k-mer `AAGCTT` is its own complement. This complicates the computation with k-mers and is a reason why odd values of k are mostly used.

2.1.2 *k*-mer sets

We focus on sets of k-mers obtained from the input sequences, denoted by K. The only information K stores about k-mers is whether they were present in the input. We therefore lose any information on the positions and counts of the respective k-mers in the input.

Most of the currently used sequencing techniques lack information on the DNA strand from which a sample originated. This means that, in most cases, we are only interested in storing whether Q **or** $\mathrm{RC}(Q)$ was present in the sample. This concept is called the *bi-directional model*, as opposed to *uni-directional model* where Q and $\mathrm{RC}(Q)$ are considered distinct. In the bi-directional model, there are only $\frac{4^k}{2}$ distinct k-mers for odd k and $\frac{4^k+4^{k/2}}{2}$ distinct k-mers for even k. In both models, it therefore holds that $\log_4 |K| \leq k$.

For practical purposes, it is often sufficient to use $k = 31$. Note that even the largest known genomes, such as the one of *T. oblanceolata* [Fer+24] (or *P. japonica* [PFL10], previously considered the largest) are approximately 150 billion base pairs in size; with $\log_4(1.5 \cdot 10^{10}) \approx 20$. Even if all k-mers in K were distinct, there would still

be $\approx 4^{10}$ k-mers not present in K for each k-mer present in K. It is also impractical to work with such large k-mer sets, so the ones used are usually much smaller. k-mers of length 31 can also be stored as 64-bit integers.

2.1.3 Overlaps

When creating K from strings $s_1 \dots s_n$, we take all substrings of length k from each s_i. In case there are no duplicate k-mers and $|s_i| \gg k$, the sum of lengths of all k-mers in K is up to k times larger than the sum of all substrings. This comes from the fact that in the original strings, the k-mers were *overlapping*. An *overlap* between k-mers Q_1 and Q_2 is a suffix [1] of Q_1 that is also a prefix of Q_2. As there may be more such strings (for example, an empty one), we are only interested in the longest one of them, the *maximal overlap*.

We refer to the maximal overlap simply as the overlap and denote it by $\mathrm{ov}(Q_1, Q_2)$. For example, $\mathrm{ov}(\texttt{ATGC}, \texttt{TGCA}) = \texttt{TGC}$, while $\mathrm{ov}(\texttt{TGCA}, \texttt{ATGC}) = \texttt{A}$. Note that

$$\mathrm{RC}(\mathrm{ov}(Q_1, Q_2)) = \mathrm{ov}(\mathrm{RC}(Q_2), \mathrm{RC}(Q_1)).$$

Overlap graph

An *overlap graph* (OG) of K is a directed graph with k-mers from K as vertices. Each pair of vertices is connected by a weighted edge with weight equal to the length of the overlap between the vertices. Each vertex is also connected to itself with a self-loop. As even edges of zero length are allowed, the graph is complete and therefore requires $\Theta\left(|K|^2\right)$ space to be stored. Overlap graphs are used, for example, in the de novo genome assembly [Riz+19].

Hierarchical overlap graph

To be able to store the overlap graph efficiently, *hierarchical overlap graph* (HOG) was developed. It requires only linear space to encode all overlaps between pairs of k-mers from K [CR20]. It can be constructed in linear time from the Aho-Corasick automaton [Par+21]. The currently best-performing algorithm first creates EHOG (extended HOG) by removing vertices that cannot be visited by following failure links from the leaves of the tree. Then it further removes the vertices that do not encode maximal overlaps [Tal+24].

de Bruijn graph

A special OG subgraph called the *node-centric de Bruijn graph* contains only edges with weights of $k - 1$. The result is that each vertex can only have a constant number of in- and out-edges (up to four in uni-directional and eight in the bi-directional model), and the graph can be stored in linear space.

Note that the *edge-centric* version of the de Bruijn graph also exists, where the nodes represent overlaps (strings of length $k - 1$) and the edges represent k-mers.

Because shorter overlaps are not included in the graph, it does not store all the information from the OG. Despite this limitation, de Bruijn graphs are widely used in bioinformatics [CPT11; DJ22] and many other fields of study [HZB24].

[1] We only consider a *proper* prefix or suffix of string s, which must be shorter than s.

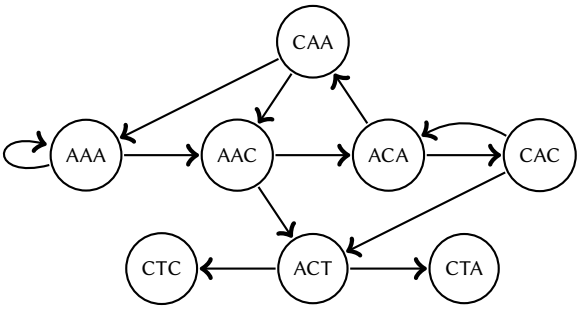


Figure 2.1 de Bruijn graph of k-mer set {AAA, AAC, ACA, ACT, CAA, CAC, CTA, CTC} with $k = 3$. Arrows mark overlaps of length $k - 1$.

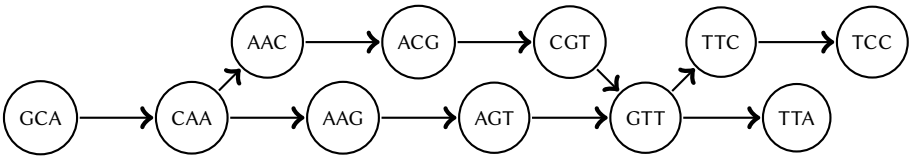


Figure 2.2 de Bruijn graph of k-mer set $K_{example}$ = {GCA, CAA, AAC, ACG, CGT, AAG, AGT, GTT, TTC, TCC, TTA} with $k = 3$. Arrows mark overlaps of length $k - 1$.

2.2 Textual representations of k-mer sets

Recall that the total length of k-mers in the set K can be up to k times larger than the initial data to be stored. This means that we need to compress the set to be able to use it efficiently.

A *superstring* representation of the set K is a string S with these properties:

1. Every substring of S of length k is a k-mer of K.
2. Every k-mer of K is a substring of S.

However, it is not always possible to create such S to store an arbitrary K. For example, if there is more than one $Q_i \in K$ such that $\forall Q_j \in K : |\operatorname{ov}(Q_i, Q_j)| < k - 1$.

There are two main approaches to solving this problem:

- Splitting K into m subsets $K_1 \dots K_m$ and encoding them separately with superstrings $S_1 \dots S_m$ (unitigs, simplitigs, and matchtigs).
- Relaxing the first property and encoding S as a masked superstring with binary mask M to distinguish the false positive k-mers.

2.2.1 Unitigs

Unitigs are defined as a set of superstrings S_i of k-mer sets K_i, with additional properties:

3. $\forall K_i, K_j : i \neq j \implies (K_i \cap K_j) = \emptyset$ (every k-mer is present in exactly one S_i),
4. $\forall Q_x$, if there is more than one Q_y such that $|\operatorname{ov}(Q_x, Q_y)| = k - 1$ (Q_x has an out-degree higher than one in the de Bruijn graph), Q_x must be the last k-mer of its superstring. Symmetrically, every k-mer with in-degree higher than one must be the first one of its superstring.

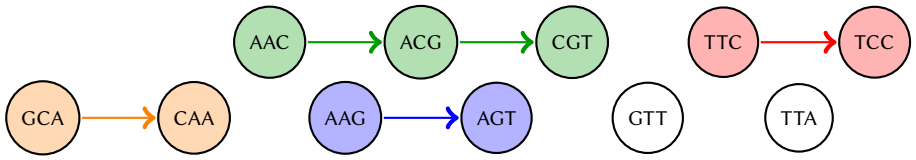


Figure 2.3 **Unitigs** representation of k-mer set $K_{example}$ visualized in its de Bruijn graph. Unitigs containing more than one k-mer are distinguished by color.

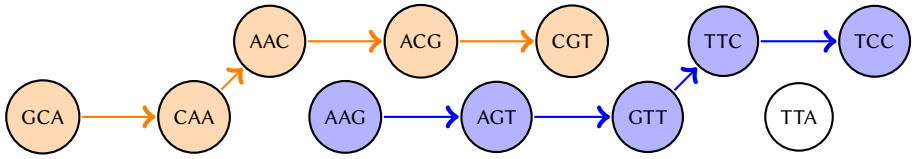


Figure 2.4 **Simplitigs** representation of k-mer set $K_{example}$ visualized in its de Bruijn graph. Simplitigs containing more than one k-mer are distinguished by color.

For optimal unitigs (with the smallest possible sum of lengths), it also holds that $\forall Q_x$ with exactly one Q_y such that $|\operatorname{ov}(Q_x, Q_y)| = k - 1$, Q_x is **not** the last k-mers in their (common) superstring.

Unitigs therefore represent non-overlapping paths in the de Bruijn graph which start and end at branching vertices. The fourth property guarantees that unitigs preserve the topology of the de Bruijn graph; they are referred to as compacted de Bruijn graphs. [Chi+14].

Optimal unitigs can be efficiently calculated using tools such as BCALM2 [CLM16], GGCAT [CT23], or CUTTLEFISH3 [KDP25]. This approach is effective for storing k-mer sets of single genomes, where de Bruijn graphs have a low number of branching nodes; its effectiveness increases with increasing k.

2.2.2 Spectrum-preserving string sets

Storing highly branching de Bruijn graphs (for example, when used to compress k-mer sets of pangenomes) results in a high number of short unitigs, which decreases the storing effectiveness.

Spectrum-preserving string sets (*SPSS*) or *simplitigs* [RM21; BBK21] relax the fourth property of unitigs, allowing them to be any vertex-disjoint paths in the de Bruijn graph. In most cases, the fourth property is not needed in applications, and if it is, unitigs can be recomputed from simplitigs without information loss.

Simplitigs can be efficiently computed greedily (using, for example, PROPHASM [BBK21]). In addition, optimal simplitigs can be computed in linear time using the Eulertigs algorithm [SA23].

2.2.3 Matchtigs

Matchtigs are a further generalization of simplitigs (also called repetitive SPSS — rSPSS) that relaxes the third property of unitigs, allowing each k-mer to occur in more sets K_i and also to occur more times in a superstring of K_i. This allows even better compression in some cases, especially when compressing bacterial pangenomes. Although matchtigs can be efficiently computed greedily with good results, only a polynomial-time algorithm is known for optimal matchtigs computation [Sch+23].

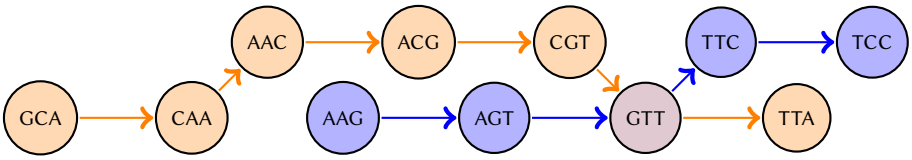


Figure 2.5 **Matchtigs** representation of k-mer set $K_{example}$ visualized in its de Bruijn graph. Matchtigs are distinguished by color. Note that k-mer GTT is present in both matchtigs.

2.3 Masked superstring

The masked superstring approach relaxes the first property of superstring representation by allowing 'false positive' k-mers to appear in S_i. This means that the whole set K can always be encoded by a single superstring S. However, the false positive k-mers (also called *ghost* k-mers) have to be distinguished from k-mers in K. This is done using a binary mask M [SVB23].

Definition 1 (Masked superstring)**.** *Masked superstring of k-mer set K is a pair of superstring S over alphabet Σ_{DNA} and mask M (a string) over alphabet $\{0,1\}$ such that:*

1. $|S| = |M|$ *(the mask is of the same length as the superstring),*
2. $\forall i \in [1,\dots,|S|-k] \;:\; M[i] = 1 \implies S[i,\dots,i+k] \in K$ *(all k-mers masked with 1 are in K),*
3. $\forall Q \in K, \exists i \;:\; M[i] = 1 \;\wedge\; S[i,\dots,i+k] = Q$ *(every k-mer from K is masked with 1 at least once),*
4. $\forall i \in [|S|-k+1,\dots,|S|] \;:\; M[i] = 0$ *(the last $k-1$ characters of the mask are always zeros).*

We denote masked superstring by (M,S) or simply by MS.

Note that not all occurrences of Q_i in S have to be masked with 1 — it is sufficient that at least one such occurrence exists. Depending on the further usage of MS for indexing, we can choose to mask any number of occurrences, but most often we mask either all or exactly one of them.

The maximal substring of M that contains only ones is called a *run of ones*. A *run of zeros* is defined similarly. Note that if M contains a run of zeros of length $\geq k$, we can remove all but the first $k-1$ characters from S and M corresponding to the run and (trivially and unambiguously) obtain a shorter $M'S'$ representing the same set K. We therefore consider all MS equivalent to their respective shortened versions and assume all runs of zeros in their masks to be at most $k-1$ characters long. This means that we can implicitly store the value of k with the length of the last run of zeros in the string.

The masked superstring is a generalization of matchtigs. This can be proven constructively — for any representation of K with matchtigs $S_1 \dots S_n$, we can construct S by concatenating all superstrings S_i and M by concatenating runs of ones of length $|S_i| - k + 1$, each followed by a run of zeros of length $k-1$.

Masked superstrings are used to store k-mer sets (as they achieve better compressibility than matchtigs [SVB23]) and as a base for k-mer set indexes [SVB24]. The generalization of properties 2 and 3 allows us to encode the presence of a k-mer in the set as any binary function of its occurrences in the mask [SVB25]. This can be used to perform various set operations with k-mer sets.

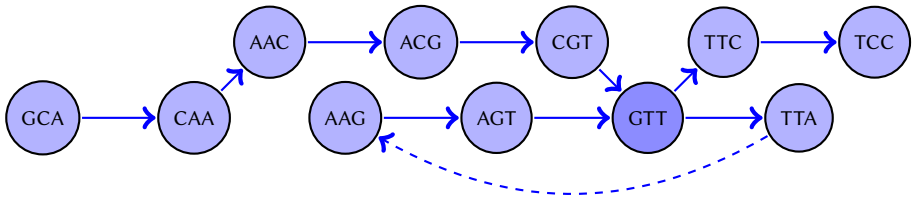


Figure 2.6 **(Masked) superstring** representation of k-mer set $K_{example}$ visualized in its de Bruijn graph. Solid arrows mark overlaps of length $k-1$. Dashed arrow marks an overlap of length $k-2$. Note that k-mer GTT appears two times in the resulting masked superstring.

2.3.1 Space complexity of storing the masked superstring

To achieve an efficient representation of K, our aim is to minimize the space required to store MS, which is simply the sum of the space required to store S and M.

We model the space complexity of S to be linear with its length, $|S|$. As most of the patterns present in the input data are not preserved during the construction of MS, we expect no predictable structure usable for compression in S [2]. The strictly bounded size of our alphabet allows us to use 2 bits per character, giving us a space complexity of $2|S|$ bits.

The space complexity of storing M depends on the structure of M and the encoding chosen. We present two of the most theoretically relevant encodings: run-length encoding and Elias-Fano encoding. They both use the fact that, typically, M contains a small number of long runs of ones [3]. When compressing pangenomes, the resulting mask usually contains more shorter runs than when compressing single genomes.

In practice, there are many highly optimized encodings that can be used to compress the mask. However, it is usually more practical to compress the whole MS at once. Compressors such as gzip, bzip2, lrzip, or xz use complex compressing techniques that can utilize the patterns present in MS to achieve an extremely efficient representation.

2.3.2 Efficient encodings for storing the mask

Run-length encoding

Run-length encoding (abbreviated as *RLE*) views M as an array of alternating blocks of ones and zeros. M is then stored as an array of numbers representing the lengths.

We assume that the zero runs in the resulting MS are of length at most $k-1$. This means that the space complexity of storing one length of a run of zeroes is $\mathcal{O}(\log k)$.

The runs of ones can have length of up to $|S|$. Storing the length of such a run requires $\mathcal{O}(\log |S|)$ space.

For each run of ones, there is exactly one run of zeros present in M. This holds trivially because each run of ones is only interrupted by one or more zeros, and the mask always starts with a run of ones and ends with a run of zeros.

Let R be the number of runs of ones in M. This gives us a total complexity of storing M with run-length encoding in the number of bits:

$$\approx R \cdot \left(\log_2(|S|) + \log_2(K)\right) \approx R \cdot \log_2(|S| \cdot K).$$

[2] This has also been proven experimentally [SVB23].

[3] We can also use the fact that the mask contains a small number of zeros, and encode their individual positions. However, since we expect an average length of run of zeros to be more than 2, encoding run lengths is more efficient.

The total complexity of storing MS (in bits) is then:

$$\approx 2\left(\log_2(|S|) \cdot R + |S|\right).$$

In practice, the effectiveness of the encoding depends on the number of runs of ones. A mask with a small number of long runs of ones is compressed more efficiently.

Elias-Fano encoding

Elias-Fano encoding (*EFE*) utilizes the fact that the starting positions of the runs form a strictly increasing sorted sequence of $\approx 2R$ integers of size $\mathcal{O}(|S|)$ [PV17].

It uses *gap compression* technique, which relies on storing the differences between integers in the sequence instead of their values (note that those differences are the lengths of runs). It also uses fewer bits per average-sized gaps than for storing the larger ones. The lower $\log_2(|S|/R)$ bits of the run length are stored explicitly, while the remaining bits are stored in unary encoding, which is optimal for geometric distributions.

EFE has been proven to be *quasi-succinct* [Vig13], which means that it provides compression close to the theoretical lower bound. It requires $\mathcal{O}\left(\log \frac{|S|}{R}\right)$ bits per run, which is better than RLE. The total complexity of storing MS is then:

$$\mathcal{O}\left(R \cdot \log \frac{|S|}{R} + |S|\right).$$

In practice, EFE requires $\approx \left(2 + \log_2 \frac{|S|}{2R}\right)$ bits per stored number, which is $\approx 2 \cdot \left(1 + \log_2 \frac{|S|}{R}\right)$ bits per run of ones. The factor of 2 comes from the fact that for each run of ones, we also need to encode the following run of zeros.

The real compression effectiveness depends on the distribution of gap lengths — distributions with less variation need less space to be stored. As with the RLE, masks with small number of long runs get encoded more effectively.

2.4 Masked superstring optimization

Finding an optimal (shortest) superstring S of K is NP-complete (in both uni-directional [GJ02] and bi-directional model [SVB23] for a sufficiently large value of k).

However, the optimization objectives for MS can be more complex than those for S, as they can also include the mask; therefore, any function of the form $f(S, M) \to \mathbb{R}$ can be used as an optimization objective. For example, if an objective is the number of runs in the mask, optimal MS can be computed in polynomial time [Sla24]. Other objectives can be NP-hard to compute and require the use of heuristics.

In Section 3.1, we prove that NP-completeness holds for the objective in the form $|S| + c \cdot R$ for every constant $c > 0$ that we later use to model the space complexity of storing the masked superstring.

2.4.1 Two-step optimization

The former work in the field of masked superstring optimization focused on two-step computation [SVB23]. In the first step, the superstring S is computed using a heuristic

method. In the second step, the mask M is optimized for S. This method allows to choose from several optimization criteria for M, including computing the least number of runs of ones.

The main advantage of the two-step optimization method is that it allows the mask for a given superstring to be re-optimized according to a different objective, which is usually the less time-consuming step.

The disadvantage is that the input of the second step and the overall quality of the result are highly dependent on the output of the first step. However, optimization in the first step does not consider the target of the second step of optimization, which can lead to significantly worse results for certain optimization criteria.

Computing the superstring

In this step, the aim is to compute the shortest possible superstring which contains all k-mers from K. As computing the shortest possible superstring for a set of strings (of non-trivial lengths) is NP-complete, approximation algorithms are used. So far, two main greedy heuristics have been used, called global and local [SVB23]. Both of them also work in the bi-directional model by removing the edges between a k-mer and its complement from the overlap graph and constructing two complementary Hamiltonian paths in the graph.

The *local greedy* algorithm is a generalization of PropHAsm [BBK21] with overlaps in the overlap graph considered instead of only the ones in the de Bruijn graph.

It constructs a superstring by repeatedly picking an arbitrary k-mer from K and extending it to the left (with a prefix) or to the right (with a suffix) as long as the respective overlap has at least the length specified with a parameter $k - d_{max}$. The k-mers used are then removed from K, and the resulting string is appended to the masked superstring created in the previous steps.

The local greedy algorithm can be implemented with linear time complexity using the Aho-Corasick automaton [SVB23] (see Section 2.4.2 for the explanation of the Aho-Corasick automaton). In practice, the hashing-based implementation is used, although its time complexity is exponential with respect to the parameter d_{max}.

The *global greedy* algorithm is based on the linear time approximation algorithm for the shortest superstring [Ukk90]. It works by iteratively constructing the Hamiltonian path by adding edges from the overlap graph in order of an increasing overlap length. During computation, each k-mer can have in-degree and out-degree at most 1. Edges that would violate this invariant or create a cycle are skipped. For each edge between k-mers Q_i, Q_j added to the path in bi-directional model, edge between $\mathrm{RC}(Q_j), \mathrm{RC}(Q_i)$ is also added to the complementary path.

The global greedy algorithm can be also implemented with linear time complexity using the Aho-Corasick automaton. The hashing-based implementation of global greedy used in practice has a linear expected time complexity for constant k.

For several datasets, the results of compressing the masked superstring computed using the global greedy algorithm are in practice better by $\approx$ 15% than the results obtained using the local greedy algorithm or the results obtained using matchtigs [Sla24].

Optimizing the mask

The mask can be optimized according to a number of objectives. The most practical ones include the minimization and maximization of the number of ones and the minimization of the number of runs of ones in the mask.

The minimization of the number of ones can be performed trivially in a single pass by assigning 1 to only one occurrence of each k-mer in MS. The maximization of the number of ones is very similar except that 1 is assigned to every occurrence of a k-mer. However, it cannot be done in a single pass, unless we assume that the first occurrence of each k-mer is masked with 1.

The minimization of the number of runs of ones in MS is an NP-complete problem, which is proven by a reduction from the set cover problem [SVB23]. The problem is solved using integer linear programming (ILP) [Sla24].

2.4.2 Aho-Corasick automaton

In the joint optimization algorithm proposed in Chapter 3, we use a modification of the *Aho-Corasick automaton* [AC75] data structure (abbreviated AC) to store k-mers of K [4].

AC is a data structure used to find the occurrences of strings in a text. It is created by extending the *trie* (prefix tree). AC contains nodes which represent the prefixes of strings and two types of oriented edges [5] (also called *links* of the respective nodes they start at):

- *Forward* (or *goto*) edges $u \to v$ connect pairs of nodes u and v where u is a proper **prefix** of v and $|u| + 1 = |v|$. The node v is called a *child* of u. Each non-leaf node has a number of forward edges between 1 and the size of the alphabet (4).
- *Failure* edges $v \to u$ connect pairs of nodes v and u, where u is the longest proper **suffix** of v.

We call the node representing the zero-length prefix a *root*. For each node, we define its *depth* as the length of the prefix represented by that node (or, equally, as the number of forward edges on a path from the root to the respective node). Nodes with a depth of k represent k-mers $Q_1 \ldots Q_n$ of K, we call them *leaves*. Nodes that are not leaves are called *internal* nodes.

We also define a *failure path* ϕ_i of a leaf Q_i as the path starting in Q_i and following the failure links until it reaches the root. Note that each node, except the root, has exactly one failure link and that a failure path contains at most k edges.

For each node, we define leaves *covered* by the node as a set of leaves that can be reached from the node by following the forward links.

[4] AC is also used in linear-time implementation of greedy algorithm for the shortest superstring problem [Ukk90].

[5] Some implementation of AC also use a third type of edges, *output* edges, which we omit in the algorithm since all our strings have the same length k.

Chapter 3

Targeted optimization of masked superstring

In this chapter, we present the target of our masked superstring optimization. We define the optimal masked superstring according to the objective function $\varphi(MS)$ in the form $|S| + c \cdot R$ and prove that obtaining such an optimal masked superstring is generally NP-complete in Section 3.1.

We then propose a heuristic algorithm for single-step masked superstring computation (joint optimization) in Section 3.2. The details of the implementation are described in Chapter 4.

3.1 Single-step (joint) optimization

We aim to optimize the space complexity of storing MS, with a mask stored in Elias-Fano encoding (see Section 2.3.2) which we model in the form of $|S|+c\cdot R$ for a constant c. Note that joint optimization of MS is a multi-objective (Pareto) optimization problem, as paying more attention to optimizing M results in longer (less optimal) S, while for shorter S, M is usually forced to contain more runs of ones.

To be able to directly compare the effects of both criteria on the resulting space complexity, we introduce the concept of the *run penalty* (see Section 3.1.2).

3.1.1 NP-completeness of joint optimization

Theorem 1 (NP-completeness of joint optimization)**.** *For any constant $c > 0$ and any value of $k > 4\log_4(|K|) + 5$, where K is a set of k-mers, computing an optimal MS of K according to the objective function $|S| + c \cdot R$ is NP-complete.*

Proof for even values of k. We reduce the NP-hardness from the NP-hardness of the problem of the shortest superstring of the ℓ-mer set L in the bi-directional model [SVB23], where $\ell = k/2$. We first transform the ℓ-mer set L into the k-mer set K, solve the joint optimization problem for K and obtain MS, which we transform into MS_L of L with optimal S_L.

We construct K in the following way: For each ℓ-mer L, we replace every character with digram according to the mapping μ:

$$\texttt{A} : \texttt{AC}, \texttt{C} : \texttt{AT}, \texttt{G} : \texttt{GC}, \texttt{T} : \texttt{GT}$$

We obtain a set of k-mers where all pairs of Q_i, Q_j have an overlap of even length, because characters `A` and `G` only appear at odd positions and characters `C` and `T` at even positions in k-mers. This also holds for complements of k-mers and therefore works in the bi-directional model as well. Therefore, no two k-mers have an overlap of length $k-1$. This means that the lowest number of runs of ones in M we can obtain is $|K|$.

However, we can also obtain $|K|$ runs by optimally arranging the k-mers (choosing the optimal permutation). If we used any k-mer not in K or used some k-mer from K more than once, the number of runs of ones in M would increase and the length of the resulting S would not decrease. This means that the optimal MS has exactly $|K|$ runs of ones and the optimality of MS depends only on the optimality of S. Solving the joint optimization therefore results in an optimal S, which can be directly transformed into the optimal S_L of L using the inverse mapping for each pair of characters in S.

Therefore, the shortest superstring problem for ℓ-mers (which is NP-hard for $\ell = k/2 > \log_4(|L|)$, see Section 2.4) can be solved by solving the problem of joint optimization of the objective function of the form $|S| + c \cdot R$ for k-mers, which means that joint optimization of such an objective function is also NP-hard for even values of k.

The solution to the decision version of the problem (whether an MS exists for which $\varphi(MS) \leq X$ for a given X). can be verified in polynomial time and with a certificate of polynomial length. This means that the problem is also in NP and therefore is NP-complete. □

Modification of the proof for odd values of k. We modify the proof for odd values of k. First, we choose ℓ to be $(k-5)/4$ if $k \equiv 1 \pmod 4$ or $(k-3)/4$ if $k \equiv 3 \pmod 4$. The other values of k are covered by the previous proof.

Using the mapping μ, we encode each ℓ-mer l from L into 2ℓ-mer l'. We then construct k-mers of K by concatenating l', `CCC`, l' or l', `CCCCC`, l'. This results in k-mers of length $4\ell + 3$ or $4\ell + 5$, respectively.

We observe that no two k-mers have an overlap of length $> k/2$, which also holds in the bi-directional model. Therefore, no two k-mers have an overlap of length $k-1$. The rest of the proof follows the same as for even values of k.

The joint optimization problem is therefore NP-complete for all values of

$$\ell = (k-5)/4 > \log_4(|L|) \implies k > 4\log_4(|K|) + 5$$

. □

3.1.2 Run penalty

The run penalty defines the length increase of $|S|$ equivalent to introducing one run of ones in M, with respect to the resulting space complexity.

We assume that EFE is used to compress M (see Section 2.3.2). We therefore define the objective function $\varphi(MS)$:

Definition 2 (Objective function)**.** *The objective function of MS with the superstring of length $|S|$ and mask M with R runs of ones is:*

$$\varphi(MS) = 2 \cdot \left(|S| + R \cdot \left(1 + \log_2 \frac{|S|}{R} \right) \right)$$

Definition 3 (Optimal run penalty)**.** *We define the optimal run penalty P^*_{run} for k-mer set K as a ratio of*

- *the overall space increase by introducing a new run of ones into the mask to*
- *the overall space increase by extending the superstring by one character*

for MS of K which has the minimum value of the $\varphi(MS)$:

$$P^*_{run} = \frac{2 \cdot \left(1 + \log_2 \frac{|S|}{R}\right)}{2} = 1 + \log_2 \frac{|S|}{R}$$

where $|S|$ and R are properties of the optimal MS.

This means that introducing a new run to M has the same effect as extending S by P^*_{run} k-mers which were already used in S (increasing the overall space to store MS by $2P^*_{run}$ bits).

In practice, we are not able to estimate P^*_{run} in the joint optimization, since the computation of the optimal MS is NP-complete. Even if we could compute a good approximation of the optimal MS, we cannot estimate P^*_{run} before the computation, as it depends on the structure of the MS computed with respect to the penalty itself.

This limits us to using the *approximate run penalty* P_{run}. We can infer its value from the optimal penalties for other similar input genomes or try to compute MS with several values of P_{run} to find the best one (see Chapter 5).

Note that the concept of joint optimization implies that the M produced for given S is optimal in terms of the number of runs of ones (given nonzero run penalty).

3.2 Algorithm for joint optimization

The algorithm uses a global greedy heuristic method. It consists of two main parts:

1. Building the AC of K — building part.
2. Computing MS using the AC — searching part.

In the searching part, we try to greedily connect k-mers into two complementary Hamiltonian paths, similarly to the global greedy algorithm from [SVB23]. We create subpaths ending with unconnected k-mers and connect those subpaths until only the two remain. Each k-mer is connected at most once, using the depth first search (DFS) in the AC to find another k-mer that is not connected or lies at the start of the subpath.

The maximum distance in AC between the k-mers to connect is limited to $k-1+P_{run}$ — connecting the k mers with a larger distance would increase $\varphi(MS)$ more than connecting two k-mers with zero overlap and inserting a run of zeros of length $k-1$. The algorithm tries to repeatedly connect all unconnected k-mers with an increasing distance limit, starting from 1.

3.2.1 Building the Aho-Corasick automaton

We build the AC (see Section 2.4.2) iteratively. In each iteration, we create all nodes of the same depth (we call the set of these nodes a *layer*). The layers are added in descending order by depth, starting with the leaves of depth k, which correspond to the k-mers of K. In the iteration, we also compute the leaves covered by every node in the current layer and create all failure links ending in the current layer.

To build a layer of depth $d < k$, we divide all the nodes of depth $d + 1$ into groups with the same prefix of length d. For each group g of nodes, we create a new node n in the current layer and set leaves covered by n to a union of all sets of leaves covered by nodes of g.

We also maintain a set F_d of suffixes of length d of the leaves. F_d represents all possible extensions of all failure paths in the current layer. For each failure path ϕ_i, we keep track of the last node f_i that extended ϕ_i in previous layers. At first, the f_i for each i is set to Q_i — the leaf to which the path corresponds. When creating a node n with the prefix p, we check whether F_d contains p. If it does, we add the failure link from the corresponding node f_i to n and update f_i to n.

To efficiently build all the layers, we first sort the leaves. This is only needed in the first layer, as prefixes of sorted leaves are also sorted. To efficiently search in F_d, we sort it in every iteration.

Note that this process does not create failure links from all the nodes, but only from those that belong to at least one failure path. This is intentional since we will only visit the nodes on the failure paths during the search.

3.2.2 Computing the masked superstring

Each leaf in the layer of depth k is equivalent to a k-mer of K. We define a leaf *chain* c as a sequence of k-mers, where each pair of k-mers Q_i, Q_{i+1} has an overlap of $k - 1$. Note that not all k-mers in c need to also be contained in K. We call k-mers not contained in K *ghost* k-mers. During computation, we maintain a set C of chains, starting with $C = K$ (each chain consisting of a single leaf). We refer to a chain by the index of the last leaf in the chain (we later explain that every chain must end with a leaf).

Every leaf Q_i is called *unused* if there exists a chain $c_j \in C$ such that the first k-mer of c_j is Q_i. All other leaves are called *used*.

Similarly, every leaf Q_i is called *unextended* if there exists a chain $c_j \in C$ such that the last k-mer of c_j is Q_i. All other leaves are called *extended*.

The occurrence of a k-mer in a chain can be of two types: *scaffold* or *filler*. Scaffold k-mers are always masked with ones. Filler k-mers can be masked with either one or zero and do not need to be present in K. At first, there are only scaffold k-mers in C. All k-mers added later are filler and each of them increases the length of $|S|$ by one.

During the computation, several invariants hold for C:

1. Each leaf has exactly one occurrence in C marked as scaffold, all other occurrences of the leaf are marked as filler. Therefore, the number of scaffold leaves in C does not change.

2. All occurrences of non-leaf (ghost) k-mers are marked as filler.

3. Every chain starts and ends with a scaffold leaf.

4. There can be between 0 and $k - 1 + P_{run}$ filler k-mers between any two scaffold leaves in the chain.

5. If a leaf Q_i is unused, there is exactly one occurrence of Q_i in C (which is scaffold, not filler).

3.2.3 Merging the chains

In the algorithm, we try to *merge* chains into each other until only one chain remains. To merge chain c_i into chain c_j, we use chain x (possibly empty). We determine the content of x by searching the AC. As a result, c_j becomes a concatenation $c_i x c_j$.

After merging, the leaves of c_i and c_j remain marked the same as before. All k-mers from x are marked as filler. The chain c_i is removed from C.

We define a *penalty* of merging as a sum of:

- The length of x (the penalty for an increase in the space required to store S),
- The number of runs of zeros in the mask of x, times P_{run} (the penalty for introducing new runs of ones in M).

Note that we can always choose x so that the number of runs of zeros is at most one, for example by masking all k-mers from x with zeros.

3.2.4 Phases of computation

The superstring computation is split into phases. Each phase is defined by the *penalty threshold* p_i — the maximum penalty that chain merging can introduce. The penalty threshold starts at 1 and increases by one in each phase.

The merging of two chains with zero overlap of respective unextended and unused leaves using chain x consisting of $k - 1$ k-mers results (in the worst case) in an overall increase of $\varphi(MS)$ by a value of $k - 1 + P_{run}$. The value of $k - 1 + P_{run}$ is also the maximal penalty threshold, since we are always able to merge two chains in this way. This means that the algorithm will have at most $k - 1 + P_{run}$ phases.

In each phase, we iterate through all the unextended leaves. For each unextended leaf Q_i, we try to find an unused leaf Q_j such that:

- Q_j does not belong to the same chain as Q_i.
- The penalty of merging chain Q_i into chain containing Q_j does not exceed the current penalty threshold.

To find a suitable Q_j, we use the DFS search from Q_i. Each node is visited at most once during the single DFS search — its next occurrences are skipped. We refer to adding node n to the DFS stack as *extending* the search with n. In each visited node, we consider two sets of nodes to extend the search with:

- The leaves covered by the current internal node (we ignore this option if the current node itself is a leaf).
- The failure node of the current node.

We end the search when we find the first unused leaf.

Note that the penalty for every merging in phase i is always p_i (the maximum allowed). If the penalty was smaller, the respective chains would be merged in the previous phase.

The Figure 3.1 shows an example of the DFS search.

3.2.5 Time and space complexity

Building the AC requires $\mathcal{O}(|K| \cdot k)$ time.

We first need to sort k-mers from K. This can be achieved in $\mathcal{O}(|K| \cdot k)$ time by using a stable radix sort or in $\mathcal{O}(|K| \cdot \log(|K|))$ time by using other sorting algorithms (note that $\mathcal{O}(\log(|K|)) \in \mathcal{O}(k)$, see Section 2.1.1).

In each of k iterations, we create $\mathcal{O}(|K|)$ new nodes. We also need to create and sort F_d. This can be achieved in $\mathcal{O}(|K|)$ time from F_{d+1} using a stable bucket sort.

Searching requires $\mathcal{O}\left(|K|^2 \cdot k(k + P_{run})\right)$ time. In each of the $\mathcal{O}(k + P_{run})$ phases, we try to extend $\mathcal{O}(|K|)$ leaves. Extending a leaf requires us to visit up to $\mathcal{O}(|K| \cdot k)$ nodes. Note that in practice, most of the leaves are extended in the first phase. However, the exact ratio depends on the content of K.

Storing the AC requires $\mathcal{O}(|K| \cdot k)$ space. Searching only requires additional data structures with $\mathcal{O}(|K|)$ space complexity.

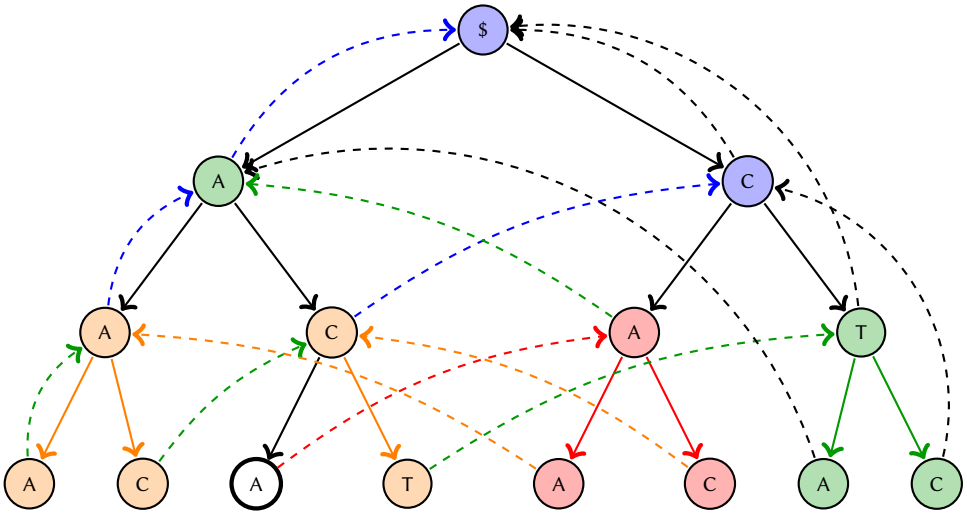


Figure 3.1 Example of all possible nodes visited during the DFS search for the leaf to extend the leaf ACA in the AC tree of k-mer set {AAA, AAC, ACA, ACT, CAA, CAC, CTA, CTC} with $P_{run} = 1$.

Goto links are marked with the solid lines, failure links with the dashed lines. Each node and edge is marked with red, orange, green or blue color according to the phase it is used in for the first time, also equal to the minimal penalty threshold required to do so.

In the first phase (red), only node CA is visited with its two leaves.

In the second phase (orange), also the nodes AA and AC are visited by searching from the red leaves. Their leaves which were not visited yet are also visited.

In the third phase (green), also the node CT with its leaves is visited from leaf ACT and node A is visited from node CA. Note that visiting node A requires penalty threshold 3, the sum of 1 for the failure link of ACA and 2 for the failure link of CA, as it might require us to start a new run.

In the fourth phase (blue), also the node C is visited from the node AC and the root is visited from the node A (which only increases the penalty by 1, as reaching the node A already required a run penalty).

Note that we would omit the fourth phase in the actual search, as we are always able to visit the root (and therefore all the leaves of the tree) in the phase number $k + P_{run}$, which is equal to 4 in this case.

Chapter 4

Efficient implementation of optimization algorithm

In this chapter, we discuss the optimization methods used in the practical implementation of the algorithm in Chapter 3.

We develop a method for efficient AC tree representation called *Cutted-Sorted AC* (CSAC). We further improve it by developing an even more efficient method called *Leaf-Only AC* (LOAC) based on tree representation in CSAC. We then use several heuristics to speed up the search.

4.1 Efficiently representing the tree

4.1.1 Cutted-Sorted Aho-Corasick automaton

This method tries to efficiently store the relevant inner nodes of the AC tree. If there is not enough memory to store the whole tree, it ”cuts” the layers with the smallest depths. It needs k-mers to be sorted. We sort them as the first step of the algorithm.

The original AC stores data that are not required for our specific use case. In the implementation, we are able to build and use the tree without using:

- links to the parent nodes,
- failure links of nodes which are not part of a failure path,
- links to child nodes (we only need to get the leaves covered by a node and only search in direction of failure links).

Therefore, each node is only stored as a pair of numbers:

- index of the first leaf covered by the node,
- index of the failure node of the node.

The leaf nodes store the index of their complement leaf instead of the covered leaf index.

The tree is stored as a sequence of layers sorted by depth. In each layer, nodes are sorted lexicographically. This allows us to create the tree in $\mathcal{O}(|K| \cdot k)$ time. In each layer, we traverse all the nodes, and group together those with common prefix of length one less than current depth (there are always at most four of those nodes;

thanks to the layers being sorted, they all appear consecutively). We then create a new node in the current layer, set its first covered leaf, and if it prolongs any failure path, extend that path by the node. If more failure paths are extended by the same node, we delete all those paths except the first one.

Possible extensions of failure paths in each layer can be sorted in linear time. This is possible because the previous set of failure paths was also sorted, and we use stable sorting into four buckets (based on the new first letters of current suffixes). Initially, the nodes on the failure paths are the leaves of K, which we sort at the beginning in $\mathcal{O}(|K| \cdot k)$ time.

Depending of the total number of k-mers, we are able to choose the right data type to store the indexes, allowing us to use 32-bit integers in case of smaller datasets.

Cutting the tree

As this approach requires $\mathcal{O}(|K| \cdot k)$ space, we are not always able to store the entire tree (all layers). In that case, we store the layers with higher depths and "cut" (ignore) the ones with lower depths. This results in lower precision and overall worse results in terms of the space required to store MS, since even low-depth layers are visited during computation (though not as much as layers with higher depths).

We are able to estimate the total space required to store each layer with a total number of leaves. However, there are usually fewer nodes in the layers, resulting in about half of the space being actually used. This can be solved by dynamically deciding if we should cut the tree after building each layer, depending on the size of the last layer and the available space.

Pros and cons of CSAC

The main advantage of CSAC is that we precompute everything we would need to search the tree later. This helps to speed up the operations in the searching part, which usually takes longer as it has a worse time complexity than the building part.

However, the space requirements of the CSAC are too large, which limits its practical usage. Tree construction times for larger values of k impact the running times of the entire algorithm. Also, most parts of the tree (namely the ones with lower depths) are not used so often in the search.

4.1.2 Leaf-Only Aho-Corasick automaton

To solve CSAC problems, we develop another tree representation method, called LOAC. In this approach, we consider not to construct the tree at all and use it only implicitly over the sorted sequence of leaves. The implicit tree is the same as that created in CSAC (except that no layers need to be cut), which simplifies the analysis.

The key observation is that we can get the sequence of an inner node by knowing its depth d and any of the leaves Q it covers (it is a simple prefix of length d of Q).

We are also able to find the first leaf covered by a node by binary searching the sorted leaves, instead of storing the index in the node. This results in a search time complexity increased by a factor of $\log(|K|)$, but in Section 4.2.1 we provide a method that reduces the time to a constant on average.

Also, we do not need to store the index of the next node of the failure path — we can always find the respective node by checking all suffixes of the current node (from

the longest one) and trying to find at least one leaf with the corresponding prefix. If any such leaf exists, the node would have been created in CSAC and put on a failure path. Otherwise, such a node does not cover any leaves and can be skipped in the search.

Total space complexity

LOAC only stores the sequence of leaves and other constant information about each leaf. It does not create any layers of nodes. The total space complexity is therefore $\mathcal{O}(|K|)$.

4.2 Speeding up the search

The theoretical time complexity of searching the tree is impractically large. We further modify the search using several heuristics to improve practical running times.

4.2.1 Reducing the binary search intervals

In practice, binary searching in an array of millions or billions of leaves is slow because of cache misses. However, using the space $\mathcal{O}(|K|)$, we can create an index consisting of all possible prefixes of length $\log(|K|)$. For each prefix, we store the first index of a leaf with the given prefix. This allows us to skip most of the binary search by only running it on an interval of the leaves with respective prefixes.

Note that there are $\mathcal{O}(|K|)$ intervals, so there are on average $\mathcal{O}(1)$ leaves to search through in each interval. We conjecture that the distribution of k-mers in the intervals is normal, which means that the chance of searching in the interval with a size larger than a constant is exponentially low, so the average search time remains constant.

4.2.2 Penalty drop cut-off

Penalty drop δ_{ji} of (an already used and extended) leaf Q_j during the search to extend leaf Q_i is the difference between the penalty threshold p_i in the current iteration and the amount of penalty that would satisfy if the leaf Q_j could be used to extend Q_i (the "remaining" penalty if we were able to extend Q_i with Q_j.

If there is a leaf Q_l that could be used to extend Q_j with penalty δ_{ji}, this leaf can also be used to extend Q_i with penalty p_i.

For each Q_j, we store the highest value of penalty drop δ_j with which the leaf was used to extend the search so far during the search. We do not extend the search by any Q_j with δ_j higher than or the same as the remaining penalty.

The idea is that unused leaves are sparse, so if we find the suitable leaf by extending the search through the current leaf, there is a high chance that searching with the same or lower allowed penalty drop from the current leaf will not result in any unused leaf being found. If we do not find any unused leaf this way, it is almost certain that searching from the current leaf next time would not yield any unused leaf.

This approach slightly reduces precision, as there might be several unused leaves that could be reached by searching through the same leaf (although they are probably possible to reach in many other ways).

However, it massively improves search speed, as during the whole computation, each δ_j only increases, and the total number of increases is limited to $k + P_{run}$. This means that every leaf is used to extend the search at most $k + P_{run}$ times during the algorithm.

4.2.3 Skipping more leaves at once

Every leaf might be checked for penalty drop many times, which is inefficient. We further improve the search by skipping some of the leaves.

For every leaf Q_i, we store an *skip-to index* ϱ_i of the next leaf Q_j with which we had extended the search (because δ_j was high enough) the last time we rejected to extend the search with Q_i.

When we reject to extend the search with leaf Q_i, we skip directly to ϱ_i. If ϱ_i is not suitable to extend the search either, we skip to ϱ_{ϱ_i} and continue until we find a suitable leaf or reach the end of the interval. For each of the rejected leaves, we then update its skip-to index to the index of the first suitable leaf.

If we increase δ_i of leaf Q_i, we reset ϱ_i to $i + 1$.

4.2.4 Other minor optimizations

Choosing the next leaf to extend

We store the unextended leaves in an array. Before searching, we shuffle the array. This slightly improves the result by prevents us from using only the lexicographically smaller leaves in the chains and having to use the remaining ones in a less effective way later.

We try to maximize the number of leaves extended in the first iteration and minimize the number of chains created in the process. To achieve this, we store the last used leaf and if it is unextended, we use it as a next leaf we try to extend. Otherwise, we use the next unextended leaf from the list.

After each iteration, we delete the extended leaves from the list. This allows us to skip checking large amounts of extended leaves in the next iterations.

Fast access of the most used layer

We further improve the retrieval time for the failure nodes with depth $k - 1$, by storing the pre-computed indexes of the first leaves covered by these nodes. The failure nodes in this layer are used the most, as extending the leaves through the nodes with depth $k - 1$ does not create new runs of ones in M.

Depths with no unused leaves

For each leaf, we store the lowest depth for which there are no unused leaves with the same prefix as the leaf. This is equivalent to the smallest possible depth of the CSAC node without unused leaves. This prevents us from checking whether any leaf covered by the node is unused. Instead, we can directly continue the search from the leaves.

4.3 Implementation in the bi-directional model

4.3.1 Storing the complements

If we work in the bi-directional model, we store both the k-mer and its reverse complement as a leaf. When creating the tree, we need to compute the index of complementary leaf for each leaf in the tree.

This is possible in $\mathcal{O}(|K| \cdot k)$ time by pairing the indexes of the leaves with their complementary k-mers, sorting by the k-mers and assigning the indexes of the complements in order.

4.3.2 Computing the masked superstring with complements

During computation, we treat k-mers and their complements as equivalent. This means that whenever a leaf Q_i is extended by Q_j and Q_j is marked as used, $\mathrm{RC}(Q_j)$ is also extended by $\mathrm{RC}(Q_i)$ and $\mathrm{RC}(Q_i)$ is marked as used. This creates a pair of complementary chains for every chain, similarly as in the global greedy algorithm used in the first step of the two-step optimization.

We also ensure that the leaf we try to use is not a complement of the leaf we try to extend, as this would create cycles in the chains. We simply skip such a leaf and continue the search.

Computing in the bi-directional model also implies that if we use an even value of k, we need to handle k-mers which are their own reverse complements. The practice that has proven to be the best for us is to keep both k-mer Q_i and $RC(Q_i)$ (with the same value) as leaves and to ensure that each of them has the other marked as its complement. Other mechanisms of the algorithm ensure that no cycles are created during the searching part.

Chapter 5

Experiments and results

In this chapter, we present the comparison of the proposed method with the KmerCamel two-step optimization method. We describe the implementation details, the datasets used for comparison, and the results obtained.

Our target is to experimentally evaluate the best run penalties for our datasets, infer the dependency of the optimal run penalty on dataset properties, verify that our implementation yields better results in terms of masked superstring compressibility, and quantify the improvements.

5.1 Framework used for implementation

We developed the implementation inside KmerCamel, an open-source software tool for the computation and optimization of masked superstrings. KmerCamel is available under the MIT license on GitHub (`https://github.com/OndrejSladky/kmercamel`).

KmerCamel is written in C++. k-mers are represented as integers (with each letter represented by two bits) with a maximum value of k being 127. Although this may seem like a limitation, the use of k-mer sets with $k = 31$ is enough for most practical purposes (see Section 2.1.2). This also allows operations with k-mers to be fast both theoretically and practically.

Our implementation is available in the GitHub repository (`https://github.com/Jajopi/Plachy-bc-thesis`). It is based on KmerCamel commit `6bdbe33` [1] from `2024-11-03`.

From KmerCamel, we use input processing functions (for reading command line parameters and input from FASTA files), libraries for efficient representation of k-mers as large integers, and several basic functions for operations with k-mers (such as loading the initial k-mer set and turning it into an `std::vector`, and efficient implementations of computing prefixes, suffixes, and complements). We also use script `verify.py` to check the correctness of our implementation against algorithms from KmerCamel, and makefile tools for building.

[1] `https://github.com/OndrejSladky/kmercamel/commit/6bdbe33`

5.2 Experimental setup

Our implementation of LOAC was compared with the KmerCamel implementation of the global greedy algorithm followed by the mask optimization algorithm for minimizing the number of runs (denoted further as GGMO — global greedy + mask optimization).

We measured the execution time and the maximum memory requirements of the computation. We also measured the length ($|S|$) of the resulting superstring and the number of runs of ones (R) in the resulting mask. For each set of parameters, we computed the objective function $\varphi(MS)$ of length and number of runs (see Definition 2).

We also compressed the resulting MS with five commonly used file compression tools: `gzip` (1.12), `bzip2` (1.0.8), `xz` (5.4.1), `lrzip` (0.651), and `zstd` (1.5.4). For each tool, we used the highest possible compression level. We measured the sizes of the resulting files in bytes.

We performed two types of measurements:

- Searching for the optimal run penalty — we computed MS with all run penalty values between 0 and k for $k \in \{23, 31\}$ for several datasets. For each dataset and k, we determined the best run penalty.

- Testing the algorithm for different values of k — we computed MS for values of $k \in \{23, 31, 45, 63, 95, 127\}$. For larger datasets, large values of k were omitted, as computation would require an impractical amount of time. For datasets created with reads of a specific length, it was also not possible to use k greater than the length of the unitigs obtained from the reads.

5.2.1 Technical specifications

The benchmarks were performed on an AMD EPYC 7302 (3 GHz) server with 251 GB of RAM and SSD storage, using a single core for each program.

The running time of the program [2] and the memory requirements [3] were measured using the standard UNIX utility GNU Time (version 1.9).

5.2.2 Commands used

GGMO

For two-step optimization (from [SVB23]), the implementation of the global greedy algorithm from KmerCamel was first used with the command:
`./kmercamel -p <input> -k <k> [-c]`.

The `[-c]` argument was used for the computation with complements.

The resulting masked superstring was then used as an input for the mask optimization algorithm from KmerCamel with the command:
`./kmercamel optimize -a runs -p <input> -k <k> [-c]`.

[2] Total number of CPU-seconds that the process spent in user mode.
[3] Maximum resident set size of the process during its lifetime, in Kbytes.

LOAC

For joint optimization, our implementation of LOAC was used with the command: `./kmercamel -a loac -p <input> -k <k> [-c] [--run-penalty <rp>]`.

The `[-c]` argument was used for the computation with complements.

5.2.3 Datasets

The whole genomes of commonly used model organisms were downloaded in FASTA format from NCBI datasets:

- *S. cerevisiae* (baker's yeast) [4] with genome size of $\approx 1.2 \cdot 10^7$ bp.
- *A. thaliana* (small plant) [5] with genome size of $\approx 1.2 \cdot 10^8$ bp.
- *D. melanogaster* (fruit fly) [6] with genome size of $\approx 1.4 \cdot 10^8$ bp.
- *C. elegans* (nematode worm) [7] with genome size of $\approx 10^8$ bp.

Pangenome datasets were downloaded from *Genomic datasets used for evaluation of k-mer representations and indexes* [VBS25] [8]. The respective sources are listed below:

- The *N. gonorrhoeae* (bacteria) pangenome of size $\approx 4 \cdot 10^7$ bp was downloaded from RASE DB N. gonorrhoeae [9]
- The *S. pneumoniae* (bacteria) pangenome of size $\approx 8 \cdot 10^7$ bp was downloaded from RASE DB S. pneumoniae [10]
- The *SARS-COV-2* (coronavirus) pangenome sc2-pg of size $\approx 4.3 \cdot 10^8$ bp was downloaded from *GISAID* [11] (version 2023/01/23).
- The metagenomic sample SRS063932 (Illumina raw reads) of the human microbiome with accession SRX023459 of size $\approx 8.8 \cdot 10^8$ bp was downloaded from the NIH Human Microbiome Project [12].
- The *E. coli* (bacteria) pangenome of size $\approx 5.5 \cdot 10^9$ bp was obtained from the *Phylogenetically compressed 661k collection* [13]. High-quality filtering was applied.

5.3 Results and discussion

The results presented are shown in Appendix A (tables) and Appendix B (plots). The exact data from all measurements performed are stored in the GitHub repository [14] in files `results.txt` and `results-penalty.txt`.

[4] `https://www.ncbi.nlm.nih.gov/datasets/genome/GCF_000146045.2/`
[5] `https://www.ncbi.nlm.nih.gov/datasets/genome/GCF_000001735.4/`
[6] `https://www.ncbi.nlm.nih.gov/datasets/genome/GCF_000001215.4/`
[7] `https://www.ncbi.nlm.nih.gov/datasets/genome/GCF_000002985.6/`
[8] `https://zenodo.org/records/14722244`
[9] `https://github.com/karel-brinda/rase-db-ngonorrhoeae-gisp`
[10] `https://github.com/c2-d2/rase-db-spneumoniae-sparc/`
[11] `https://gisaid.org/`
[12] `https://www.hmpdacc.org/hmp/HMASM/`
[13] `https://doi.org/10.5281/zenodo.4602622`
[14] `https://github.com/Jajopi/Plachy-bc-thesis`

5.3.1 Searching for the optimal run penalty

To determine the optimal run penalty (see Definition 3), we would need to compute the optimal MS, which is NP-complete. We therefore hypothesize that the results of LOAC are a good approximation of the optimal results for all values of run penalty used. This allows us to approximate the optimal run penalty as the one with which LOAC performs the best (according to $\varphi(MS)$, see Definition 2).

From our measurements, we conclude that for $k = 31$, the optimal run penalty for genomes is ≈ 12, while the optimal run penalty for pangenomes is ≈ 7.

For $k = 23$, the optimal run penalty for genomes is ≈ 11, while the optimal run penalty for pangenomes is also ≈ 7. The difference is probably due to the fact that for pangenomes, the optimal MS contains a larger number of runs, resulting in a lower value of $\frac{|S|}{R}$ in the formula for $\varphi(MS)$.

Genomes

The value of 12 is optimal for $k = 31$ and genomes of size $\approx 10^8$ base pairs, such as *A. thaliana* in Figure B.2, *C. elegans* in Figure B.3 or *D. melanogaster* in Figure B.4.

For smaller genomes (such as *S. cerevisiae* in Figure B.1 or *S. pneumoniae* that we did not include in the displayed data), this value is slightly higher (≈ 13).

For larger genomes, such as the human or mouse genomes, further measurement would be needed to observe whether the optimal penalty decreases or increases with increasing genome size.

For $k = 23$, the optimal run penalty was lower by 1 for each dataset. The differences in the values of $\varphi(MS)$ when using run penalties 11 and 12 are so small that we can ignore this difference and use 12 even for $k = 23$.

Pangenomes

For pangenomes (see Figure B.5, Figure B.6, Figure B.7, Figure B.8, and Figure B.9), the size range is much larger (from $\approx 10^7$ to $\approx 10^9$ base pairs); therefore, we can conclude that the values of ≈ 7 for both $k = 31$ and $k = 23$ are good approximations of P^*_{run} for pangenomes.

We also conclude that the difference between the optimal penalty for genomes and pangenomes is related to the different structure of their respective de Bruijn graphs, with the graphs of pangenomes having higher average degrees of nodes. More research with pangenomes differing in the average degree of nodes would be needed to further investigate the dependence.

Relation to two-step optimization and matchtigs

The results of GGMO are closest to the results of LOAC with run penalty 0. However, when computing the MS with LOAC with run penalty 0, the resulting mask is not optimal and has to be optimized as in the two-step optimization.

The run penalty $k - 1$ corresponds to matchtigs, where each run of zeros has a length of $k - 1$. Although LOAC does not compute the optimal matchtigs, we expect the results to be very close. As GGMO gives better results than matchtigs in terms of $|S|$ (according to [SVB23]), we conclude that the results of the computation with LOAC are better in terms of compressibility than both the masked superstring computed using a two-step computation and matchtigs.

5.3.2 Testing the algorithm for different values of k

The LOAC was run with optimal run penalty for each dataset, as it was determined in the previous search.

For higher values of k, the de Bruijn graphs of the genomes have lower average degrees of nodes; therefore, the difference between the results of LOAC and GGMO decreases with increasing k.

Genomes

The improvement of LOAC against GGMO for genomes is very low (as genomes are much easier to compress effectively even using simple techniques such as unitigs) and is therefore not of particular interest. As we can see in Table A.1, the actual improvements for our datasets were between 0.1% and 0.3%.

However, when comparing the practical compressibility of our datasets, the improvements were better, between 0.5% and 1.2% for $k = 23$.

The number of runs of ones in the resulting *MS* computed using LOAC was about 30% smaller for all datasets.

For genomes, LOAC was faster; in most cases, it took only between 40% and 60% the time required for GGMO. The details are in Table A.2.

For some datasets, LOAC used $\approx$ 15% less memory, but for *C. elegans* and $k = 63$, it used 20% more. See Figure B.10, Figure B.11, Figure B.12, and Figure B.13 for higher values of k.

Pangenomes

The improvements for pangenomes were better than for genomes. For $k = 23$, the improvements of LOAC were 4% for *SARS-COV-2* pangenome and between 0.4% and 0.7% for other datasets. For $k = 31$, they were 3% for *SARS-COV-2* pangenome and between 0.25% and 0.35% for other datasets (see Table A.3).

The improvements in practical compressibility were even better, about 5.5% for *SARS-COV-2* pangenome, 0.5% for the human microbiome metagenome, and between 1.7% and 3% for other pangenomes.

Unlike genome datasets, LOAC was between 1.5 and 5.5 times slower for pangenome datasets (see Table A.4). With increasing k, the relative speed was even worse. See Figure B.14, Figure B.15, Figure B.16, Figure B.17, and Figure B.18) for higher values of k.

The number of runs of ones in the resulting *MS* computed using LOAC was about 10% smaller for all datasets, except for *SARS-COV-2 pangenome* dataset, where the resulting mask had only about 50% of runs of ones compared to the one computed with GGMO.

Conclusion

In this thesis, we have developed a new method for the joint optimization of masked superstrings representing k-mer sets.

In Chapter 3, we have proven that the optimization of masked superstrings with an objective of the form $|S| + c \cdot R$ is NP-complete for sufficiently large values of k. We have proposed an objective function $\varphi(MS)$ of this form to model the space complexity of masked superstrings stored in the Elias-Fano encoding.

We have then proposed an efficient greedy-based heuristic algorithm for the joint optimization of masked superstrings considering $\varphi(MS)$ and implemented it within the framework KmerCamel🐫. We call the implementation Leaf-Only Aho-Corasick automaton or LOAC (see Chapter 4).

In Chapter 5, we searched for the optimal value of the *run penalty*, the parametrization of LOAC which allows us to interpolate between the results of the two-step optimization of the masked superstring and the computation of the matchtigs. We conclude that the optimal value for k-mers of lengths 23 and 31 for genomes is ≈ 12 and for pangenomes ≈ 7, suggesting that neither matchtigs nor the results of the two-step optimization are an optimal representation of k-mer sets.

We have evaluated the performance of LOAC against the two-step optimization method from KmerCamel🐫. The theoretical quadratic time complexity of LOAC prevents its usage for larger pangenomes or higher values of k, but for datasets with hundreds of thousands of k-mers, we were able to reach comparable running times and even outperform current two-step methods on genome datasets. The memory usage of LOAC was comparable to two-step methods.

The results of masked superstring compressibility show that, for genomes, our implementation was about 0.1% better considering the theoretical objective function, but about 0.5% better in practice (when compressed with `xz`). For pangenomes, LOAC practically performed 1.7% to 3% better for most of the datasets (except for the *human microbiome metagenome*, where the improvement was only 0.5%). For *SARS-COV-2 pangenome* dataset, LOAC was about 5.5% better for $k = 31$ and even 6.5% better for $k = 23$.

We were also able to decrease the number of runs of ones in the mask by about 30% for genomes and about 10% for pangenomes (except for *SARS-COV-2 pangenome* dataset, where the resulting mask had about 50% of runs of ones).

We suggest several possible ways in which future research might continue:

- The main goal is to to develop asymptotically faster and more efficient algorithms for the joint optimization of masked superstrings. Theoretical time complexity can be improved using advanced data structures or preforming an informed search in the AC. Improving practical running-time and memory requirements is also possible by further optimizing the code or introducing parallelization.
- Although our results show that the optimal value of the run penalty parameter is the same for many distinct datasets, more datasets should be analyzed, for example, larger genomes and pangenomes. Inspecting the relation between the optimal run penalty for a given dataset and its properties, such as the average degree in the de Bruijn graph, can also lead to interesting results with a possibility for improvement in the masked superstring computation. Computing or predicting the optimal run penalty for a given dataset before the computation of masked superstring (or integrating the decision of run penalty into the optimization algorithm) can simplify the process and lead to better results.
- We only considered one objective function that we used to model the space complexity of masked superstrings. However, different objective functions can be used to model the practical compressibility of masked superstrings. Developing algorithms for joint optimization of masked superstrings according to those functions might lead to better practical results. Focusing on functions for which the optimal masked superstring can be constructed in linear or polynomial time, instead of NP-hard objectives, can lead to even faster practical algorithms.

In conclusion, we consider joint optimization of masked superstrings a promising method of masked superstring construction, with a great number of possible applications for genomic data compression and efficient k-mer-based index construction. We look forward to further studying the interesting topic of masked superstrings.

Bibliography

[Abo+15] Ryan P Abo, Matthew Ducar, Elizabeth P Garcia, Aaron R Thorner, Vanesa Rojas-Rudilla, Ling Lin, Lynette M Sholl, William C Hahn, Matthew Meyerson, Neal I Lindeman, et al. "BreaKmer: detection of structural variation in targeted massively parallel sequencing data using kmers". In: *Nucleic acids research* 43.3 (2015), e19–e19.

[AC75] Alfred V. Aho and Margaret J. Corasick. "Efficient string matching: an aid to bibliographic search". In: *Commun. ACM* 18.6 (June 1975), pp. 333–340. ISSN: 0001-0782. DOI: 10.1145/360825.360855. URL: https://doi.org/10.1145/360825.360855.

[Alt+90] Stephen F Altschul, Warren Gish, Webb Miller, Eugene W Myers, and David J Lipman. "Basic local alignment search tool". In: *Journal of molecular biology* 215.3 (1990), pp. 403–410.

[Ama+20] Shanika L Amarasinghe, Shian Su, Xueyi Dong, Luke Zappia, Matthew E Ritchie, and Quentin Gouil. "Opportunities and challenges in long-read sequencing data analysis". In: *Genome biology* 21.1 (2020), p. 30.

[Bai+19] Amanda Baizan-Edge, Peter Cock, Stuart MacFarlane, Wendy McGavin, Lesley Torrance, and Susan Jones. "Kodoja: A workflow for virus detection in plants using k-mer analysis of RNA-sequencing data". In: *Journal of General Virology* 100.3 (2019), pp. 533–542.

[BBK21] Karel Břinda, Michael Baym, and Gregory Kucherov. "Simplitigs as an efficient and scalable representation of de Bruijn graphs". In: *Genome biology* 22 (2021), pp. 1–24.

[BBS18] Florian P Breitwieser, Daniel N Baker, and Steven L Salzberg. "KrakenUniq: confident and fast metagenomics classification using unique k-mer counts". In: *Genome biology* 19 (2018), pp. 1–10.

[BM13] James K Bonfield and Matthew V Mahoney. "Compression of FASTQ and SAM format sequencing data". In: *PloS one* 8.3 (2013), e59190.

[Bři+20] Karel Břinda, Alanna Callendrello, Kevin C Ma, Derek R MacFadden, Themoula Charalampous, Robyn S Lee, Lauren Cowley, Crista B Wadsworth, Yonatan H Grad, Gregory Kucherov, et al. "Rapid inference of antibiotic resistance and susceptibility by genomic neighbour typing". In: *Nature microbiology* 5.3 (2020), pp. 455–464.

[Bři+25] Karel Břinda, Leandro Lima, Simone Pignotti, Natalia Quinones-Olvera, Kamil Salikhov, Rayan Chikhi, Gregory Kucherov, Zamin Iqbal, and Michael Baym. "Efficient and robust search of microbial genomes via phylogenetic compression". In: *Nature Methods* 22.4 (2025), pp. 692–697.

[Bři16] Karel Břinda. “Novel computational techniques for mapping and classification of Next-Generation Sequencing data”. PhD thesis. Université Paris-Est, Nov. 2016. DOI: 10.5281/zenodo.1045317. URL: https://doi.org/10.5281/zenodo.1045317.

[Chi+14] Rayan Chikhi, Antoine Limasset, Shaun Jackman, Jared T Simpson, and Paul Medvedev. “On the representation of de Bruijn graphs”. In: *International conference on Research in computational molecular biology*. Springer. 2014, pp. 35–55.

[CLM16] 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. URL: https://doi.org/10.1093/bioinformatics/btw279.

[Cor+25] Molitor Corentin, Labidi Timothy, Rimbert Antoine, Cariou Bertrand, Di Filippo Mathilde, and Bardel Claire. “KILDA: identifying KIV-2 repeats from kmers”. In: *bioRxiv* (2025), pp. 2025–01.

[CPT11] Phillip EC Compeau, Pavel A Pevzner, and Glenn Tesler. “How to apply de Bruijn graphs to genome assembly”. In: *Nature biotechnology* 29.11 (2011), pp. 987–991.

[CR20] Bastien Cazaux and Eric Rivals. “Hierarchical Overlap Graph”. In: *Information Processing Letters* 155 (2020), p. 105862. ISSN: 0020-0190. DOI: https://doi.org/10.1016/j.ipl.2019.105862. URL: https://www.sciencedirect.com/science/article/pii/S0020019019301450.

[Cri58] Francis H Crick. “On protein synthesis”. In: *Symp Soc Exp Biol*. Vol. 12. 138-63. 1958, p. 8.

[CT23] Andrea Cracco and Alexandru I Tomescu. “Extremely fast construction and querying of compacted and colored de Bruijn graphs with GGCAT”. In: *Genome Research* 33.7 (2023), pp. 1198–1207.

[DG13] Sebastian Deorowicz and Szymon Grabowski. “Data compression for sequencing data”. In: *Algorithms for Molecular Biology* 8 (2013), pp. 1–13.

[DJ22] Keith Dufault-Thompson and Xiaofang Jiang. “Applications of de Bruijn graphs in microbiome research”. In: *Imeta* 1.1 (2022), e4.

[Erl89] Henry A Erlich. “Polymerase chain reaction”. In: *Journal of clinical immunology* 9.6 (1989), pp. 437–447.

[Fer+24] Pol Fernández, Rémy Amice, David Bruy, Maarten JM Christenhusz, Ilia J Leitch, Andrew L Leitch, Lisa Pokorny, Oriane Hidalgo, and Jaume Pellicer. “A 160 Gbp fork fern genome shatters size record for eukaryotes”. In: *Iscience* 27.6 (2024).

[Fle+13] Christopher Fletez-Brant, Dongwon Lee, Andrew S McCallion, and Michael A Beer. “kmer-SVM: a web server for identifying predictive regulatory sequence features in genomic data sets”. In: *Nucleic acids research* 41.W1 (2013), W544–W556.

[GJ02] Michael R Garey and David S Johnson. *Computers and intractability*. Vol. 29. wh freeman New York, 2002.

[GMM16] Sara Goodwin, John D McPherson, and W Richard McCombie. “Coming of age: ten years of next-generation sequencing technologies”. In: *Nature reviews genetics* 17.6 (2016), pp. 333–351.

[Goo+15] Sara Goodwin, James Gurtowski, Scott Ethe-Sayers, Panchajanya Deshpande, Michael C Schatz, and W Richard McCombie. “Oxford Nanopore sequencing, hybrid error correction, and de novo assembly of a eukaryotic genome”. In: *Genome research* 25.11 (2015), pp. 1750–1756.

[Her+19] Mikel Hernaez, Dmitri Pavlichin, Tsachy Weissman, and Idoia Ochoa. “Genomic data compression”. In: *Annual Review of Biomedical Data Science* 2.1 (2019), pp. 19–37.

[Hu+21] Taishan Hu, Nilesh Chitnis, Dimitri Monos, and Anh Dinh. “Next-generation sequencing technologies: An overview”. In: *Human immunology* 82.11 (2021), pp. 801–811.

[HZB24] Shien Huang, Hang Zhang, and Ergude Bao. “A Comprehensive Review of the de Bruijn Graph and Its Interdisciplinary Applications in Computing”. In: *Engineered Science* 28 (2024), p. 1061. ISSN: 2576-9898. DOI: 10.30919/es1061. URL: http://dx.doi.org/10.30919/es1061.

[Kar+24] Mikhail Karasikov, Harun Mustafa, Daniel Danciu, Marc Zimmermann, Christopher Barber, Gunnar Rätsch, and André Kahles. “Indexing all life’s known biological sequences”. In: *BioRxiv* (2024).

[KD15] Jolanta Kawulok and Sebastian Deorowicz. “CoMeta: classification of metagenomes using k-mers”. In: *PloS one* 10.4 (2015), e0121453.

[KDP25] Jamshed Khan, Laxman Dhulipala, and Rob Patro. “Fast and Scalable Parallel External-Memory Construction of Colored Compacted de Bruijn Graphs with Cuttlefish 3”. In: *bioRxiv* (2025). DOI: 10.1101/2025.02.02.636161. eprint: https://www.biorxiv.org/content/early/2025/02/06/2025.02.02.636161.full.pdf. URL: https://www.biorxiv.org/content/early/2025/02/06/2025.02.02.636161.

[KM95] John D Kececioglu and Eugene W Myers. “Combinatorial algorithms for DNA sequence assembly”. In: *Algorithmica* 13.1 (1995), pp. 7–51.

[Lee+16] Hayan Lee, James Gurtowski, Shinjae Yoo, Maria Nattestad, Shoshana Marcus, Sara Goodwin, W. Richard McCombie, and Michael C. Schatz. “Third-generation sequencing and the future of genomics”. In: *bioRxiv* (2016). DOI: 10.1101/048603. eprint: https://www.biorxiv.org/content/early/2016/04/13/048603.full.pdf. URL: https://www.biorxiv.org/content/early/2016/04/13/048603.

[Liu+12] Lin Liu, Yinhu Li, Siliang Li, Ni Hu, Yimin He, Ray Pong, Danni Lin, Lihua Lu, and Maggie Law. “Comparison of next-generation sequencing systems”. In: *BioMed research international* 2012.1 (2012), p. 251364.

[LZL19] Yuansheng Liu, Leo Yu Zhang, and Jinyan Li. “Fast detection of maximal exact matches via fixed sampling of query K-mers and Bloom filtering of index K-mers”. In: *Bioinformatics* 35.22 (2019), pp. 4560–4567.

[Mar+21] Camille Marchet, Christina Boucher, Simon J Puglisi, Paul Medvedev, Mikaël Salson, and Rayan Chikhi. “Data structures based on k-mers for querying large collections of sequencing data sets”. In: *Genome research* 31.1 (2021), pp. 1–12.

[McN+11] Nicole McNeil Ford, Cristina Montagna, Michael Difilippantonio, and T Ried. “Comparative cancer cytogenetics”. In: *Atlas of Genetics and Cytogenetics in Oncology and Haematology* (Feb. 2011). DOI: 10.4267/2042/38033.

[Moe+24] Camille Moeckel, Manvita Mareboina, Maxwell A. Konnaris, Candace S.Y. Chan, Ioannis Mouratidis, Austin Montgomery, Nikol Chantzi, Georgios A. Pavlopoulos, and Ilias Georgakopoulos-Soares. “A survey of k-mer methods and applications in bioinformatics”. In: *Computational and Structural Biotechnology Journal* 23 (2024), pp. 2289–2303. ISSN: 2001-0370. DOI: https://doi.org/10.1016/j.csbj.2024.05.025. URL: https://www.sciencedirect.com/science/article/pii/S2001037024001703.

[Par+21] Sangsoo Park, Sung Gwan Park, Bastien Cazaux, Kunsoo Park, and Eric Rivals. “A linear time algorithm for constructing hierarchical overlap graphs”. In: *arxiv preprint arxiv:2102.12824* (2021).

[PFL10] JAUME PELLICER, MICHAEL F. FAY, and ILIA J. LEITCH. “The largest eukaryotic genome of them all?” In: *Botanical Journal of the Linnean Society* 164.1 (Sept. 2010), pp. 10–15. ISSN: 0024-4074. DOI: 10.1111/j.1095-8339.2010.01072.x. URL: https://doi.org/10.1111/j.1095-8339.2010.01072.x.

[PV17] Giulio Ermanno Pibiri and Rossano Venturini. “Dynamic elias-fano representation”. In: *28th Annual symposium on combinatorial pattern matching (CPM 2017)*. Schloss Dagstuhl–Leibniz-Zentrum für Informatik. 2017, pp. 30–1.

[RA15] Anthony Rhoads and Kin Fai Au. “PacBio Sequencing and its Applications”. In: *Genomics, Proteomics & Bioinformatics* 13.5 (Nov. 2015), pp. 278–289. ISSN: 1672-0229. DOI: 10.1016/j.gpb.2015.08.002. URL: https://doi.org/10.1016/j.gpb.2015.08.002.

[Ren+17] Jie Ren, Nathan A Ahlgren, Yang Young Lu, Jed A Fuhrman, and Fengzhu Sun. “VirFinder: a novel k-mer based tool for identifying viral sequences from assembled metagenomic data”. In: *Microbiome* 5 (2017), pp. 1–20.

[RG19] Edward S Rice and Richard E Green. “New approaches for genome assembly and scaffolding”. In: *Annual review of animal biosciences* 7.1 (2019), pp. 17–40.

[Riz+19] Raffaella Rizzi, Stefano Beretta, Murray Patterson, Yuri Pirola, Marco Previtali, Gianluca Della Vedova, and Paola Bonizzoni. “Overlap graphs and de Bruijn graphs: data structures for de novo genome assembly in the big data era”. In: *Quantitative Biology* 7 (2019), pp. 278–292.

[RM21] Amatur Rahman and Paul Medevedev. “Representation of k-Mer Sets Using Spectrum-Preserving String Sets”. In: *Journal of Computational Biology* 28.4 (2021). PMID: 33290137, pp. 381–394. DOI: 10.1089/cmb.2020.0431. URL: https://doi.org/10.1089/cmb.2020.0431.

[SA23] Sebastian Schmidt and Jarno N Alanko. “Eulertigs: minimum plain text representation of k-mer sets without repetitions in linear time”. In: *Algorithms for Molecular Biology* 18.1 (2023), p. 5.

[Sch+23] Sebastian Schmidt, Shahbaz Khan, Jarno N Alanko, Giulio E Pibiri, and Alexandru I Tomescu. “Matchtigs: minimum plain text representation of k-mer sets”. In: *Genome Biology* 24.1 (2023), p. 136.

[SGA18] Barton E Slatko, Andrew F Gardner, and Frederick M Ausubel. “Overview of next-generation sequencing technologies”. In: *Current protocols in molecular biology* 122.1 (2018), e59.

[Sim+16] Jared T Simpson, Rachael Workman, P. C. Zuzarte, Matei David, L. J. Dursi, and Winston Timp. “Detecting DNA Methylation using the Oxford Nanopore Technologies MinION sequencer”. In: *bioRxiv* (2016). DOI: 10.1101/047142. eprint: https://www.biorxiv.org/content/early/2016/04/04/047142.full.pdf. URL: https://www.biorxiv.org/content/early/2016/04/04/047142.

[Sla24] Ondřej Sladký. *Masked Superstrings for Efficient k-Mer Set Representation and Indexing*. May 2024. DOI: 10.5281/zenodo.11076871. URL: https://doi.org/10.5281/zenodo.11076871.

[SNH98] Martin N. Szmulewicz, Gabriel E. Novick, and Rene J. Herrera. “Effects of Alu insertions on gene function”. In: *ELECTROPHORESIS* 19.8-9 (1998), pp. 1260–1264. DOI: https://doi.org/10.1002/elps.1150190806. URL: https://analyticalsciencejournals.onlinelibrary.wiley.com/doi/abs/10.1002/elps.1150190806.

[Ste+15] Zachary D. Stephens, Skylar Y. Lee, Faraz Faghri, Roy H. Campbell, Chengxiang Zhai, Miles J. Efron, Ravishankar Iyer, Michael C. Schatz, Saurabh Sinha, and Gene E. Robinson. “Big Data: Astronomical or Genomical?” In: *PLOS Biology* 13.7 (July 2015), pp. 1–11. DOI: 10.1371/journal.pbio.1002195. URL: https://doi.org/10.1371/journal.pbio.1002195.

[STK10] Eric E. Schadt, Steve Turner, and Andrew Kasarskis. “A window into third-generation sequencing”. In: *Human Molecular Genetics* 19.R2 (Sept. 2010), R227–R240. ISSN: 0964-6906. DOI: 10.1093/hmg/ddq416. eprint: https://academic.oup.com/hmg/article-pdf/19/R2/R227/1798881/ddq416.pdf. URL: https://doi.org/10.1093/hmg/ddq416.

[SVB23] Ondřej Sladký, Pavel Veselý, and Karel Břinda. “Masked superstrings as a unified framework for textual k-mer set representations”. In: *bioRxiv* (2023). DOI: 10.1101/2023.02.01.526717. URL: https://www.biorxiv.org/content/early/2023/02/03/2023.02.01.526717.

[SVB24] Ondřej Sladkỳ, Pavel Veselỳ, and Karel Břinda. “FroM Superstring to Indexing: a space-efficient index for unconstrained k-mer sets using the Masked Burrows-Wheeler Transform (MBWT)”. In: *bioRxiv* (2024), pp. 2024–10.

[SVB25] Ondřej Sladký, Pavel Veselý, and Karel Břinda. “Towards Efficient k-Mer Set Operations via Function-Assigned Masked Superstrings”. In: *bioRxiv* (2025). DOI: 10.1101/2024.03.06.583483. URL: https://www.biorxiv.org/content/early/2025/02/22/2024.03.06.583483.

[Tal+24] Saumya Talera, Parth Bansal, Shabnam Khan, and Shahbaz Khan. “Practical algorithms for Hierarchical overlap graphs”. In: *arxiv preprint arxiv:2402.13920* (2024).

[Ukk90] Esko Ukkonen. “A linear-time algorithm for finding approximate shortest common superstrings”. In: *Algorithmica* 5.1 (1990), pp. 313–323.

[Val+13] C Alexander Valencia, M Ali Pervaiz, Ammar Husami, Yaping Qian, Kejian Zhang, C Alexander Valencia, M Ali Pervaiz, Ammar Husami, Yaping Qian, and Kejian Zhang. “Sanger sequencing principles, history, and landmarks”. In: *Next Generation Sequencing Technologies in Medical Genetics* (2013), pp. 3–11.

[VBS25] Pavel Veselý, Karel Břinda, and Ondřej Sladký. *Genomic datasets used for evalution of k-mer representations and indexes*. Zenodo, Jan. 2025. DOI: 10.5281/zenodo.14722244. URL: https://doi.org/10.5281/zenodo.14722244.

[Vig13] Sebastiano Vigna. “Quasi-succinct indices”. In: *Proceedings of the sixth ACM international conference on Web search and data mining*. 2013, pp. 83–92.

[WS14] Derrick E Wood and Steven L Salzberg. “Kraken: ultrafast metagenomic sequence classification using exact alignments”. In: *Genome biology* 15 (2014), pp. 1–12.

[Zhu+15] Zexuan Zhu, Yongpeng Zhang, Zhen Ji, Shan He, and Xiao Yang. “High-throughput DNA sequence data compression”. In: *Briefings in bioinformatics* 16.1 (2015), pp. 1–15.

Appendix A

Attachments — tables

Table A.1, Table A.2, Table A.3, and Table A.4 display the results of the computation with LOAC and GGMO and the resources used for the computation for the respective datasets and values of k.

Every second row in column *Dataset* is omitted for readability, but the row represents the results for the same dataset as the row above.

The number in parentheses next to LOAC in column *Method* is the value of the run penalty used for the computation. The column $|S|$ displays the length of the resulting masked superstring. The column R displays the number of runs of ones in its respective mask.

The column *Objective* shows the ratio of the objective function of the resulting *MS* computed with LOAC and GGMO. The value is therefore omitted in the lines with GGMO.

The column *Compressed* shows the ratio of the file size of the resulting *MS* computed with LOAC and GGMO and compressed with the `xz` tool. The value is therefore omitted in the lines with GGMO. `xz` was chosen as it gives the best results for pangenomes (see the discussion in Appendix B.2).

The columns *Relative time* and *Relative memory* display the ratio of computational resources used for the computation of masked superstrings with LOAC and GGMO. The value is therefore omitted in the lines with GGMO.

Dataset	Method	$\lvert S\rvert$	R	Objective	Compressed
$k = 23$					
S. cerevisiae	LOAC (13)	11584512	1923	0.999346	0.995386
	GGMO	11580327	2922		
A. thaliana	LOAC (12)	112216529	61030	0.997174	0.988857
	GGMO	111999245	115370		
C. elegans	LOAC (12)	94020979	42921	0.997191	0.993065
	GGMO	93881137	83044		
D. melanogaster	LOAC (12)	122300362	48838	0.997610	0.994190
	GGMO	122153486	91507		
$k = 31$					
S. cerevisiae	LOAC (13)	11627676	1486	0.999727	0.998872
	GGMO	11625770	1899		
A. thaliana	LOAC (12)	114037702	37574	0.998908	0.993524
	GGMO	113949518	57337		
C. elegans	LOAC (12)	95435532	28203	0.998964	0.995577
	GGMO	95381939	42144		
D. melanogaster	LOAC (12)	124074400	38215	0.999212	0.996181
	GGMO	124009310	53106		
$k = 63$					
S. cerevisiae	LOAC (13)	11716966	692	0.999956	0.998710
	GGMO	11716482	766		
A. thaliana	LOAC (12)	116729307	10563	0.999823	0.998773
	GGMO	116719560	12931		
C. elegans	LOAC (12)	97600883	10324	0.999816	0.999907
	GGMO	97594081	12289		
D. melanogaster	LOAC (12)	127974579	22483	0.999810	0.999189
	GGMO	127959959	25754		

Table A.1 The values of $\lvert S\rvert$ and R when using LOAC and GGMO, relative value of $\varphi(MS)$ for LOAC compared to GGMO and relative improvement of size of the resulting file compressed with several tools for LOAC compared to GGMO for several datasets with the respective run penalties. See Appendix A for details.

Dataset	Method	Time	Relative time	Memory	Relative memory
$k = 23$					
S. cerevisiae	LOAC	18	0.486486	1207544	1.058918
	GGMO	37		1140356	
A. thaliana	LOAC	249	0.651832	11038640	0.822168
	GGMO	382		13426260	
C. elegans	LOAC	236	0.771242	9322680	1.024791
	GGMO	306		9097152	
D. melanogaster	LOAC	253	0.532632	12109044	0.858869
	GGMO	475		14098828	
$k = 31$					
S. cerevisiae	LOAC	16	0.355556	1206536	1.056485
	GGMO	45		1142028	
A. thaliana	LOAC	232	0.486373	11267528	0.830820
	GGMO	477		13561940	
C. elegans	LOAC	204	0.503704	9505204	1.032748
	GGMO	405		9203796	
D. melanogaster	LOAC	256	0.506931	12284252	0.864211
	GGMO	505		14214412	
$k = 63$					
S. cerevisiae	LOAC	24	0.380952	1708468	1.211903
	GGMO	63		1409740	
A. thaliana	LOAC	341	0.485755	16501056	0.918193
	GGMO	702		17971220	
C. elegans	LOAC	319	0.527273	13837384	1.208069
	GGMO	605		11454136	
D. melanogaster	LOAC	493	0.610905	17974848	0.964258
	GGMO	807		18641124	

Table A.2
The computation time and maximal memory used during the computation using LOAC and GGMO, and relative values of time and memory used for LOAC compared to GGMO for several datasets with the respective run penalties. See Appendix A for details.

Dataset	Method	$\lvert S\rvert$	R	Objective	Compressed
$k = 23$					
N. gonorrheae	LOAC (7)	4367760	36547	0.996386	0.978657
	GGMO	4354278	41346		
S. pneumoniae	LOAC (7)	9936107	81193	0.994681	0.964713
	GGMO	9893344	96905		
Sars-cov-2	LOAC (8)	12016468	132930	0.960956	0.936081
	GGMO	11749233	280651		
Human microbiome	LOAC (7)	410228894	4447511	0.996634	0.990488
	GGMO	408999424	4904713		
E. coli	LOAC (7)	441825210	3692862	0.993190	0.961811
	GGMO	439532030	4574985		
$k = 31$					
N. gonorrheae	LOAC (7)	5200341	34615	0.997626	0.983853
	GGMO	5191039	37956		
S. pneumoniae	LOAC (7)	11997642	72775	0.996825	0.971492
	GGMO	11969896	82761		
Sars-cov-2	LOAC (8)	17412395	148659	0.970118	0.945147
	GGMO	17160887	289435		
Human microbiome	LOAC (7)	445667523	4882757	0.998751	0.995327
	GGMO	445120323	5074697		
E. coli	LOAC (7)	557692109	3523947	0.996446	0.972309
	GGMO	556326586	4037550		
$k = 63$					
N. gonorrheae	LOAC (7)	8638894	32417	0.999074	0.987941
	GGMO	8634634	34075		
S. pneumoniae	LOAC (7)	19271279	59958	0.999053	0.985642
	GGMO	19261049	63667		
Sars-cov-2	LOAC (8)	43559589	195886	0.985000	0.968601
	GGMO	43329732	328561		

Table A.3 The values of $\lvert S\rvert$ and R when using LOAC and GGMO, relative value of $\varphi(MS)$ for LOAC compared to GGMO and relative improvement of size of the resulting file compressed with several tools for LOAC compared to GGMO for several datasets with the respective run penalties. See Appendix A for details.

Dataset	Method	Time	Relative time	Memory	Relative memory
$k = 23$					
N. gonorrheae	LOAC	18	1.200000	383856	0.864650
	GGMO	15		443944	
S. pneumoniae	LOAC	42	1.105263	837044	0.883241
	GGMO	38		947696	
Sars-cov-2	LOAC	72	1.756098	956476	0.875955
	GGMO	41		1091924	
Human microbiome	LOAC	4065	2.567909	35791164	1.005203
	GGMO	1583		35605912	
E. coli	LOAC	15436	8.254545	37786980	1.017919
	GGMO	1870		37121796	
$k = 31$					
N. gonorrheae	LOAC	28	1.473684	433280	0.902764
	GGMO	19		479948	
S. pneumoniae	LOAC	71	1.365385	1023144	0.938485
	GGMO	52		1090208	
Sars-cov-2	LOAC	160	2.388060	1330012	1.065394
	GGMO	67		1248376	
Human microbiome	LOAC	5398	2.763953	35936752	1.003942
	GGMO	1953		35795656	
E. coli	LOAC	15029	5.487039	45788092	0.825558
	GGMO	2739		55463192	
$k = 63$					
N. gonorrheae	LOAC	231	4.714286	901404	1.192087
	GGMO	49		756156	
S. pneumoniae	LOAC	415	3.516949	2127404	0.920717
	GGMO	118		2310596	
Sars-cov-2	LOAC	1772	7.003953	4340844	0.922072
	GGMO	253		4707704	

Table A.4
The computation time and maximal memory used during the computation using LOAC and GGMO, and relative values of time and memory used for LOAC compared to GGMO for several datasets with the respective run penalties. See Appendix A for details.

Appendix B

Attachments — plots

B.1 Searching for the optimal run penalty

Figure B.1, Figure B.2, Figure B.3, Figure B.4, Figure B.5, Figure B.6, Figure B.7, Figure B.8, and Figure B.9 display the results of searching for the optimal run penalty for the respective datasets.

The `X`-axis displays different values of the run penalty used as input parameter for LOAC [1]. The orange points correspond to the results of the computation with the respective run penalties. The blue horizontal line displays the results for GGMO. Note that only one measurement was made, since GGMO does not use the run penalty.

In the upper half of each figure, the results for $k = 23$ are shown. In the lower half, the results for $k = 31$ are shown.

Objective displays the value of the objective function (see Definition 2) on the left side of the plot and the relative value of the objective function (compared to the value of GGMO) on the right side.

[1] For measurements with a run penalty 0, the resulting M from LOAC is not optimal with respect to the resulting S, as the computation is equivalent to the first step of the two-step optimization instead of single-step optimization. Therefore, we used the mask optimization from KmerCamel🐫. For other values of the run penalty, the resulting mask is optimal, and no other optimization was required.

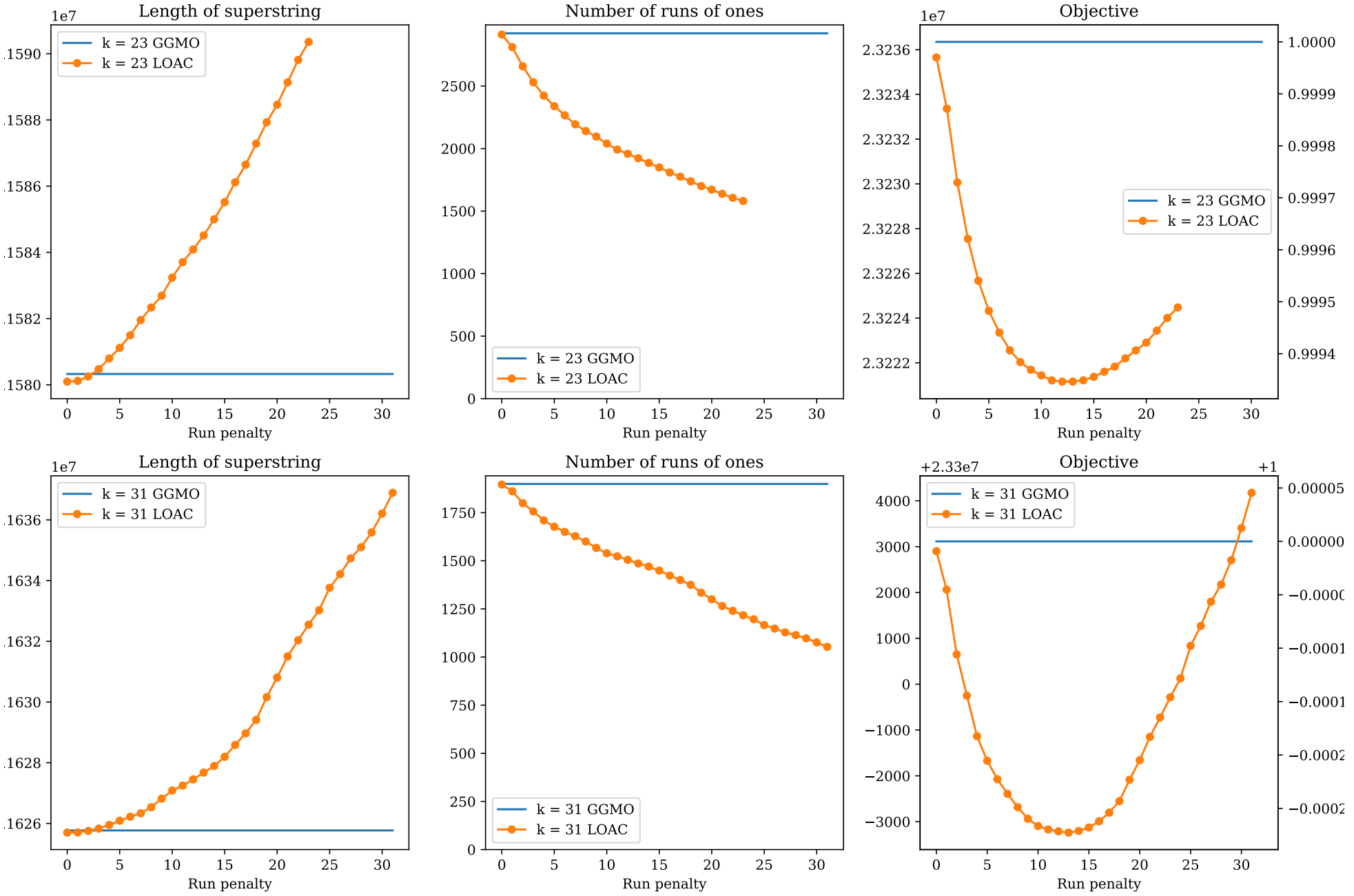


Figure B.1 The dependence of $|S|$, R and relative values of $\varphi(MS)$ on the run penalty parameter used in LOAC for *S. cerevisiae*, compared to results of GGMO. Optimal run penalty for $k = 23$: $P_{run} = 12$, for $k = 31$: $P_{run} = 13$. See Appendix B.1 for details.

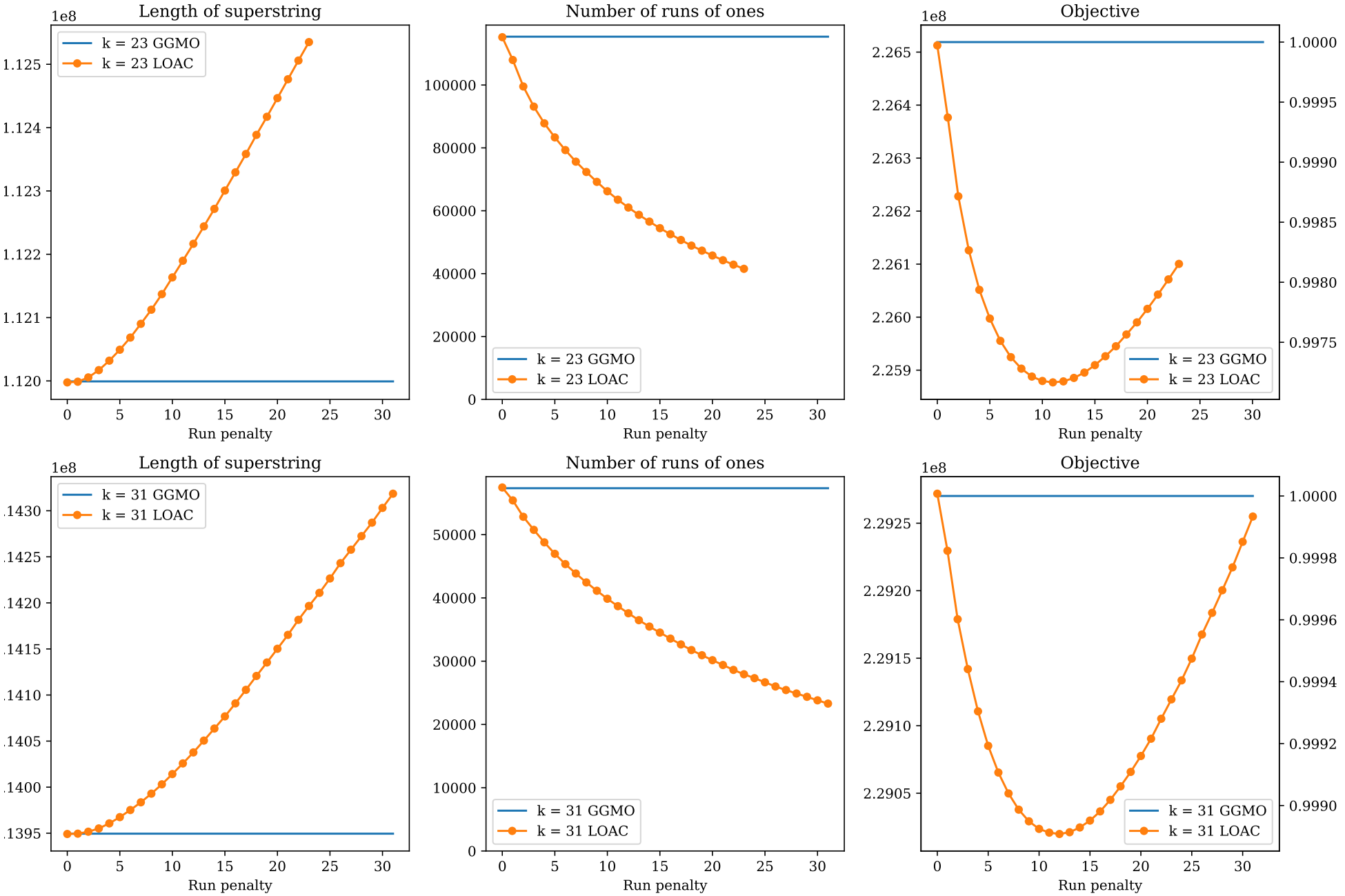


Figure B.2 The dependence of $|S|$, R and relative values of $\varphi(MS)$ on the run penalty parameter used in LOAC for *A. thaliana*, compared to results of GGMO. Optimal run penalty for $k = 23$: $P_{run} = 11$, for $k = 31$: $P_{run} = 12$. See Appendix B.1 for details.

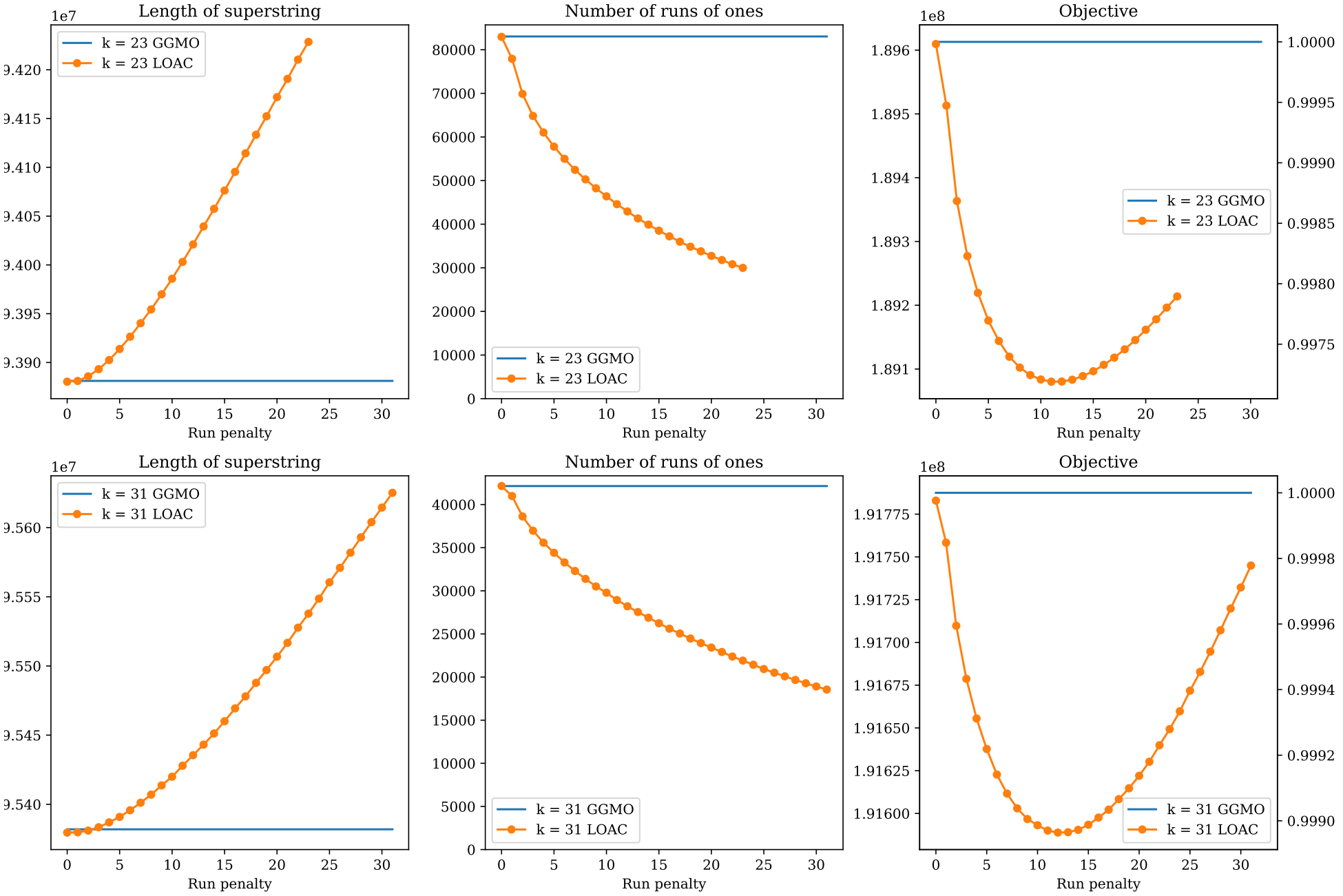


Figure B.3 The dependence of $|S|$, R and relative values of $\varphi(MS)$ on the run penalty parameter used in LOAC for *C. elegans*, compared to results of GGMO. Optimal run penalty for $k = 23$: $P_{run} = 11$, for $k = 31$: $P_{run} = 12$. See Appendix B.1 for details.

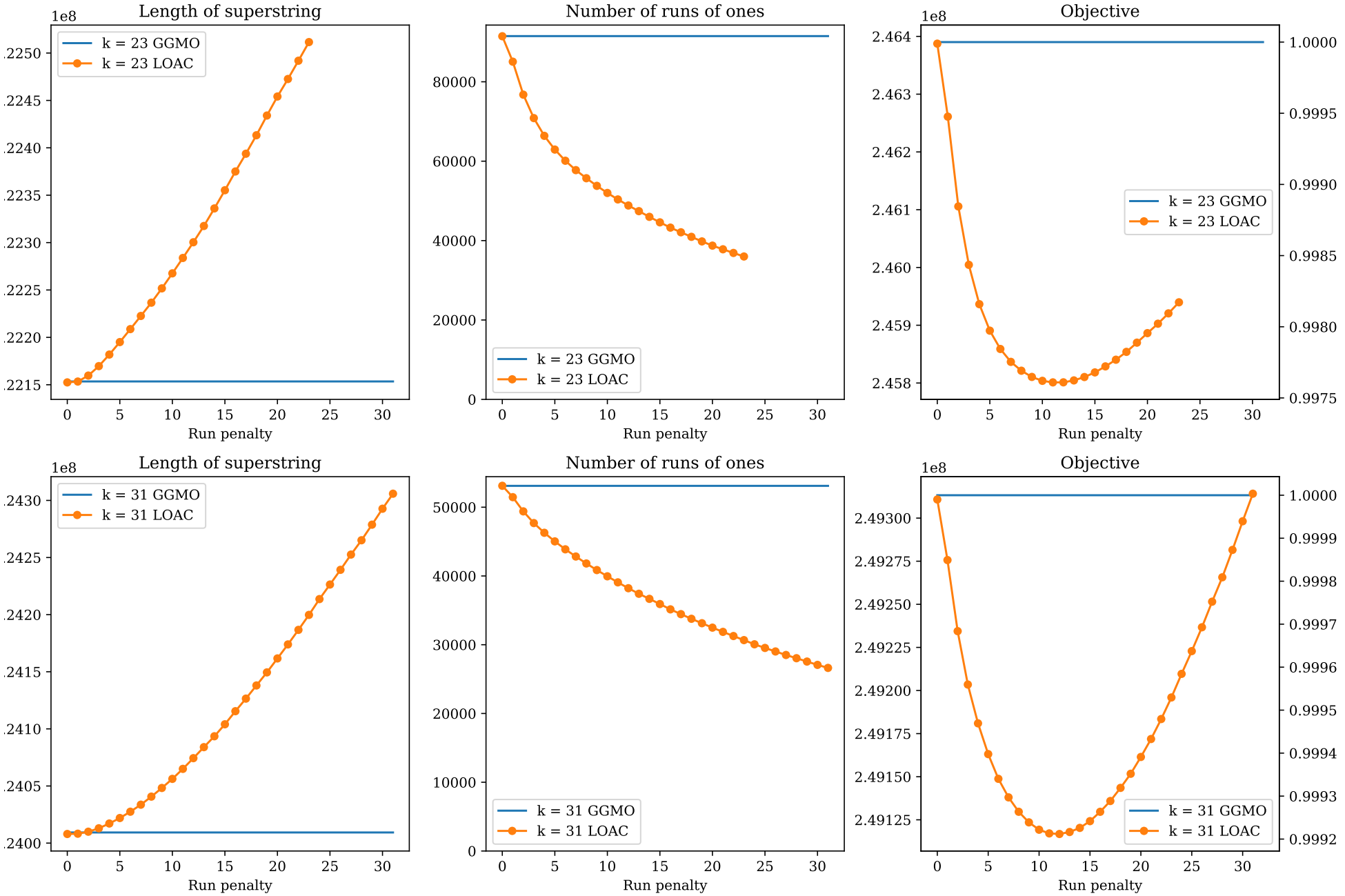


Figure B.4 The dependence of $|S|$, R and relative values of $\varphi(MS)$ on the run penalty parameter used in LOAC for *D. melanogaster*, compared to results of GGMO. Optimal run penalty for $k = 23$: $P_{run} = 11$, for $k = 31$: $P_{run} = 12$. See Appendix B.1 for details.

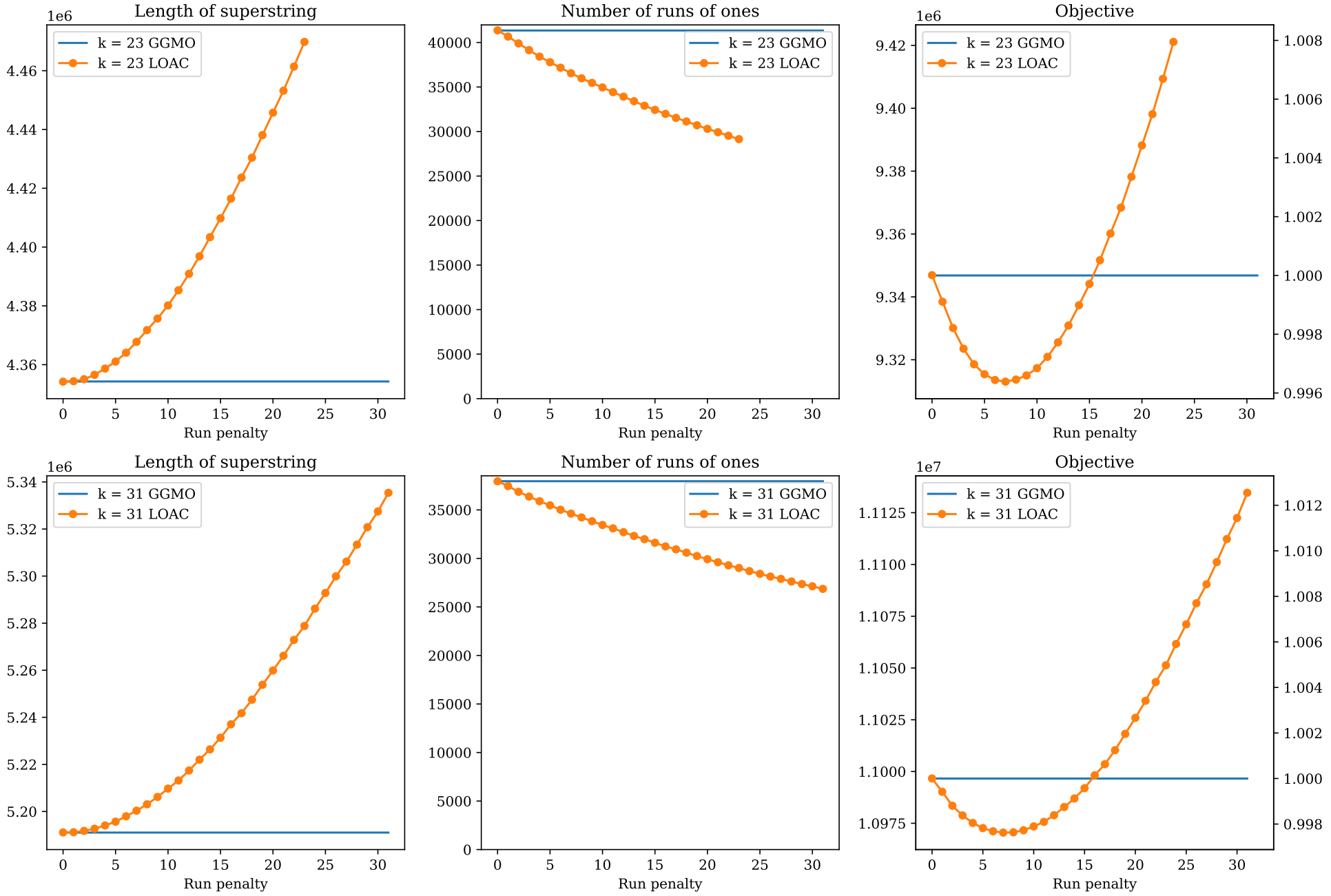


Figure B.5 The dependence of $|S|$, R and relative values of $\varphi(MS)$ on the run penalty parameter used in LOAC for *N. gonorrheae pangenome*, compared to results of GGMO. Optimal run penalty for $k = 23$: $P_{run} = 7$, for $k = 31$: $P_{run} = 7$. See Appendix B.1 for details.

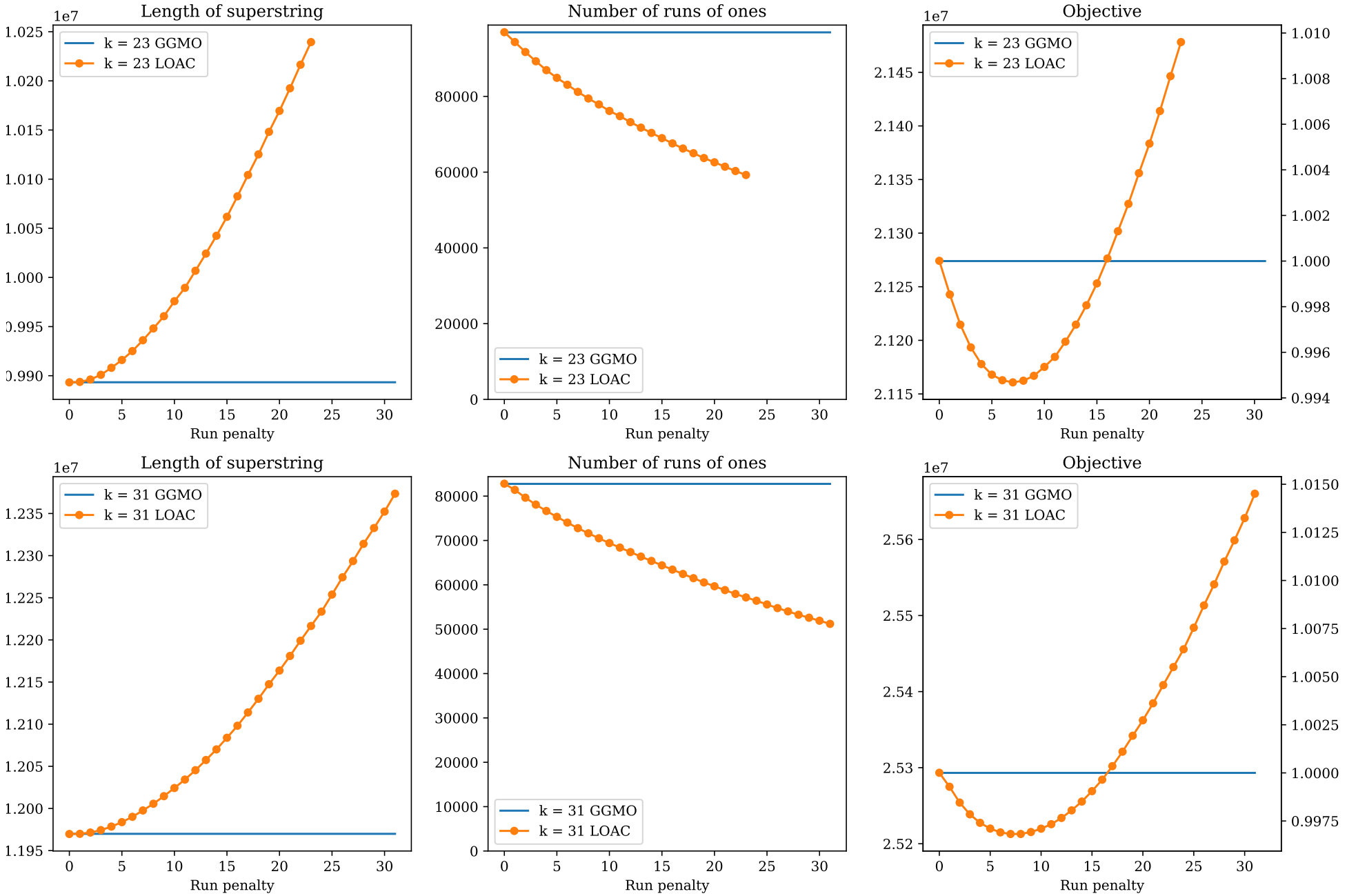


Figure B.6 The dependence of $|S|$, R and relative values of $\varphi(MS)$ on the run penalty parameter used in LOAC for *S. pneumoniae pangenome*, compared to results of GGMO. Optimal run penalty for $k = 23$: $P_{run} = 7$, for $k = 31$: $P_{run} = 7$. See Appendix B.1 for details.

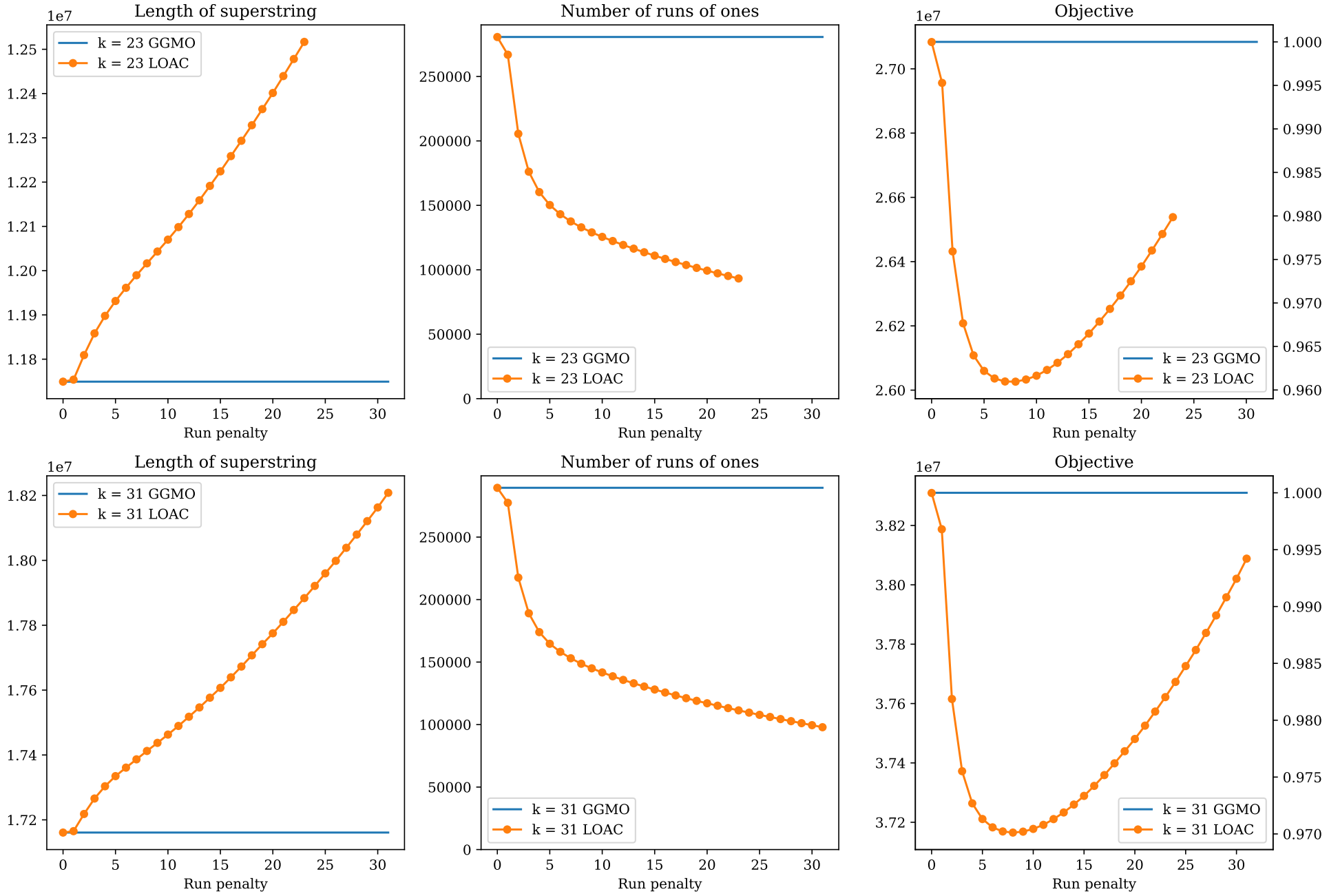


Figure B.7 The dependence of $|S|$, R and relative values of $\varphi(MS)$ on the run penalty parameter used in LOAC for *SARS-COV-2 pangenome*, compared to results of GGMO. Optimal run penalty for $k = 23$: $P_{run} = 8$, for $k = 31$: $P_{run} = 8$. See Appendix B.1 for details.

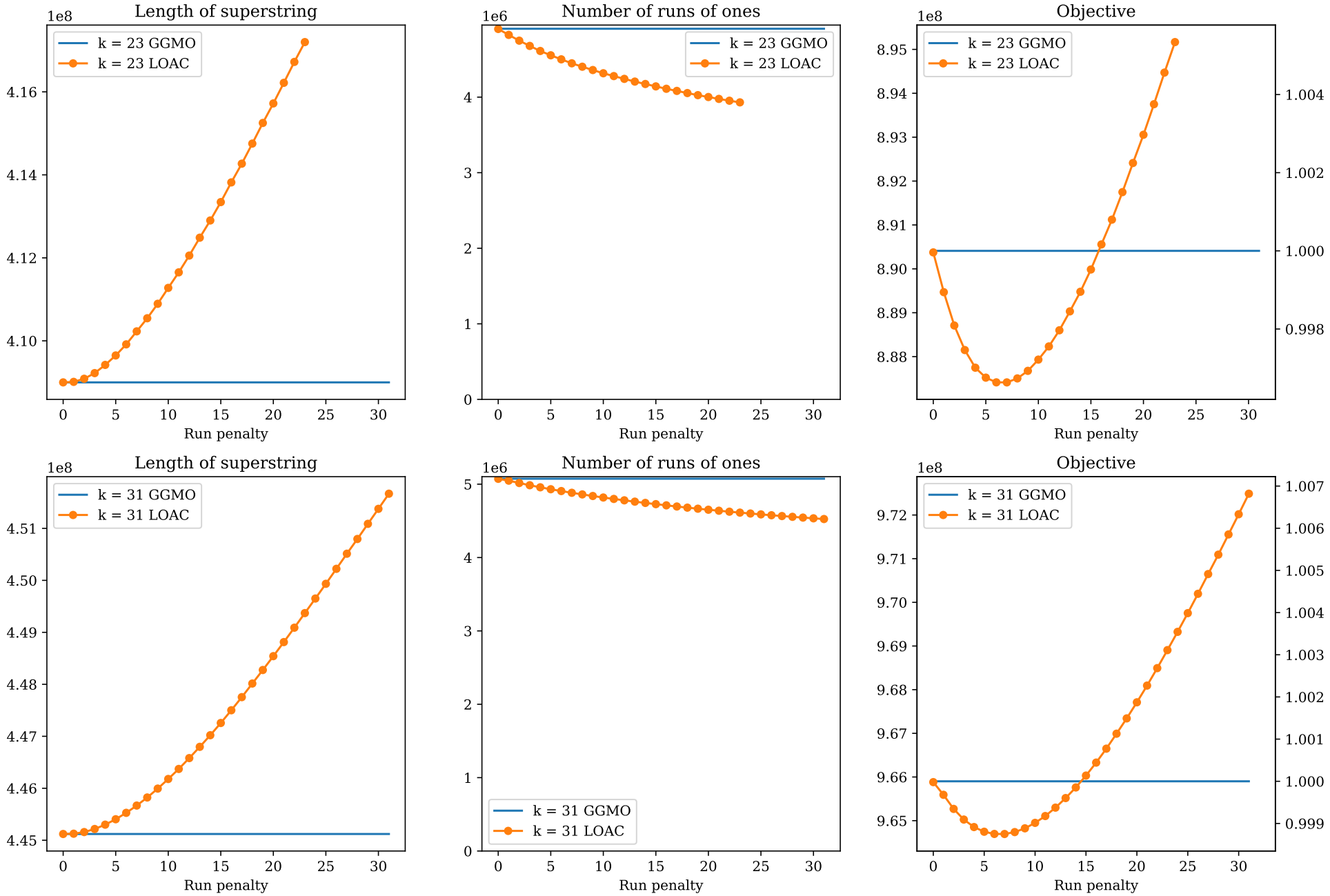


Figure B.8 The dependence of $|S|$, R and relative values of $\varphi(MS)$ on the run penalty parameter used in LOAC for *Human microbiome pangenome*, compared to results of GGMO. Optimal run penalty for $k = 23$: $P_{run} = 7$, for $k = 31$: $P_{run} = 7$. See Appendix B.1 for details.

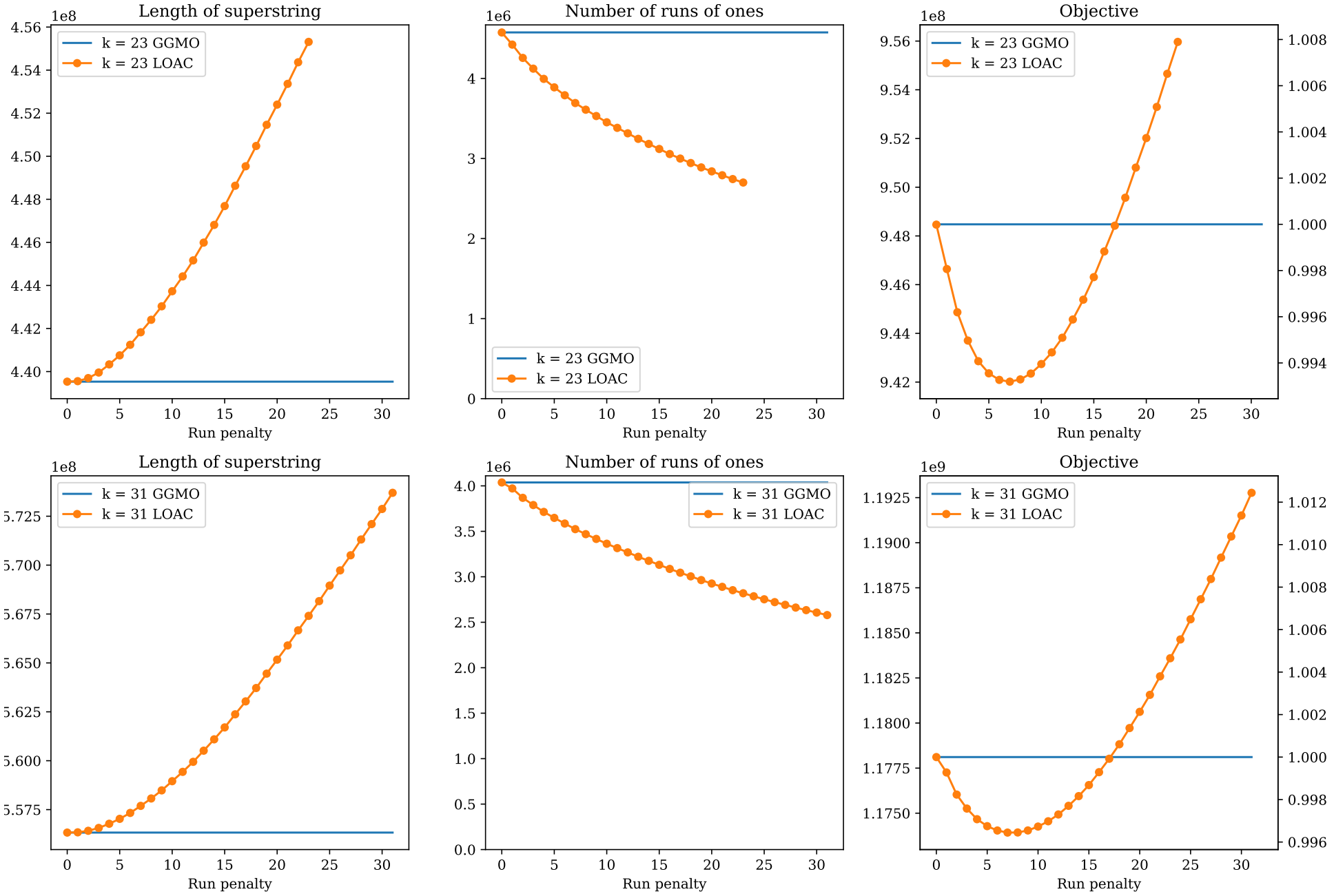


Figure B.9 The dependence of $|S|$, R and relative values of $\varphi(MS)$ on the run penalty parameter used in LOAC for *E. coli pangenome*, compared to results of GGMO. Optimal run penalty for $k = 23$: $P_{run} = 7$, for $k = 31$: $P_{run} = 7$. See Appendix B.1 for details.

B.2 Testing the algorithm for different values of k

The X-axis displays values of k used as input parameters for LOAC and GGMO.

The *Relative objective* plots show the ratio of the value of the resulting objective function for LOAC and for GGMO.

The *Relative compressed* size plots show the compressed file size ratios of the outputs of LOAC and GGMO after compression with the respective tool and parameters.

The relative performance of the compression tools is not shown in the plot. For pangenomes, `xz` performed the best, sometimes being about half the resulting file size of `gzip` and `bzip2`, and was therefore chosen to represent the compressibility improvement. `zstd` was comparable to `xz`. `lrzip` performed better than `xz` for genomes and human microbiome metagenome, but much worse for pangenomes.

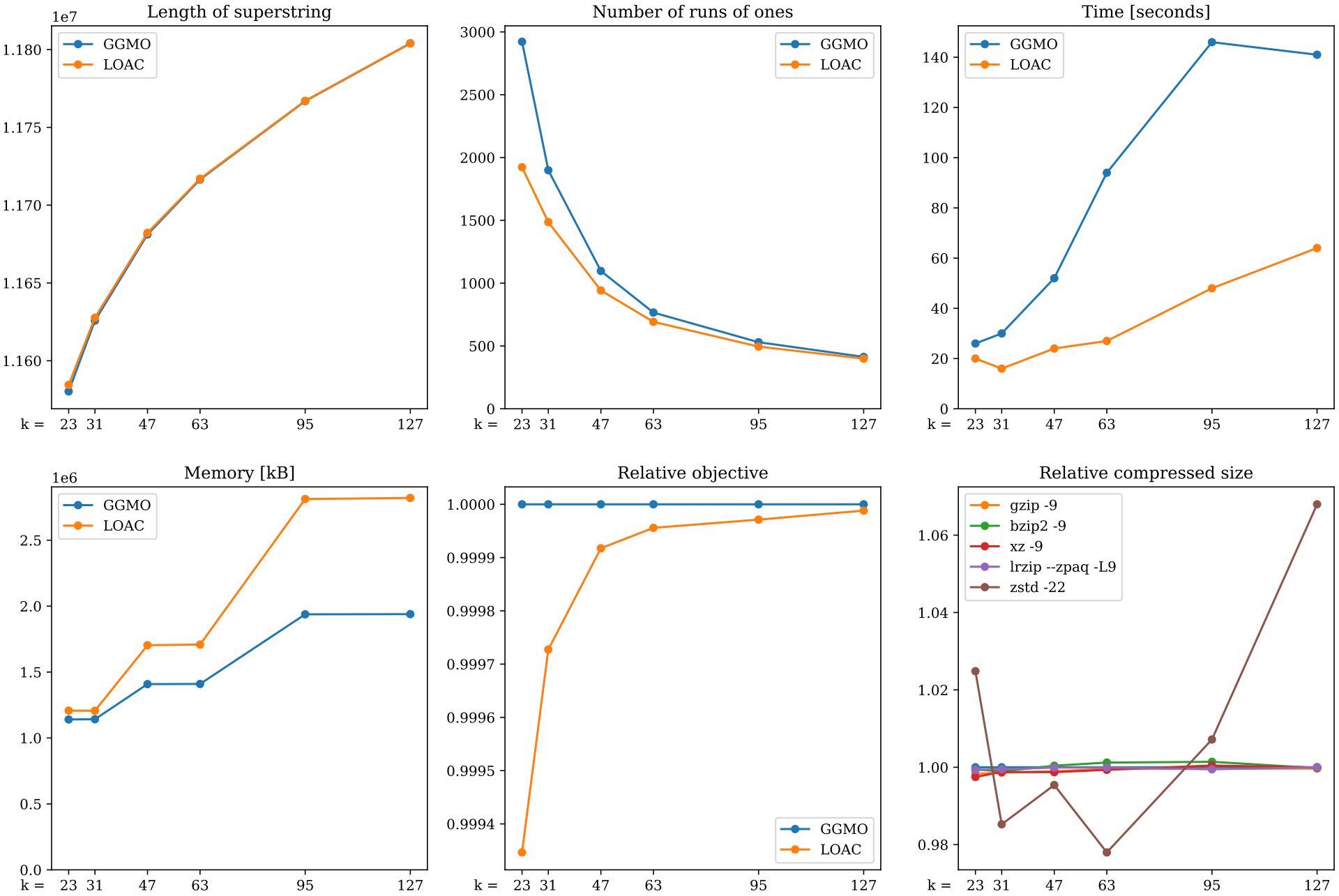


Figure B.10 The dependence of $|S|$, R, computation time and maximal memory used on the value of k for LOAC and GGMO, relative value of $\varphi(MS)$ for LOAC compared to GGMO and relative improvement of size of the resulting file compressed with several tools for LOAC compared to GGMO. For dataset *S. cerevisiae*, run penalty $P_{run} = 13$. See Appendix B.2 for details.

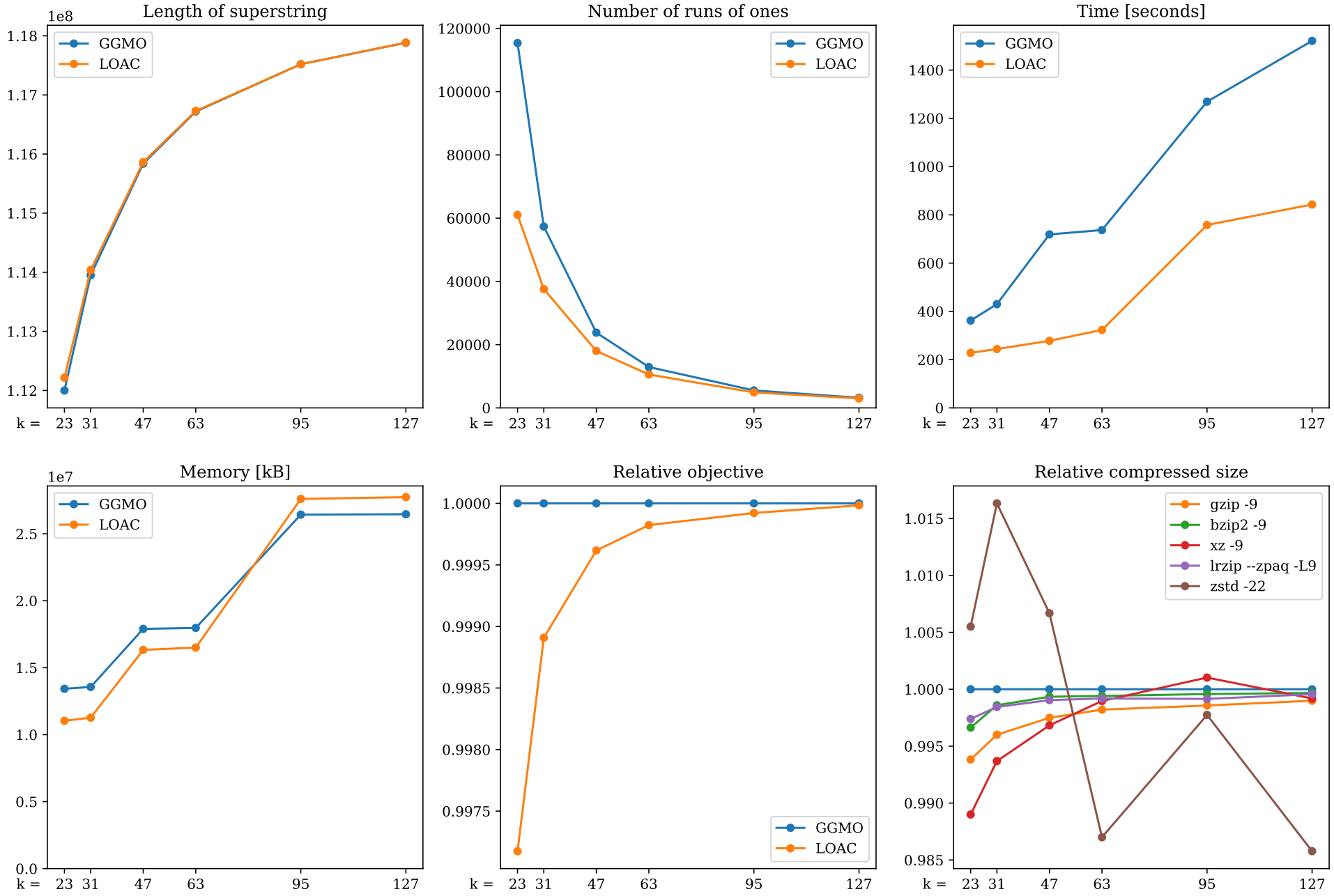


Figure B.11 The dependence of $|S|$, R, computation time and maximal memory used on the value of k for LOAC and GGMO, relative value of $\varphi(MS)$ for LOAC compared to GGMO and relative improvement of size of the resulting file compressed with several tools for LOAC compared to GGMO. For dataset *A. thaliana*, run penalty $P_{run} = 12$. See Appendix B.2 for details.

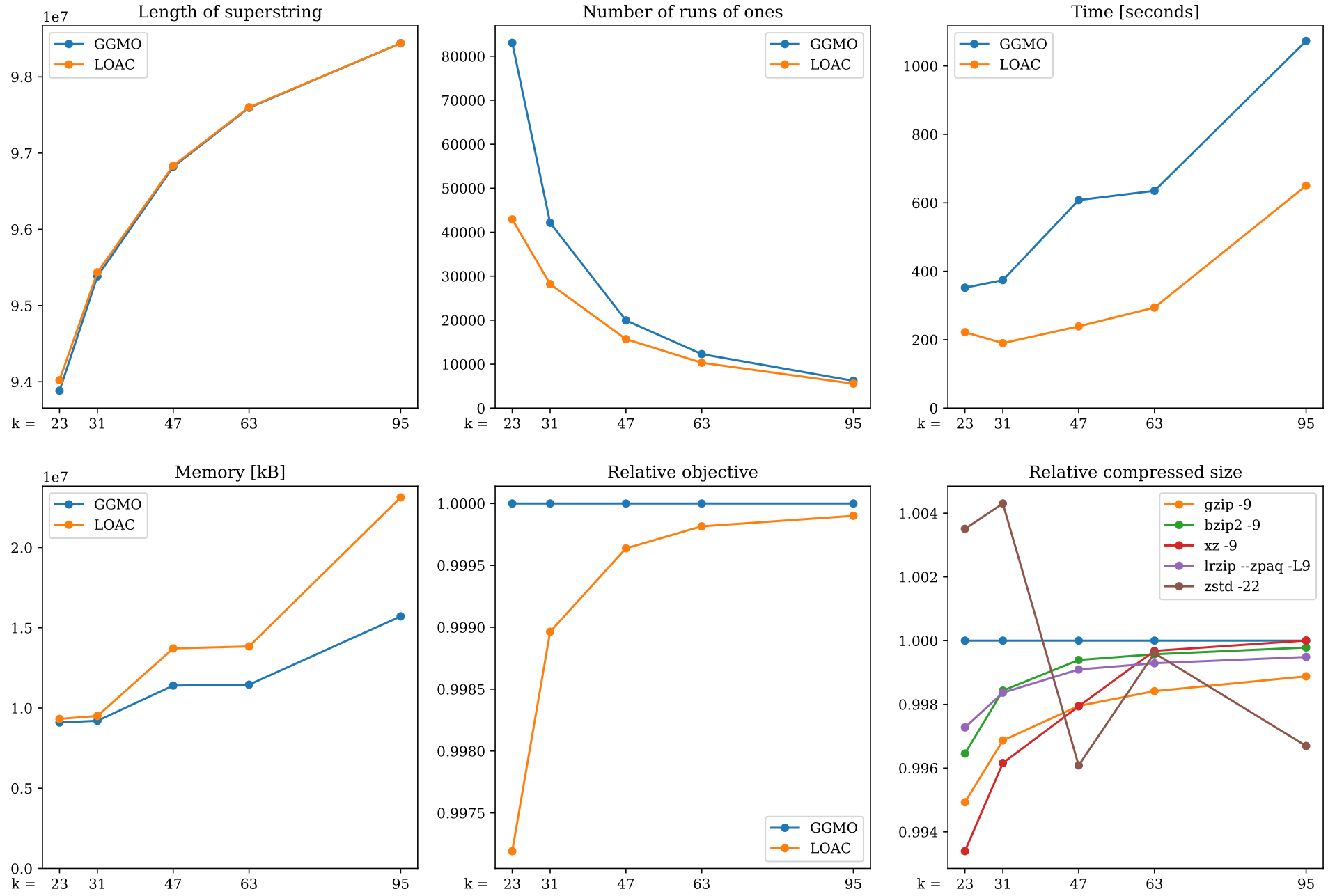


Figure B.12 The dependence of $|S|$, R, computation time and maximal memory used on the value of k for LOAC and GGMO, relative value of $\varphi(MS)$ for LOAC compared to GGMO and relative improvement of size of the resulting file compressed with several tools for LOAC compared to GGMO. For dataset *C. elegans*, run penalty $P_{run} = 12$. See Appendix B.2 for details.

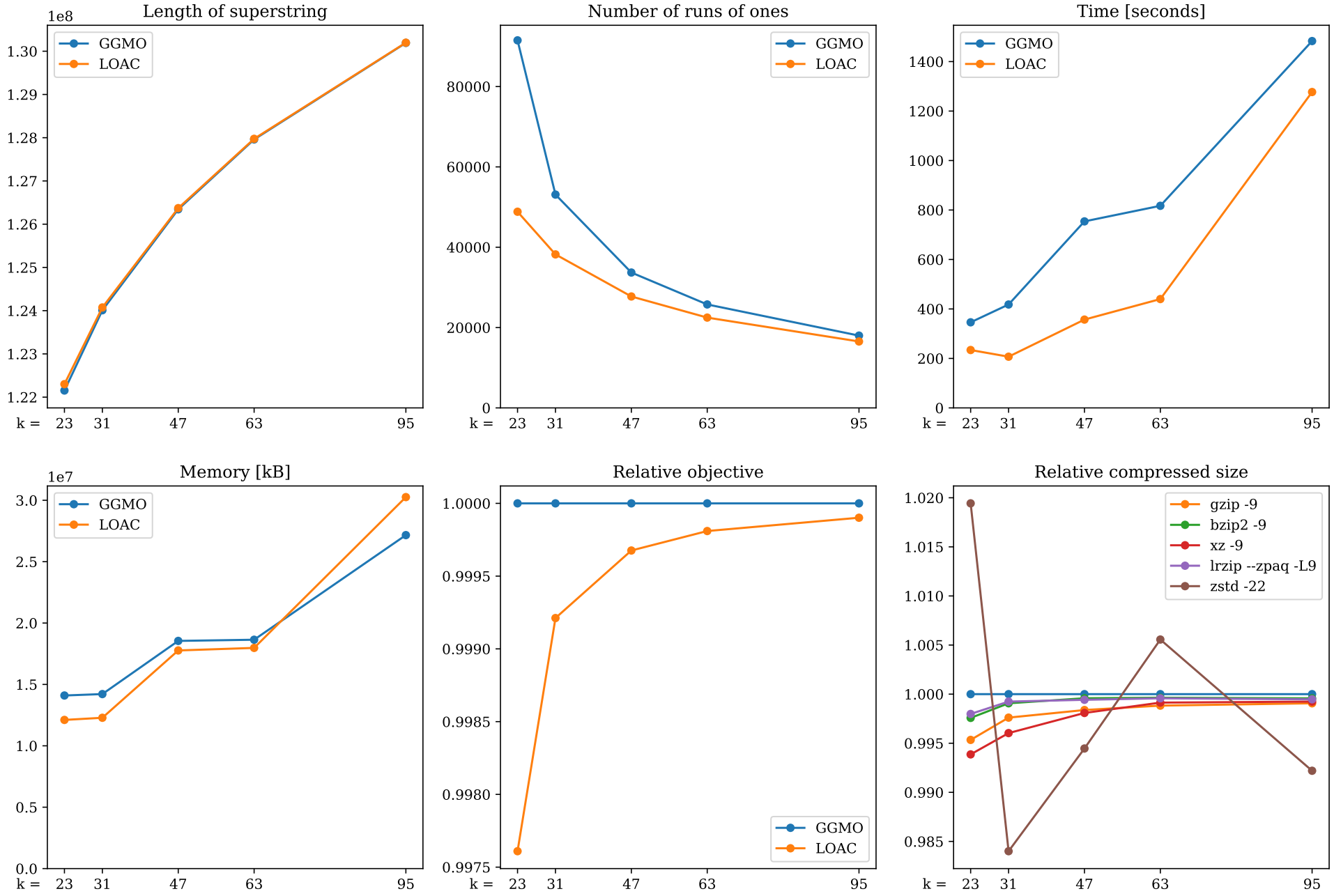


Figure B.13 The dependence of $|S|$, R, computation time and maximal memory used on the value of k for LOAC and GGMO, relative value of $\varphi(MS)$ for LOAC compared to GGMO and relative improvement of size of the resulting file compressed with several tools for LOAC compared to GGMO. For dataset *D. melanogaster*, run penalty $P_{run} = 12$. See Appendix B.2 for details.

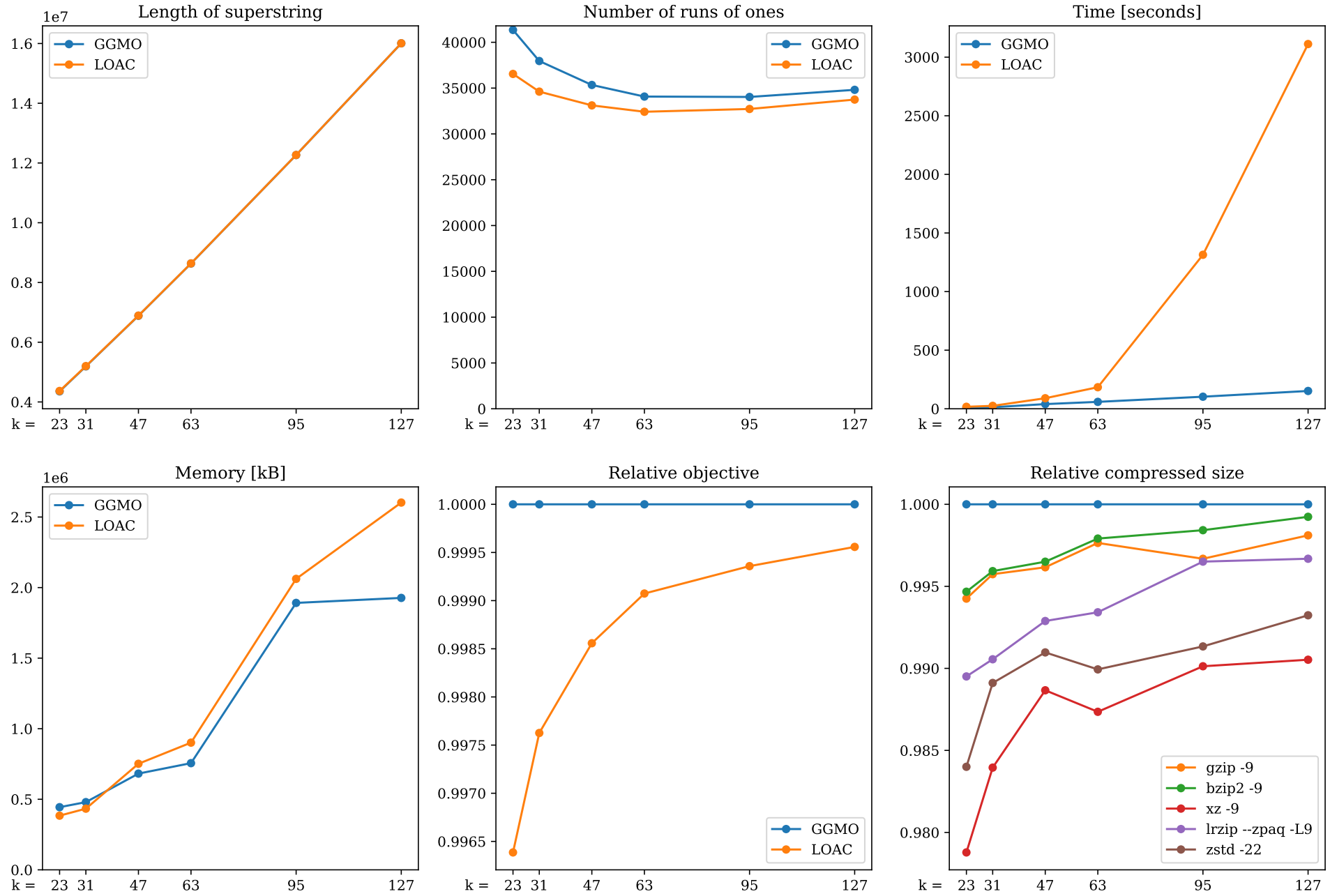


Figure B.14 The dependence of $|S|$, R, computation time and maximal memory used on the value of k for LOAC and GGMO, relative value of $\varphi(MS)$ for LOAC compared to GGMO and relative improvement of size of the resulting file compressed with several tools for LOAC compared to GGMO. For dataset *N. gonorrheae pangenome*, run penalty $P_{run} = 7$. See Appendix B.2 for details.

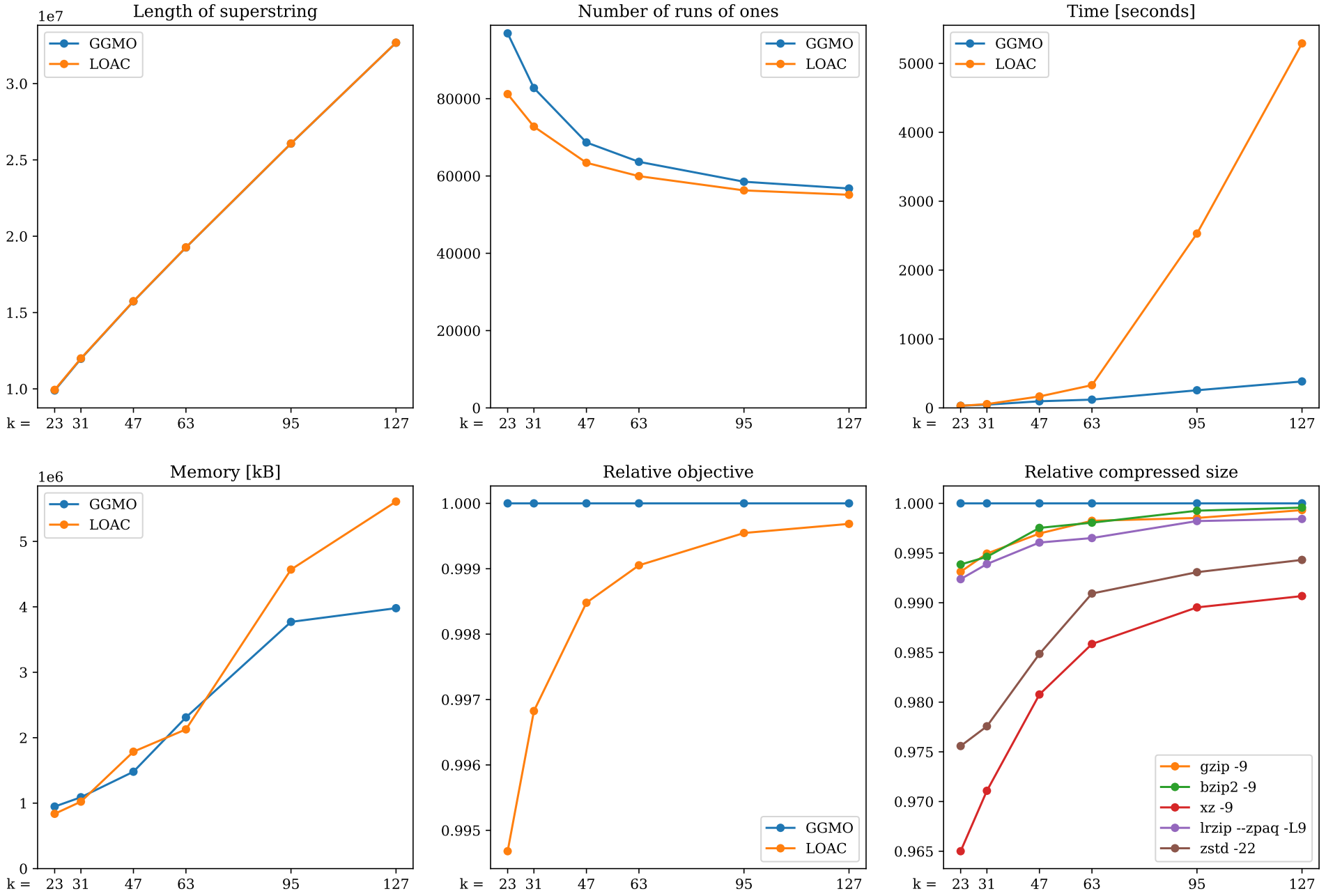


Figure B.15 The dependence of $|S|$, R, computation time and maximal memory used on the value of k for LOAC and GGMO, relative value of $\varphi(MS)$ for LOAC compared to GGMO and relative improvement of size of the resulting file compressed with several tools for LOAC compared to GGMO. For dataset *S. pneumoniae pangenome*, run penalty $P_{run} = 7$. See Appendix B.2 for details.

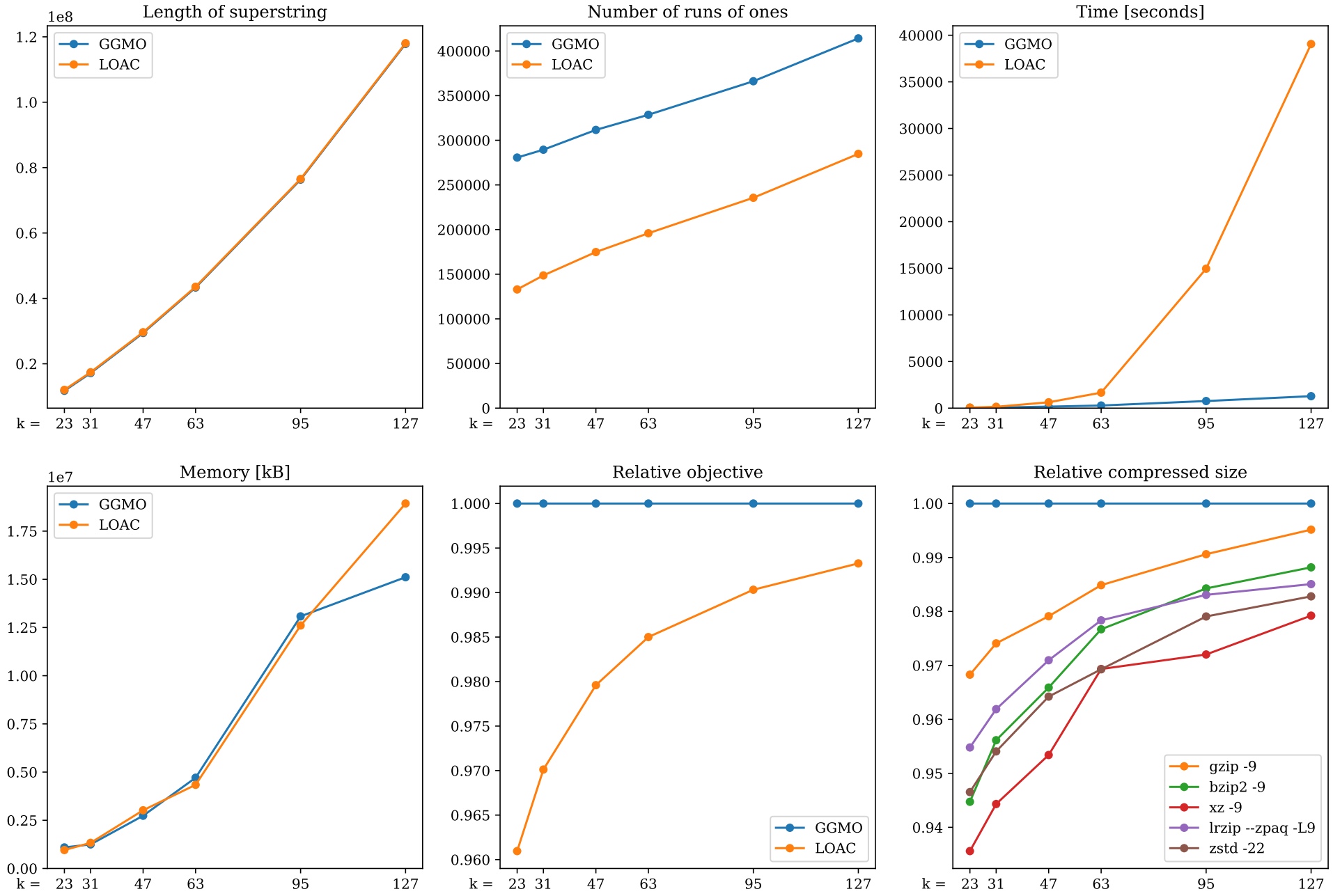


Figure B.16 The dependence of $|S|$, R, computation time and maximal memory used on the value of k for LOAC and GGMO, relative value of $\varphi(MS)$ for LOAC compared to GGMO and relative improvement of size of the resulting file compressed with several tools for LOAC compared to GGMO. For dataset *SARS-COV-2 pangenome*, run penalty $P_{run} = 8$. See Appendix B.2 for details.

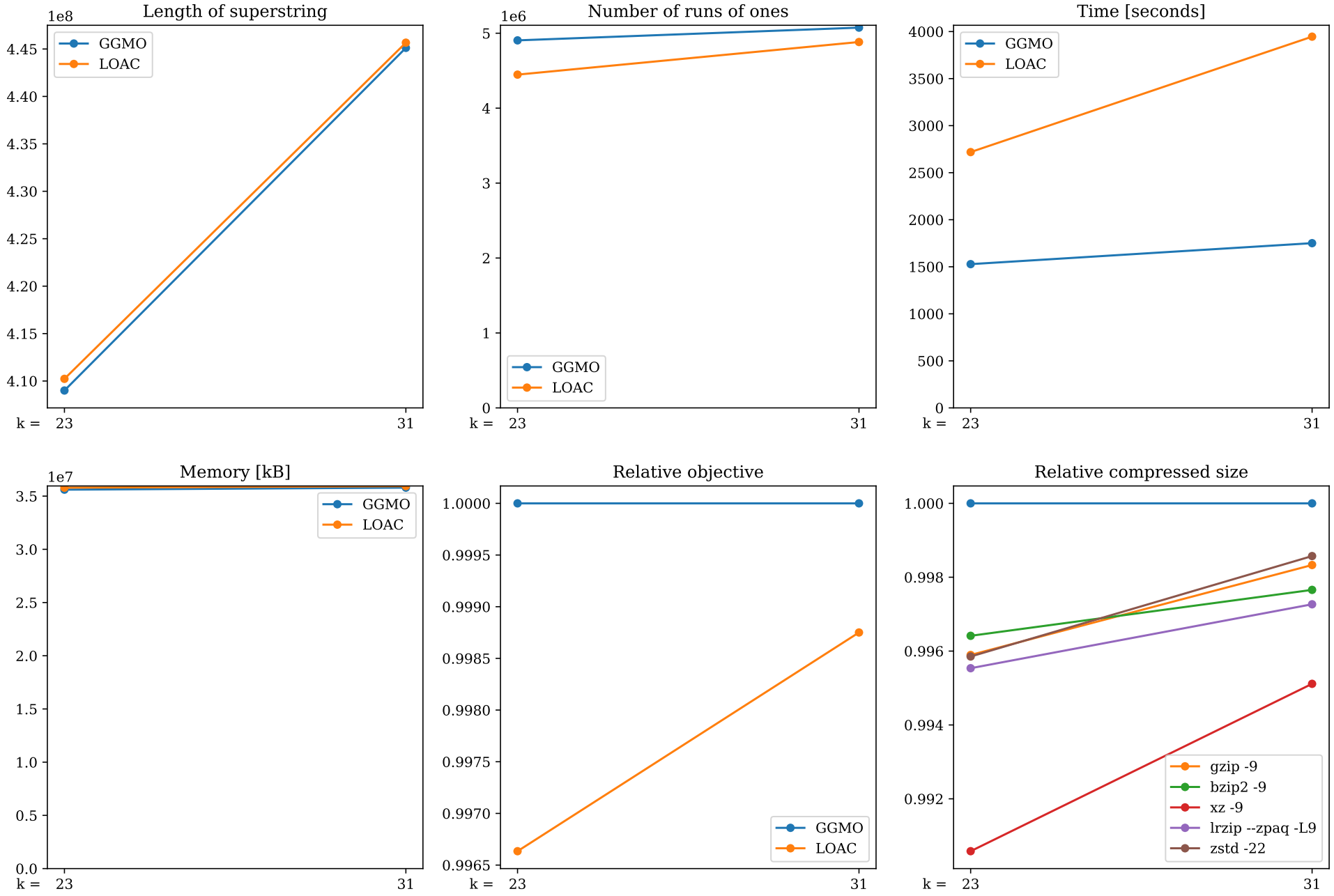


Figure B.17 The dependence of $|S|$, R, computation time and maximal memory used on the value of k for LOAC and GGMO, relative value of $\varphi(MS)$ for LOAC compared to GGMO and relative improvement of size of the resulting file compressed with several tools for LOAC compared to GGMO. For dataset *Human microbiome pangenome*, run penalty $P_{run} = 7$. See Appendix B.2 for details.

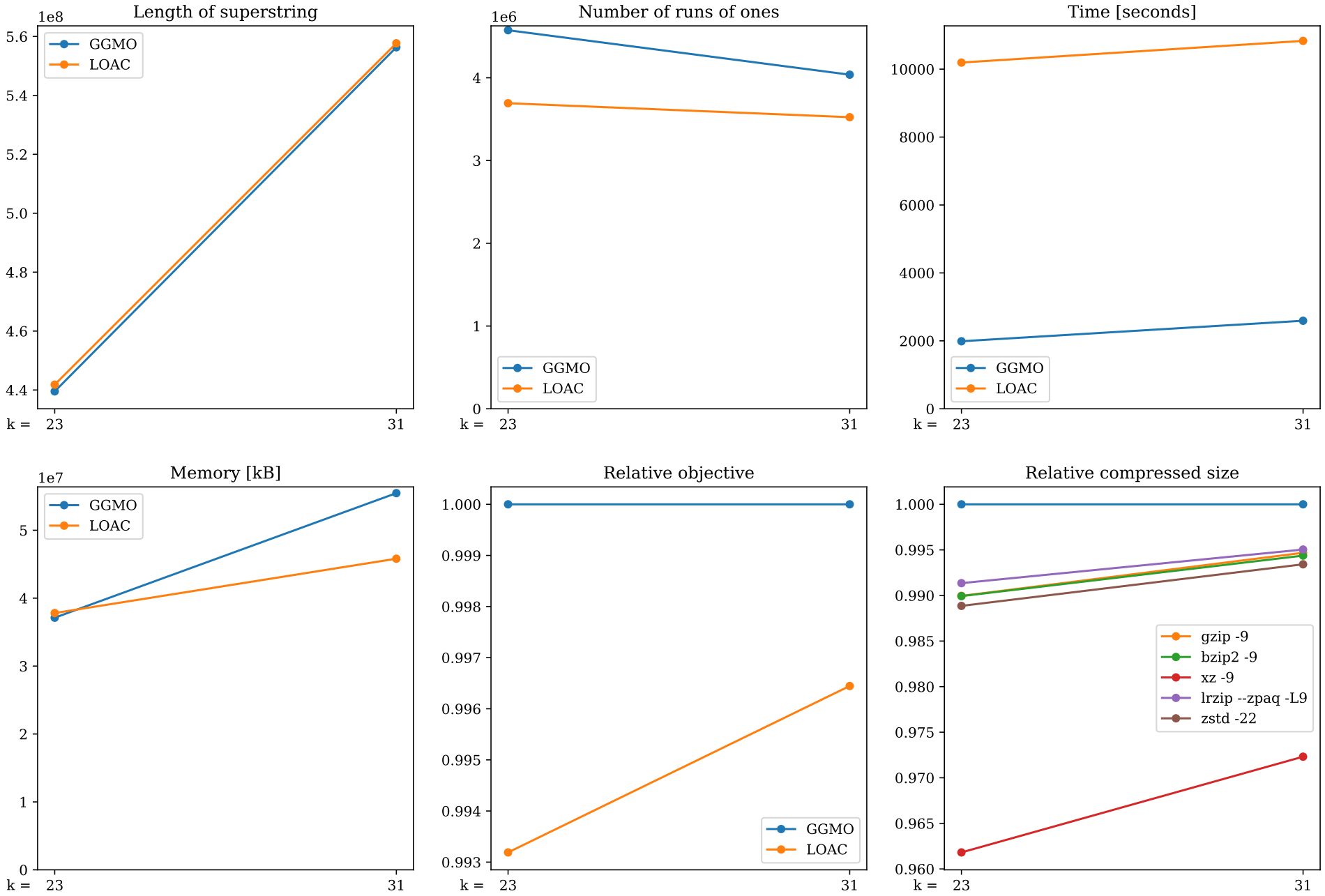


Figure B.18 The dependence of $|S|$, R, computation time and maximal memory used on the value of k for LOAC and GGMO, relative value of $\varphi(MS)$ for LOAC compared to GGMO and relative improvement of size of the resulting file compressed with several tools for LOAC compared to GGMO. For dataset *E. coli pangenome*, run penalty $P_{run} = 7$. See Appendix B.2 for details.