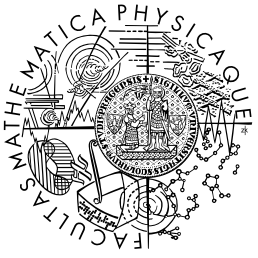


FACULTY
OF MATHEMATICS
AND PHYSICS
Charles University

BACHELOR THESIS

Ondřej Sladký

Masked Superstrings for Efficient k-Mer Set Representation and Indexing

Computer Science Institute of Charles University

Supervisor of the bachelor thesis: Mgr. Pavel Veselý, Ph.D.
Co-supervisor of the bachelor thesis: Ing. Karel Břinda, Ph.D.
Study programme: Computer Science

Prague 2024

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 extremely grateful to my supervisors Pavel Veselý and Karel Břinda for giving me the opportunity to work on this exciting research topic and introducing me to the field of computational biology. I appreciate their enthusiasm and support as well as their guidance and time spent discussing further research directions. I am grateful to Pavel for his help with running the experiments and to both Pavel and Karel for their help with corrections of the thesis. I also acknowledge the support of GA ČR project 22-22997S.

Title: Masked Superstrings for Efficient k-Mer Set Representation and Indexing

Author: Ondřej Sladký

Institute: Computer Science Institute of Charles University

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

Co-supervisor: Ing. Karel Břinda, Ph.D., Inria, Rennes

DOI: 10.5281/zenodo.11076871

Abstract: The exponential growth of genomic data calls for novel space-efficient algorithms for compression and search. State-of-the-art approaches often rely on tokenization of the data into k-mers, which are substrings of a fixed length. The popularity of k-mer based methods has led to the development of compact textual k-mer set representations, however, these rely on structural assumptions about the data which may not hold in practice. In this thesis, we demonstrate that all these representations can be viewed as superstrings of the k-mers, and as such can be generalized into a unified framework that we call the masked superstrings of k-mers. We provide two different greedy heuristics for their computation and implement them in a tool called KmerCamel🐫. We further demonstrate that masked superstrings can serve as a building block of a novel, simple k-mer set index which we call FMS-index. Additionally, if masked superstrings further integrate a demasking function f, the resulting f-masked superstrings framework allows for seamless set operations with k-mers. We experimentally evaluate the performance of masked superstrings, as well as of our FMS-index implementation, FMSI, and show that masked superstrings achieve better compression in situations where the previous methods were far from optima. Furthermore, we demostrate that using FMS-index leads to memory savings compared to state-of-the-art indexing methods. Overall, our results demonstrate the usefulness of masked superstrings as a unified theoretical framework as well as their potential in designing data structures for k-mers.

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

Název práce: Maskované nadřetězce pro efektivní reprezentaci a indexování množin k-merů

Autor: Ondřej Sladký

Ústav: Informatický ústav Univerzity Karlovy

Vedoucí bakalářské práce: Mgr. Pavel Veselý, Ph.D., Informatický ústav Univerzity Karlovy

Spoluškolitel bakalářské práce: Ing. Karel Břinda, Ph.D., Inria, Rennes

DOI: 10.5281/zenodo.11076871

Abstrakt: Současný exponenciální nárůst genomických dat vyžaduje nové prostorově úsporné algoritmy pro jejich kompresi a vyhledávání. Moderní přístupy často místo původních dat využívají příslušných množin k-merů, což jsou podřetězce pevné délky k. Popularita metod založených na k-merech vedla k vzniku kompaktních textových reprezentací množin k-merů, jež však stojí na strukturálních předpokladech, které pro data v praxi nemusí platit. V této bakalářské práci ukážeme, že na všechny tyto reprezentace lze nahlížet jako na nadřetězce množin k-merů a jako takové je zobecníme pomocí uceleného konceptu, kterému říkáme maskované nadřetězce k-merů. Navrhneme dva různé hladové algoritmy na jejich výpočet a implementujeme je v nástroji KmerCamel🐫. Dále demonstrujeme, že maskované nadřetězce fungují jako stavební kámen pro nový a jednoduchý index pro množiny k-merů, který nazýváme FMS-index. Pokud k maskovaným nadřetězcům přiřadíme navíc odmaskovávací funkci f, výsledný koncept f-maskovaných nadřetězců umožňuje jednoduché provádění množinových operací s k-mery. Experimentálně ověříme prostorovou úspornost maskovaných nadřetězců, stejně tak i naší implementace FMS-indexu. Ukážeme, že maskované nadřetězce jsou lépe komprimovatelné v situacích, kde předchozí přístupy byly daleko od optima a že FMS-index je prostorově efektivnější než současné nejlepší přístupy k indexování. Naše výsledky dokládají užitečnost maskovaných nadřetězců jakožto sjednocujícího teoretického rámce a stejně tak i jejich potenciál v návrhu datových struktur pro k-mery.

Klíčová slova: algoritmy, bioinformatika, výpočetní genomika, datové struktury, množiny k-merů, problém nejkratšího nadřetězce

Contents

Introduction

The advances in DNA sequencing technologies have resulted in an exponential growth of collections of genomic data reaching a petabyte scale. This increase brings new computational challenges as it is increasingly more difficult to analyze the data using traditional approaches [SLF+15]. To tackle this challenge, one of the heavily used key techniques is to tokenize the data into k-mers, which are distinct substrings of length k, and instead of working with the genomic data directly, analyze the set of k-mers appearing in the data.

Methods based on k-mers have found applications in large-scale genomic data search [BDR+19; BBGI19; KMD+20; BLP+23], metagenomic classification [WS14; BSPK17], rapid diagnostics of infectious diseases [BGW+15; BCM+20], and transcript abundance estimation [BPMP16; PDL+17], to name at least a few. Furthermore, k-mers find their use in the direct study of biological phenomena [LGB+18; SRM23].

This shift towards k-mer sets calls for their representations that are simultaneously highly storage-efficient and allow for fast and memory-efficient operations such as membership queries and set operations. This challenge gets increasingly more difficult by the enormous differences of k-mer sets in their size, diversity, and structural characteristics. While the most common value of k is 31, the values used in practice range from less than 10 to more than 100. The sizes of k-mer sets vary even more from several tens to hundreds of billions. Furthermore, different k-mer sets can have very different structure, based on, for instance, whether they come from single genomes, or are constructed from a collection of genomes at once.

The original approaches for representing sets of k-mers based on the information theory [CB11] provided unsatisfactory guarantees as they modeled the worst-case situations assuming independence of k-mers. These worst cases, however, may not be ever encountered in practice as k-mers are not independent in data sets used in biological applications. Not only are k-mers not independent, the typically encountered k-mer sets share the property that the k-mers are substrings of a few larger strings, known as the *spectrum-like property* (SLP) [CHM21].

Building upon this observation, modern techniques leverage the non-independence of k-mer sets [Chi21]. Particularly efficient have been textual representations of k-mers [CLJ+14; BBK21; RM21; SKA+23], which represent a k-mer set by a set of strings such that each represented k-mer appears as a substring in one of the strings. All these representations provide an efficient storage format when combined with additional data compressors and they can be easily turned into an efficient data structure for membership queries when combined with a full-text index [LD09; LD10] or a minimum perfect hash function [ASSP18; HM20; MKL21].

However, all these representations are limited by relying on the existence of $(k-1)$ long overlaps between k-mers, which in many applications are not present, for instance in modern applications combining long-read sequencing data with sub-sampling techniques. In such cases, all these representations contain a large number of very short strings and even isolated k-mers, which results in an undesirable overhead.

In this thesis, we introduce *masked superstrings* as a representation of k-mer sets that is efficient both with and without SLP and does not rely on the existence of $(k-1)$ long overlaps. We show that this representation is simultaneously memory-efficient, simple to query and supports set operations on the underlying k-mers. Throughout the work, we demonstrate that masked superstrings provide a useful and efficient building block for future k-mer based data structures and applications.

The thesis is organized as follows. In Chapter 2, we introduce the concept of masked superstrings combining the ideas of representing k-mer sets via their superstring and using a mask to mask out newly introduced "false positive" k-mers. We show that masked superstrings unify and generalize the existing representations, which also makes them a theoretical framework to study the properties of the existing representations. In Chapter 3, we show that computing optimal masked superstrings is in general NP-hard, but we provide two approximation algorithms to compute them. In Chapter 4, we describe the FMS-index, a simple full-text index and demonstrate that masked superstrings can be used to simplify the data structures for indexing k-mer sets. In Chapter 5, we introduce function-assigned masked superstrings, a generalization of masked superstrings by additionally equipping a masked superstring with a function which determines which k-mers are "false positives". We demonstrate that this framework can be used to perform set operations and when combined with the FMS-index, it provides an efficient dynamic full-text index for k-mer sets. Finally, in Chapter 6, we introduce KmerCamel🐫, a tool for efficient masked superstring computation, and FMSI, an implementation of the FMS-index, and we provide a thorough experimental evaluation of the masked superstring framework.

The thesis is based on the following papers:

Paper I [SVB23] Ondřej Sladký, Pavel Veselý, and Karel Břinda. Masked superstrings as a unified framework for textual k-mer set representations. *bioRxiv*, 2023. Presented at *RECOMB-seq*, 2023.

Paper II [SVB24] Ondřej Sladký, Pavel Veselý, and Karel Břinda. Function-assigned masked superstrings as a versatile and compact data type for k-mer sets. *bioRxiv*, 2024.

Chapters 2 and 3 are based on Paper I, Chapters 4 and 5 and Appendices B and C are based on Paper II, and Chapter 6 provides experimental evaluation for both Paper I and Paper II. Appenix A and parts of Chapters 3 and 4 are not covered in either of these papers.

1 Background

1.1 Preliminaries

Alphabet and strings. We consider strings over a constant-size alphabet Σ, typically the `ACGT` or binary alphabets. Let Σ^* be the set of all finite strings over Σ. For an empty sequence of alphabet letters, we use ϵ. By $|S|$ we denote the length of a string S and by $|S|_c$ we denote the number of occurrences of the letter c in S. For two strings S and T, let $S + T$ be their concatenation.

Runs and run-length encoding. A *run* of a letter c in a string S is a maximal substring of S consisting only of the letter c. We denote the number of different runs of c in S by $\texttt{runs}_c(S)$. The total number of runs is then the sum of the number of runs of each letter in the alphabet, i.e., $\texttt{runs}(S) = \sum_{c\in\Sigma} \texttt{runs}_c(S)$. In the case of the binary alphabet, it holds that $|\texttt{runs}_0 - \texttt{runs}_1| \leq 1$ and therefore the total number of runs in this case can be bounded as $\texttt{runs}(S) \leq 2\texttt{runs}_1(S)+1$. Also observe that in general the total number of runs is bounded by the length of S.

A *run-length encoding* of a string S is obtained by replacing each run of a letter c by the pair (c, z), where z is the length of the run. The number of bits required to store the run-length encoding of S is then $\mathcal{O}(\texttt{runs}(S) \log |S|)$.

k-mers and their sets. We refer to strings of size k over the `ACGT` alphabet as *k-mers* and we typically denote by Q an individual k-mer and by K a set of k-mers. Given a k-mer Q we call a *reverse complement* (RC) of Q, denoted by $\mathrm{RC}(Q)$, the k-mer obtained from Q by applying the following two operations.

1. Reverse the order of the letters.
2. Replace each letter by the letter corresponding to its complementary nuclear basis, i.e., `A` get substituted by `T` and vice versa and `C` gets substituted by `G` and vice versa.

Note that applying the reverse complement operation twice results in the original k-mer, i.e., $\mathrm{RC}(\mathrm{RC}(Q)) = Q$. Furthermore, we call the lexicographically smaller of Q and $\mathrm{RC}(Q)$ the *canonical k-mer* of Q.

Furthermore, we distinguish the uni-directional model and the bi-directional model, which are two different equivalence relations on the set of all k-mers. In the uni-directional model, a k-mer is considered equivalent only to itself. In the bi-directional model we further consider a k-mer and its reverse complement to be equivalent. The bi-directional model is more representative of the real world, as we cannot guarantee whether we have sequenced a k-mer or its reverse complement (see Section 1.2). Unless explicitly stated otherwise, our results apply both in the uni-directional and bi-directional model, but for clarity we provide examples in the uni-directional model.

A *k-mer set*, which we typically denote by K, is then a subset of all the equivalence classes of k-mers in the respective model. If we want to give a readable representation of a k-mer set, we represent each equivalence class by an arbitrary k-mer in it and we typically refer to the equivalence classes as to k-mers. If disambiguation is needed, we always use the canonical k-mer.

Example. Given a k-mer $Q = \texttt{GTTC}$, to get its reverse complement, we first reverse the order (which results in $\texttt{CTTG}$) and then substitute each letter by its complement to get the reverse complement $\texttt{GAAC}$, which is also the canonical k-mer of Q.

Observe that if k is even, the reverse complement of a k-mer could be the same as the k-mer itself, for example the k-mer $\texttt{ACGT}$ is its own reverse complement. However, if k is odd, the reverse complement is always different as it must differ in the middle letter.

This gives us the maximum sizes of k-mer sets for a fixed k. In the uni-directional model there are always 4^k different k-mers, while in the bi-directional model the maximum sizes differ based on the parity of k. For odd k, there are $\frac{4^k}{2}$ non-equivalent k-mers as each equivalence class contains exactly two k-mers. For even k there are $\frac{4^k+4^{\frac{k}{2}}}{2}$ non-equivalent k-mers since there are $4^{\frac{k}{2}}$ k-mers that are their own reverse complement.

De Bruijn and overlap graphs. An *overlap* between to k-mers P and Q, denoted by $\texttt{ov}(P, Q)$ is the longest suffix of P that is also a prefix of Q. The *overlap length* is simply the length of the overlap. Note that neither overlaps nor overlap length are in general symmetric. As an example, consider $P = \texttt{ACGT}$ and $Q = \texttt{CGTA}$, then $\texttt{ov}(P, Q) = \texttt{CGT}$ and $\texttt{ov}(Q, P) = \texttt{A}$. In the bi-directional model, we define the overlap as to be the maximal overlap between either P or its reverse complement and Q or its reverse complement. Note that even though the overlap in the bi-directional is not uniquely determined as there might be multiple maximal overlaps, the overlap length is always unique.

A *de Bruijn graph* of a set of k-mers K is a directed unweighted graph possibly with self-loops, where vertices are exactly k-mers from K (or more precisely the equivalence classes from K) and there is an edge from k-mer P to k-mer Q if the overlap length between P and Q is $k-1$. If the set of vertices are all the k-mers of a fixed length, we call such a de Bruijn graph a complete de Bruijn graph. Note that this definition of de Bruijn graphs used in computational biology differs from the combinatorial version, where de Bruijn graphs are always complete. In this thesis we stick to the definition from computational biology.

An *overlap graph* of a set K is a complete directed graph with weighted edges and self-loops, where the vertices are K and the weight of the edge from P to Q is the overlap length between P and Q. Note that de Bruijn graphs are just subgraphs of overlap graphs where we consider only the edges of weight $k-1$.

Unlike de Bruijn graphs which can always be stored in linear space with respect to the number of k-mers, overlap graphs require quadratic space to store the weights of all edges. To address this, a variant of the overlap graph was proposed, which is called the *hierarchical overlap graph* [CR20] (HOG) which encodes pairwise overlaps as well, but requires only linear space. Furthermore, HOGs can be constructed in linear time [PPC+21; Kha21]. HOGs found applications in peptide vaccines design [SDK23] and a subgraph of the (truncated) HOG, called the *Superstring graph*, was studied in the context of genome assembly [CSR16].

1.2 Biological Context

DNA is a macromolecule consisting of multiple nucleotide units, each of which includes one of four bases – adenine, cytosine, guanine, or thymine – attached to a deoxyribose-phosphate backbone. DNA, present in all living organisms, serves as the molecular blueprint for genetic information and guides the development and functioning of the organism. The *genome* of a particular organism encompasses all its genetic information, typically distributed across multiple DNA molecules (chromosomes) in eukaryotes and one or multiple DNA molecules (one chromosome and possibly plasmids) in most prokaryotes. In nature, genome lengths vary from several thousand base pairs for viruses [HL10] up to more than a hundred billion base pairs for some plants (*P. japonica*) [PFL10].

For the purpose of this thesis, we make the following simplifying assumptions. We model the DNA molecules as strings over the `ACGT` alphabet, where each letter encodes a different nucleotide base – adenine, cytosine, guanine, and thymine, respectively. A DNA molecule consists of two complementary strands with the nucleotides paired by hydrogen bonds; i.e., the adenines are paired with thymines and cytosines with guanines. Each DNA strand has a directionality defined by its sugar-phosphate backbone, with ends designated as 5' and 3'. The complementary strand runs antiparallel, meaning its 5' to 3' direction is opposite to that of its counterpart.

Genomes are studied using DNA sequencing, which reads fragments of the DNA molecules. The sequenced strings contain sequencing errors, therefore, the resulting strings are not exactly substrings of the original DNA, but substrings with possible additions, deletions or modifications. State-of-the-art sequencing technologies can usually read hundreds of base pairs for Illumina sequencing [LLL+12], up to several thousand base pairs for PacBio [RA15], and up to several million for Oxford Nanopore [ASD+20] sequencing. The error rate ranges from less than one percent for Illumina [SDI+16] to about 10% error rate for Nanopore [GMM16] sequencing. For a review of sequencing technologies, see [GMM16].

Usually, sequencing reads are assembled into genome assemblies [PSTU83; KM95; RG19]. This task is not simple due to the errors, possible incompleteness or ambiguity and the fact that the reads come from multiple DNA molecules. In many cases, collections of genomes, for instance of the same species, are used; we refer to them as *pan-genomes*.

To overcome this necessity of reconstructing the genome, approaches based on sets of k-mers, which can be computed both from assembled genomes (or pan-genomes) as well as directly from the reads, were proposed. Using k-mer sets help mitigate sequencing errors as we can filter out k-mers that appear in the reads only infrequently – those often result from sequencing errors. Furthermore, using sets of k-mers is beneficial not only for reads, but even for assembled genomes, as they are easier to work with.

Methods based on k-mers have been extremely successful throughout computational biology. Notable applications of k-mer-based methods include large-scale data search [BDR+19; BBGI19; KMD+20; BLP+23] where k-mer-set-based methods do not require the reference genomes to be assembled, metagenomic classification [WS14; BSPK17], infectious disease diagnostics [BGW+15; BCM+20], and transcript abundance quantification [BPMP16; PDL+17]. Furthermore,

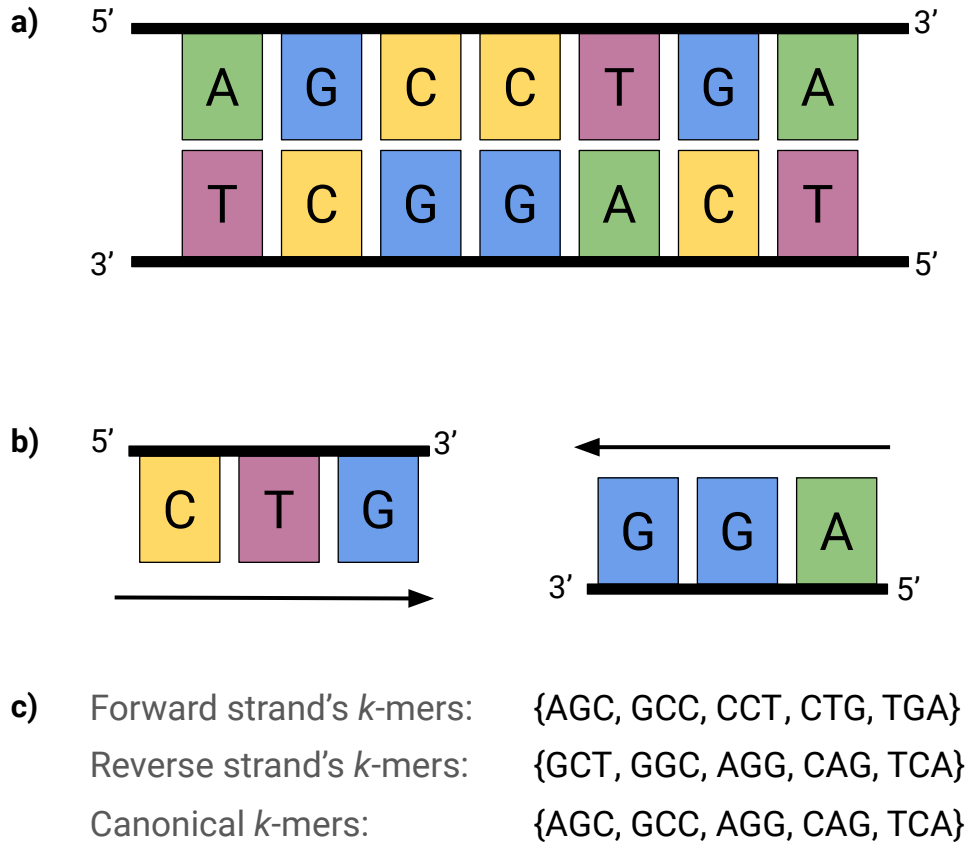


Figure 1.1: **An illustration of DNA and sequenced reads. a)** An example of chromosomal DNA depicting the paired bases C with G and A with T. **b)** Two possible sequenced reads. Note that although the orientation is inferrable from the position of the 5' and 3' ends, the reads can be from either of the strands and as the 5' and 3' ends are reversed on the second strand, the direction for each of the strand is different. Note that both the chromosome sizes and the read lengths are in reality several orders of magnitude larger than on this illustration. **c)** k-mer sets with $k = 3$ for each of the strands and canonical k-mers which are the lexicographically smaller k-mers from the corresponding k-mers in the two strands.

k-mers find their use in the direct study of biological phenomena, such as in Genome-Wide Association Studies [LGB+18] or the studies of bacterial defense systems [SRM23].

1.3 k-Mer Set Representations

Properties of k-mer sets. As follows from information theory, to encode a set of independent k-mers K, we need at least $\log \binom{4^k}{|K|}$ bits in the worst case [CB11]. However, in practice k-mer sets are not independent and therefore the general information-theoretic lower bounds can be improved.

In fact, as noted in [CHM21], k-mer sets usually share a so-called *spectrum-like property (SLP)* [CHM21], which currently lacks a formal mathematical definition, but can be informally described as that there are only a few strings containing as substrings precisely the k-mers in the set. This also implies that for the majority of k-mers in the set, there are other k-mers such that they overlap by $k-1$ characters.

However, despite that the SLP holds for many real-world k-mer sets, there are datasets which seem to be far from satisfying SLP, for instance sub-sampled sets used in bacterial pan-genomics [WS14].

Unitigs. To exploit the SLP, multiple space-efficient textual representations have been introduced. The first of those were *unitigs* [CLJ+14], which is a set $\mathcal{S}$ of strings S_i with the following properties:

(U1) Every k-mer Q in the set K is a substring of some string S_i in the set.
(U2) Every k-long substring of any string S_i is in K.
(U3) Every k-mer Q occurs as a substring only in one of the strings S_i and only once.
(U4) If a k-mer Q has in-degree in the de Bruijn graph of K higher than 1, it must be at the beginning of a string and similarly, if a k-mer has out-degree higher than 1, it must at the end of a string.

We remark that in some works, unitigs are defined to further have the smallest cumulative length possible. To distinguish, we call these *optimal unitigs*. Unitigs can best be described as a set of paths in the de Bruijn graph obtained by merging non-branching vertices. Since unitigs do not merge branching vertices which is guaranteed by the property (U4), it is a topology-preserving compaction of the de Bruijn graph. Therefore, optimal unitigs are also referred to as compacted de Bruijn graphs.

Optimal unitigs can be computed efficiently using highly optimized tools, for example BCALM2 [CLM16], TwoPaCo [MPM17], Cuttlefish 2 [KKDP22], and GGCAT [CT23]. Since unitigs were originally designed for assembly-like applications, they are quite efficient for k-mer sets corresponding to single genomes. However, when computed on highly branching de Bruijn graphs which are common for instance in bacterial pan-genomics, the unitigs can be inefficient.

Simplitigs/SPSS. To represent even branching de Bruijn graphs efficiently, *simplitigs/spectrum preserving string sets* [Bři16; BBK21; RM21; Rah23] (SPSS) have recently been introduced. They generalize over unitings by relaxing the

property (U4), i.e., not requiring the preservation of the de Bruijn graph topology. In the graph point of view, simplitigs are a set of vertex disjoint paths covering the whole de Bruijn graph. As the authors note, the property (U4) is not necessary to store the k-mer set and also in many downstream applications, which enables simplitigs to lower the storage requirements over unitigs and as a result speed up downstream applications [BBK21]. Moreover, unitigs can always be recomputed from simplitigs.

Simplitigs can be efficiently computed heuristically with ProphAsm [BBK21], which greedily constructs simplitigs by extending them by one character to both directions, or UST [RM21], which greedily glues unitigs. Although both these approaches are heuristic, for practical instances they are nearly optimal [RM21]. Simplitigs can even be computed optimally in linear time using the Eulertigs algorithm [SA23].

Matchtigs. Matchtigs [SKA+23; Sch23] generalize even further, by additionally relaxing the property (U3), thus allowing multiple occurrences of a k-mer as a substring in the string set. This relaxation can lead to further compression as it allows to merge more strings together. Matchtigs can be computed optimally in polynomial time and the authors also provide a heuristic algorithm for their computation [SKA+23].

To unify notion, since a k-mer may appear multiple times in matchtigs, we also refer to them as *Repetitive Spectrum Preserving String Sets* (rSPSS). We refer to members of SPSS as simplitigs and members of rSPSS as matchtigs. We use (r)SPSS to denote any of these representations based on the de Bruijn graph.

Approaches based on simplitigs encoding. Another improvement over SPSS focuses on encoding similar simplitigs using a larger alphabet, which requires fewer characters in total. ESS-compress [RCM21] encodes the k-mer set as a set of strings over the alphabet `ACGT[]+-`, where the non-alphabetic characters allow for efficient encoding of multiple occurrences of the same $(k-1)$-long patterns in the SPSS. Although this enhances compressibility, achieving about 27% improvement in compressibility over SPSS [RCM21], which is about the same as matchtigs [SKA+23], as k-mers are no longer substrings of the strings, it makes it more difficult to work with the representation directly without decompressing it first.

1.4 The Shortest Superstring Problem

The Shortest Superstring Problem (SSP) is a heavily studied string problem. Given a set of strings $S_1, S_2, \ldots, S_n$, the problem is to find a single string S that contains all the strings S_i as substrings and is the shortest possible such string. Such a string is then called the shortest superstring of the given set.

The SSP is known to be NP-hard even for the case of binary alphabet [GJ79] and despite a long history of research the best possible approximation ratio is widely open. The best-known approximation guarantee is about 2.466 [EMV23]. In the other direction, it has only been shown that computing a 1.003-approximation implies P=NP [KS13].

One of the widely used approximation algorithms for SSP is the *global greedy* algorithm called Greedy in the string algorithms literature. Greedy works as follows: it starts with the original set of strings and in each step, it merges the two most overlapping strings, breaking ties arbitrarily. This merging is done in a way that the result is the shortest string possible containing both the initial strings u and v in this order. Note that the size of this string is $|u| + |v| - |\mathtt{ov}(u, v)|$. The algorithm continues until there is only one string left, which is then a superstring of the original set. The approximation ratio of Greedy is known to be between 2 and 3.396 [EMV23] and is conjectured to be exactly 2 [TU88].

Other approximation algorithms for SSP are based on a graph reformulation of the problem. If we consider the overlap graph of the input strings, each superstring corresponds to a Hamiltonian path in the overlap graph and vice versa. It further holds that the longer the Hamiltonian path, the shorter the superstring. Therefore, SSP corresponds to the Maximum asymmetric travelling salesman problem (Max-ATSP). The algorithms based on the overlap graph, namely TGreedy [BJL+94] and approaches based on the Max-ATSP reformulation [KPS94; BJJ97] achieve better upper bounds on approximation ratios than Greedy, 2.698 and 2.466 respectively [EMV23]. Despite better approximation ratios, it is unknown whether these algorithms bring any real advantage over Greedy.

1.5 Data Structures for Strings

In this section we provide a basic overview of the data structures used in this thesis.

1.5.1 Aho-Corasick Automaton

The Aho-Corasick (AC) automaton [AC75] is an extension of a trie (prefix tree) constructed from the input words, in our case from a set of k-mers. As in a trie, every state corresponding to a prefix P is equipped with a forward function which for every letter x of the alphabet Σ points to the state corresponding to the prefix Px if it exists. Alongside the forward function, every state has also a fail function f. For a state s, $f(s)$ is the longest proper suffix of the state that is also a valid state. In the typical AC automaton, each state is also assigned an output function. However, in our use case the output function is not needed as no k-mer is a proper prefix on a different state and hence the AC automaton used here is always without the output function. The AC automaton can be constructed in linear time by first constructing the trie and then the failure function layer by layer. For a more detailed description, we refer to a standard algorithms textbook, e.g. [CLRS22]. For an illustration of the AC automaton over the `ACGT` alphabet, see Figure 1.2.

1.5.2 Suffix Array

Given a string S, consider all its suffixes and the lexicographical order of these suffixes. The suffix array [MM90] SA of S is then such that $\mathrm{SA}[i] = j$ if the suffix $S[j:]$ is the i-th in the order of all the suffixes. Suffix array can be constructed in linear time [KA03].

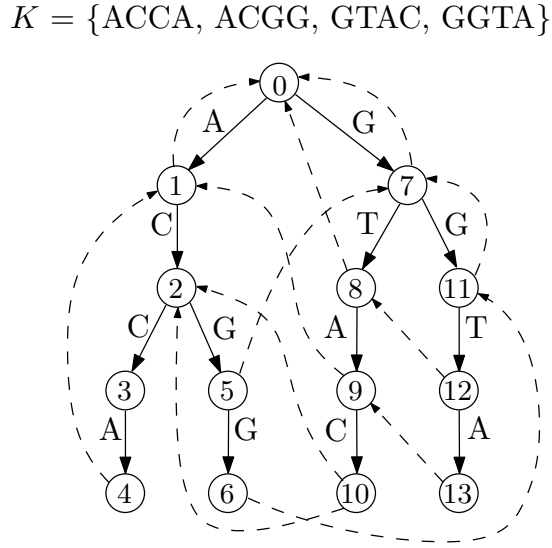


Figure 1.2: **An illustration of the AC automaton for a set of k-mers.** The AC automaton is constructed from 4-mers ACCA, ACGG, GTAC, and GGTA in the uni-directional model. Forward edges (solid) are labeled by letters ACGT, while the failure function f is depicted by dashed edges (which are not part of the trie of the k-mers). State 0 corresponds to the empty prefix and is the root of the trie, while, for example, state 2 corresponds to prefix AC and state 9 to prefix GTA.

Additionally, the lowest common prefix (LCP) array [MM90] can be constructed alongside the suffix array. The LCP array at a given position i contains the length of the longest common prefix of the i-th and $(i+1)$-th suffixes which are consecutive in the suffix array. LCP array can be built in linear time alongside the creation of the suffix tree [KA03] or can be computed in linear time directly from the string and the suffix array [KLA+01].

Suffix and LCP arrays can be together used for fast indexing in constant time. The only drawback of suffix and LCP arrays is that they require $\Theta(n \log n)$ bits to store unlike the FM index (see Section 1.5.5) which for constant-sized alphabets requires only $\Theta(n)$ bits to store.

For an illustration of SA and the LCP array, depicting their relationship to the Burrows-Wheeler transform (Section 1.5.4), see Figure 1.3.

1.5.3 Rank and Query Support

Given a string S over the alphabet Σ, except for querying the i-th character of S, there are two additional fundamental query operations that are often needed: the rank and the select operation. The rank operation $\texttt{rank}_c(i)$ returns the number of occurrences of c in the first i characters of S. Conversely, the select operation $\texttt{select}_c(i)$ returns the position of the i-th occurrence of c in S. The rank and select support can be efficiently implemented in the case of bit vectors and in order to generalize this to general alphabets, we use the wavelet trees [GGV03].

Input: CAGGTAG$

SA	LCP	Burrows-Wheeler Matrix							
7		$	C	A	G	G	T	A	G
	0								
5		A	G	$	C	A	G	G	T
	2								
1		A	G	G	T	A	G	$	C
	0								
0		C	A	G	G	T	A	G	$
	0								
6		G	$	C	A	G	G	T	A
	1								
2		G	G	T	A	G	$	C	A
	1								
3		G	T	A	G	$	C	A	G
	0								
4		T	A	G	$	C	A	G	G

Figure 1.3: **Example of the relationship between SA and BWT.** For a sample input string CAGGTAG$ its suffix array (SA), longest common prefix array (LCP), Burrows-Wheeler matrix and Burrows-Wheeler transform (in blue). Note that the LCP array corresponds tho the length of the longest prefix shared by the two consecutive rows in the Burrows-Wheeler matrix. The i-the element of SA is the index of the first character of the corresponding row of the Burrows-Wheeler matrix. It is also the position of the $ in the corresponding row from the rear.

Rank and select on bit vectors. Note that in the case of bit vectors, it is easy to compute $\texttt{rank}_0(i)$ from $\texttt{rank}_1(i)$ as $\texttt{rank}_0(i) + \texttt{rank}_1(i) = i$. Given a bit vector, it is possible to determine $\texttt{rank}_1$ of each position in a constant time with sublinear-space data structure [Jac88; Jac89]. The main idea behind this is to twice partition the bit vector into blocks of small sizes, store the ranks only for the beginnings of each block and inside the block store only the relative ranks. The result can then be computed as the sum of the rank from the beginning of the current block and the relative rank inside the block. Using a similar approach, it is also possible to compute the select in constant time with sublinear space [Cla97].

There have also been several practically efficient implementations of rank and select on bit vectors which typically do not give sublinear additional overhead but instead a small linear factor overheads which is typically indistinguishable in practice [GGMN05; NP12].

Wavelet trees. Wavelet trees [GGV03] enable the rank and select data structures to be applicable to general alphabets. In a wavelet tree, the leaves correspond to the letters of the alphabet and they do not contain anything else. In each internal node there is a bit vector of length equal to the number of occurrences of all the letters in the subtree. At the i-th position of such a bit vector, there is a 0

if the i-th occurrence in S among the letters from the considered subtree is from the subtree corresponding to the left child and 1 if from the right child's subtree.

On such a structure, we can determine the i-th character of S by traversing the corresponding path to the leaf, while updating the queried index using rank queries. Similarly, we can determine the rank by traversing from the root to the leaf and select by going in this case from the leaf to the root. These operations have the time complexity linear in the height of the tree, which in the case of constant-sized alphabets is constant.

Typically, the bit vectors in wavelet trees are compressed, which in the case of genomic data does not help much, as genomic data are in general not very well compressible. Furthermore, for the `ACGT` alphabet without any compression, it is easy to see that the bit vectors occupy 2 bits per character.

1.5.4 Burrows-Wheeler Transform

Assume we are given a string S over an alphabet Σ with a special termination symbol \$ that is considered to be alphabetically smaller than all the symbols in Σ and we modify S by appending \$ to it. Consider all the rotations of S and sort them lexicographically. Note that since \$ appears exactly once, the order is unambiguous. If we put all these rotated strings in a matrix, where the rows correspond to the lexicographical order, we obtain the Burrows-Wheeler matrix. The last column of this matrix is the Burrows-Wheeler transform (BWT) of S [BW94]. Since every character of S appears as the last character of one of the rotations of S, BWT is a permutation of S.

BWT of S can be computed in linear time from suffix arrays. In order to do this, first notice that the order of the rotations is the same as of the corresponding suffixes. Hence if we have computed the suffix array SA, the first character in the i-th row of the Burrows-Wheeler matrix is the SA$[i]$-th character in S. Thus the i-th character of BWT (i.e., the last character in the i-th row of the Burrows-Wheeler matrix) is the (SA$[i] - 1$)-th character of S (considered cyclically).

An additional key property of the BWT is reversibility in linear time for constant-sized alphabets [BW94]. To do so, we need apart from the last column of the Burrows-Wheeler matrix (which is the BWT itself) also the first column, which is easy to get based on the frequencies of each letter since it is by definition lexicographically sorted. The only remaining observation is that the i-th occurrence of a given character in the first and last column correspond since they are both sorted based on their right context. With this, for each letter in the last column we can get position of the letter in the first column – it is the j-th occurrence of the letter where j is the rank of the given letter in the last column. This mapping is referred to as the LF mapping. To revert the BWT, we reconstruct the string from the end, starting with the \$ character. We then start in the first row and note that the character in the last column is the one preceding the character in the first column. By the LF mapping, we can get to the occurrence of this character in the first column and by repeating reconstruct the whole string.

For an example of BWT, compared to the SA and the LCP array, see Figure 1.3. For an example of the LF mapping and how BWT is reverted, see Figure 1.4.

Although BWT was designed for lossless compression, it can be used in the core of a very efficient data structure for indexing (see Section 1.5.5).

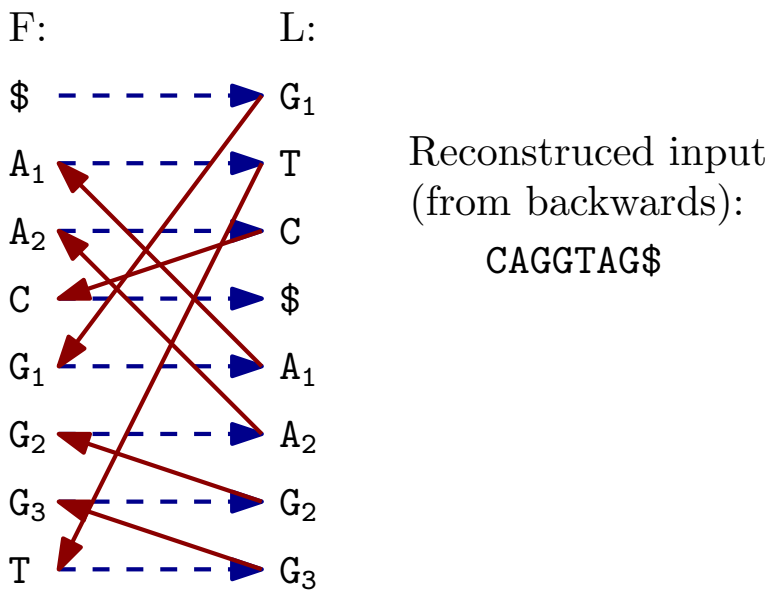


Figure 1.4: **Example of the LF mapping and the BWT reversal.** The first (F) and last (L) columns of the Burrows-Wheeler matrix, where the last column is the Burrows-Wheeler transform (BWT). The red solid arrows show the LF mapping, which maps each character to its position in the first column. The blue dashed lines connect the letters in the same row, where the character in L is the one preceding the letter in F in the original text. By starting in the first row (the row starting with the `$`), traversing along the blue and red edges, and storing the encountered letters, we reconstruct the original text from backwards.

1.5.5 FM-Index

It is possible to use the Burrows-Wheeler transform to build a full-text search over the original sequence using a data structure called FM-index [FM05]. Similarly to the suffix array (see Section 1.5.2), it can be used for fast indexing. However, the main advantage of the FM-index is that unlike the suffix array it requires only linear number of bits to store (for constant sized alphabets).

The core of the FM-index is the Burrows-Wheeler transform of the input string S equipped with the rank function, that is typically referred to as the occurrence function (Occ) in the terminology of the FM-index. Note that a possible implementation of this is the wavelet tree. Furthermore, for each character, the number of characters lexicographically smaller than it which occur in S is stored.

In order to search for a pattern P in S, we search for the pattern from back to front and we realize that each such suffix of P corresponds to a prefix of a row in the Burrows-Wheeler matrix and furthermore, since the rows in the Burrows-Wheeler matrix are sorted lexicographically, the occurrences of each suffix of P

correspond to just a single range of rows in the Burrows-Wheeler matrix. We only need to specify how to obtain the new range by prepending a letter to the searched pattern. Note that for each occurrence of the prolonged pattern, the first letter appears in the last column of the Burrows-Wheeler matrix in the range of the occurrences of the previous pattern. With this observation, using the rank and total number of lexicographically smaller characters (which is essentially the LF mapping), we can update the range in constant time.

This gives us the range in the rows of the Burrows-Wheeler matrix that correspond to the occurrences of the pattern P. We can use this to easily check whether a pattern appears and even how many times. However, if we want to find the positions of the occurrences in S, we need an additional data structure, the sampled suffix array, which stores the suffix array value for some of the rows in the Burrows-Wheeler matrix. The others can be then computed from those and using the rank. Note however that this translation to the original coordinates is the most time-consuming part of the FM-index search.

The FM-index can also be used to store more than a single string. In such a case, we typically concatenate the strings and store at which positions a new string starts. Querying whether a pattern appears in any of the strings, we use the backward search as before, however we need to translate each occurrence into the original coordinates as it could be that the occurrence spans over two strings which is not a valid occurrence.

Moreover, additional variants of FM-index have been proposed, most notably the bidirectional FM-index (the bi-directionality of FM-index is unrelated to the bi-directionality of k-mers) which allows for searched pattern to be extended in both directions [LLT+09].

1.6 Data Structures for k-Mer Sets

A large body of work focused on data structures for k-mer sets and their collections; we refer to [CHM21; MBP+21] for recent surveys. One approach for individual k-mer sets is to combine the aforementioned textual k-mer set representations based on the de Bruijn graph of the set [CLJ+14; BBK21; RM21; SKA+23], i.e., (r)SPSS, with efficient string indexes such as the FM index (Section 1.5.5) or BWA [LD09; LD10; Li12; Li13] (see also ProPhex [SBPK; Sal17]). Another approach based on BWT is the BOSS data structure [BOSS12] that was implemented as an index for a collection of k-mer sets in VARI [MBN+17], VARI-merge [MAB19], and Metagraph [KMD+20]. Taking BOSS as an inspiration, Spectral Burrows-Wheeler Transform (SBWT) [APV23] has been proposed as a compact representation of the de Bruijn graph, together with various approaches for its indexing. Themisto [AVMP23] further builds on SBWT and provides an index for collections of k-mer sets.

Hashing techniques have also been successful in the design of k-mer data structures. In particular, Bifrost [HM20] uses hash tables for indexing colored de Bruijn graphs, which encode collections of k-mer sets. Other popular data structures, such as BBHash [LRCP17], BLight [MKL21], and SSHash [Pib22], employ minimal perfect hash functions and serve as a base for constructing indexes such as FDBG [CKB+18], REINDEER [MIG+20], Fulgor [FKS+24], and pufferfish2 [FKPP23].

While the aforementioned approaches yield exact data structures, further space compression may be achieved by employing probabilistic techniques and allowing for a certain low-probability error, such as false positives. Namely, the counting quotient filter was used for k-mers in Squeakr [PBJP18], and this data structure was extended to an efficient index called dynamic Mantis [AKM+22]. Another line of work, e.g., [BDR+19; BBGI19; GYC+21; LMCP22], employs variants of the Bloom filter to further reduce space requirements. Recently, Invertible Bloom Lookup Tables combined with sub-sampling have been used to estimate the Jaccard similarity coefficient [SBK22].

Very recently, the Conway-Bromage-Lyndon (CBL) data structure [MCLM24] builds on the work of Conway and Bromage [CB11] and combines smallest cyclic rotations of k-mers with sparse bit-vector encodings, to yield a dynamic and exact k-mer index supporting set operations, such as union, intersection, and difference. Finally, we note that set operations can also be carried out using some k-mer lists and counters, e.g., [KLR15; KDD17]; however, these methods are unable to exploit structural properties of k-mer sets such as the SLP.

2 Masked Superstrings as a Representation of k-Mer Sets

In this chapter, we introduce the concept of masked superstrings for representing sets of k-mers and demonstrate that they unify and generalize all previous textual k-mer set representations.

Definition 1. *Given K a set of k-mers, a pair (S, M) where S is a superstring of all k-mers in K and M is a binary mask of the same length L is called a* masked superstring *(or MS for short) representing K if it satisfies*

$$K = \{S_i \dots S_{i+k-1} \mid M_i = \mathtt{1}, i \in \{0, \dots, L-k\}\}.$$

S alone is called the k-mer superstring and M its mask. By convention, the last $k-1$ characters of M are always set to zero.

We call all the k-mers that appear as a substring in S as the *appearing k-mers*. An appearing k-mer can have multiple occurrences in S and we call an occurrence ON if the bit corresponding to the occurrence, that is the bit at the position of the first character of the k-mer, is set to `1`. Conversely, an occurrence is OFF if there is a `0` at the corresponding place in the mask. The k-mer set that we aim to represent are then all the appearing k-mers with at least one ON occurrence. We call those the *represented k-mers*. There are also appearing k-mers that are not represented, i.e., they have no ON occurrence, we call them the *ghost k-mers* and typically denote them by X.

To study the properties of masked superstrings, we also introduce the following definitions, which are analogs to the definitions of represented and ghost k-mers for multisets of their occurrences.

Definition 2. *For a masked superstring (S, M) of length ℓ that represents a set K of k-mers, we define the following multisets.*

$$\mathbb{K} = \left\{S_i \cdots S_{i+k-1} \mid M_i = \mathtt{1},\ i \in \{0, \dots, \ell-k\}\right\}$$
$$\mathbb{X} = \left\{S_i \cdots S_{i+k-1} \mid M_i = \mathtt{0},\ i \in \{0, \dots, \ell-k\} \text{ and } S_i \cdots S_{i+k-1} \notin K\right\}$$
$$\mathbb{Y} = \left\{S_i \cdots S_{i+k-1} \mid M_i = \mathtt{0},\ i \in \{0, \dots, \ell-k\} \text{ and } S_i \cdots S_{i+k-1} \in K\right\}$$

That is, $\mathbb{K}$ contains exactly the k-mers in K, with the multiplicity of a k-mer a in $\mathbb{K}$ equal to the number of ON occurrences of a in S. Similarly, $\mathbb{X}$ contains all ghost k-mers of (S, M), each with multiplicity equal to the number of its total occurrences which are all OFF. Finally as it is useful to talk about OFF occurrences of represented k-mers, $\mathbb{Y}$ contains the represented k-mers with at least one OFF occurrence in the superstring, with multiplicity equal to the number of its OFF occurrences.

Furthermore, as a represented k-mer can have multiple occurrences but it is sufficient to have at least one ON occurrence, it follows that for a fixed superstring S and a k-mer set K, there can be multiple masks M such that (S, M) represents K. We call these masks *compatible masks*. Observe that if we equip the set of all

compatible masks with a partial order which is the inclusion on the positions set to 1, then the partially ordered set of all compatible masks forms a semi-lattice where meet is the intersection. As a practical consequence of the non-uniqueness of masks, they can be optimized for specific applications as we demonstrate in Section 3.2.

In Definition 1, we defined masks to be of the same length as the superstrings even though the last $k-1$ bits in the mask are redundant. This has several practical advantages. The first being that as the k-th last bit of the mask is set to 1 in typical usage, the masked superstring alone stores all the information about the k-mer set, because k can be inferred from the number of trailing zeros in the mask.

As outlined in Fig. 2.1d, there are several ways to store a masked superstring in a text file. For (r)SPSS, one may just join all the strings by a delimiter, such as the newline character (see enc0 in Fig. 2.1d); however, this does not always work for the more general masked superstrings. The most obvious encoding of masked superstrings is to first write down the superstring (over the ACGT alphabet) and then append the binary mask (enc1 in Fig. 2.1d). For practical purposes, it is often convenient to combine them into a single string, that we call *mask-cased superstring*, over the ACGTacgt alphabet, where uppercase letters represent 1 in the mask and lowercase letters correspond to 0 (enc2 in Fig. 2.1d). Finally, while a superstring is typically not much compressible beyond two bits per character, masks tend to be quite sparse, having relatively few runs of consecutive 0s (under SLP). Thus, one may apply the run-length encoding (RLE), which is a sequence of lengths of maximal runs of consecutive 1s or 0s in the mask (enc3 in Fig. 2.1d).[1]

Example. Consider the set of 3-mers $K = \{\texttt{ACG}, \texttt{GGG}\}$ of the k-mers to be represented and the superstring ACGGGG resulting from their concatenation. There are three compatible masks – 101100, 101000 and 100100 – since each of the two k-mers must have at least one ON occurrence, and the occurrence of the ghost k-mer CGG must always be OFF. With the last mask, the masked-cased encoding would be AcgGgg. Conversely, when parsing AcgGgg as a masked superstring, we can deduce from its suffix that $k = 3$ and then decode the set of represented k-mers $\{\texttt{ACG}, \texttt{GGG}\}$.

2.1 Relation to (r)SPSS Representations

There are two different viewpoints in which masked superstrings can be seen as a generalization of (r)SPSS representations. One entirely based on strings and another based on graphs of the k-mer sets.

In the (r)SPSS representations a set of k-mers is represented as a set of strings with the two following properties (with some representations requiring additional properties):

[1] According to the mask format convention, the last run of consecutive characters has length $k-1$ and consists of 0s only, and we thus omit its length from encoding enc3. After that, enc3 still implicitly determines the value of k, since the superstring length ℓ minus the sum of all run lengths equals $k-1$. Finally, while enc3 does not explicitly specify whether the first (or any other) run consists of 1s or 0s, it can be computed from the parity of the number of runs in enc3 (excluding the final run of 0s): if this number is odd, the first run consists of 1s, otherwise the first run contains 0s. Hence, the full binary mask can be recomputed from enc3.

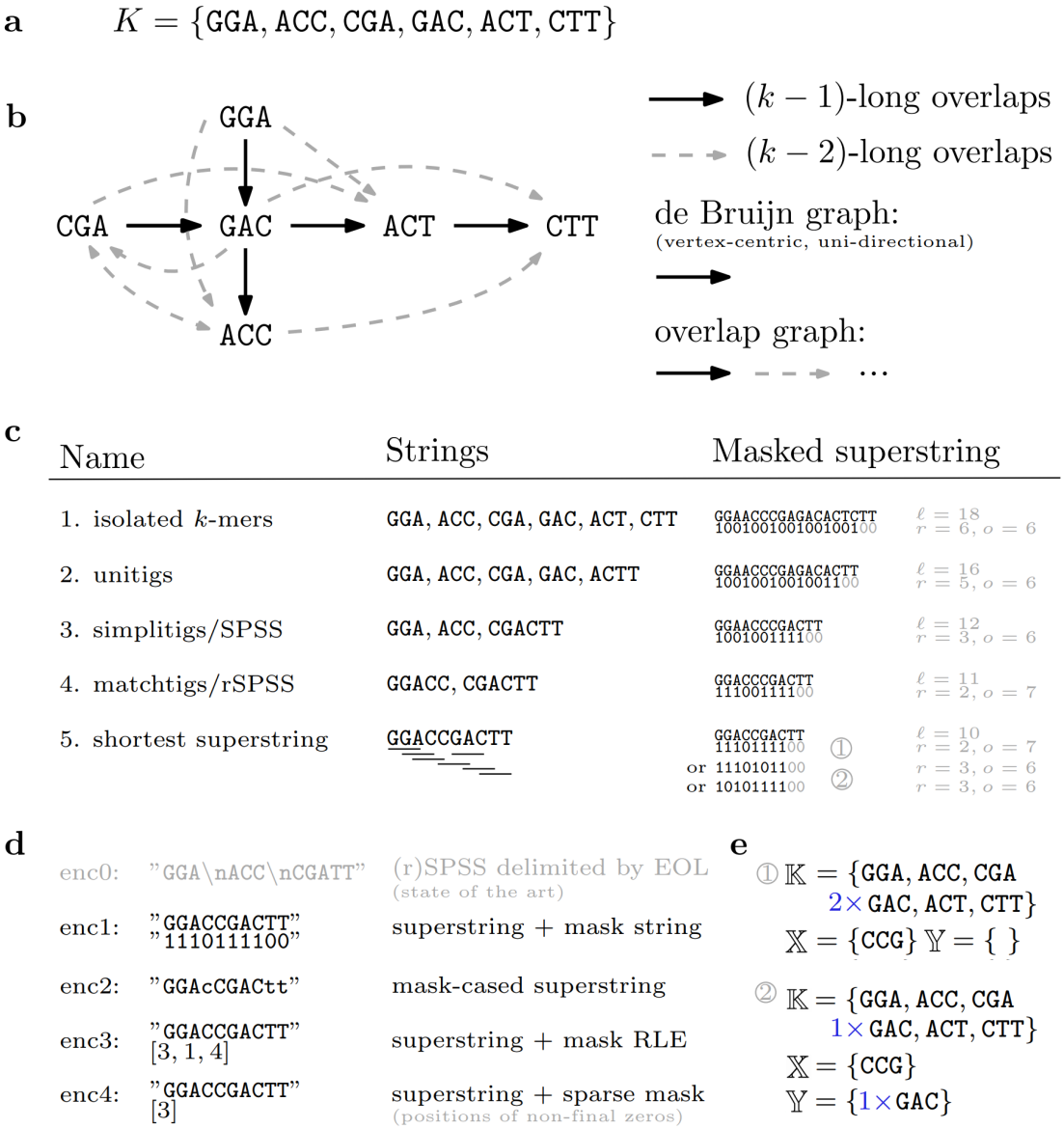


Figure 2.1: **The concept of masked superstrings for representing k-mer sets.** We focus on the uni-directional model for simplicity. **a)** An example of a k-mer set ($k = 3$). **b)** The corresponding de Bruijn (solid edges) and overlap graphs (solid and dashed edges, respectively). **c)** Individual representations of the k-mer set and the corresponding masked superstring, sorted with respect to the length. Value ℓ is the superstring length (generalizing the cumulative length in (r)SPSS), r is the number of runs of ones (generalizing the number of sequences in (r)SPSS), and o is the total number of ones in the mask. Note that the individual strings in c1–c4 could be concatenated in different orders, which would result in different sets of ghost k-mers. **d)** Examples of encodings of masked superstrings (for all encodings, it is possible to use 2-bit or 3-bit representation of individual characters). **e)** The k-mer multisets $\mathbb{K}$, $\mathbb{X}$, and $\mathbb{Y}$ (see Def. 2) for masked superstrings ① and ② from c5.

- Every k-mer appears as a substring of at least one string.
- Every substring of length k is a represented k-mer.

Furthermore, note that optimal matchtigs even compute optimal such representations with respect to the total length of the strings [SKA+23].

Under this setting, the masked superstring framework can be viewed as relaxing the second condition to enable even smaller total length while still enabling to recover the original k-mer set by introducing masks.

In the graph-based view, all (r)SPSS representations correspond to paths in the de Bruijn graph of the k-mer set such that each vertex is covered by at least one of the paths. Here, masked superstrings can be also seen as a path covering all the vertices of the graph, but in this case the graph is not the de Bruijn graph, but the overlap graph of the k-mer set, where we allow edges between k-mers of all overlap lengths, not just $k-1$.

Most importantly, what truly enables masked superstrings to generalize all (r)SPSS representations is the fact that every (r)SPSS representation can be directly viewed as a masked superstring.

Theorem 1. *For every (r)SPSS representation $\mathcal{R}$ representing a k-mer set K, there exists a masked superstring (S, M) representing the same set K such that the cumulative length of $\mathcal{R}$ is the same as length of S.*

Proof. Let $S_1, \ldots, S_r$ be the (r)SPSS representation, i.e., a set of strings such that every k-mer in K is a substring of at least one string in $\mathcal{R}$ and vice versa. For each such string S_i we take a masked superstring where the superstring is S_i and the mask is all ones except for the last $k-1$ positions and as a masked superstring (S, M) corresponding to $S_1, \ldots, S_r$ we consider the concatenation of all these masked superstrings. To verify that this represents the same set in the point of view of masked superstrings, observe that every k-mer in K has at least one ON occurrence and conversely no k-mer outside of K is represented. As the masked superstring is a concatenation of the strings, its length matches the cumulative length of the (r)SPSS representation. □

Theorem 1 shows that masked superstrings unify (r)SPSS representations from the theoretical perspective. However, regarding the practicality of such generalization, a natural objection arises – we have unified the representations, but at a cost of introducing a completely new concept of a mask which needs to be stored. Yet, in (r)SPSS representations we have to delimit the sequences somehow. In practice it is typically done by concatenating the strings and storing separately at which position each string starts. This, however, can be viewed as a specific compressed form of a mask. Hence masks were present even in the (r)SPSS representations in some form, they were just not explicitly recognized as such.

Furthermore, objectives which were typically used with (r)SPSS representations, namely the total cumulative length of the strings and the number of strings, naturally translate to the masked superstrings framework. The cumulative length corresponds to the length of the superstring, as directly follows from Theorem 1, and the number of strings to the number of runs of consecutive ones in the mask. In this regard, masked superstrings not only unify the (r)SPSS representations, but they also generalize them, since although for any masked superstring we can find a (r)SPSS representation representing the same set, all those may be longer

that the masked superstring. Hence as Theorem 1 gives us for any set of strings a corresponding masked superstring, the converse, i.e., to obtain a set of strings from a masked superstring with the same properties, is not always possible.

To see an illustration of this correspondence, see Figure 2.1. Furthermore, to see an overview of the textual representations with their restrictions and corresponding algorithms, see Table 2.1.

Representation	**Restriction on** $S = s^{(1)} + ... + s^{(p)}$	**Restriction on** M	**Restriction on** $\mathbb{K}$	**Optimality**	**Algorithms** (optimal)	**Heuristics** (suboptimal)
k-mer list	$\forall i : \lvert s^{(i)}\rvert = k$ and $\forall i \neq j : s^{(i)} \neq s^{(j)}$	runs of 0s of len. $k-1$ and runs of 1s of len. 1	$\max_{a \in K} f_{\mathbb{K}}(a) = 1$	(trivial)	(trivial)	(optimal by definition)
unitigs [CLJ+14]	terminating $s^{(i)}$ whenever multiple k-mer extensions admitted	runs of 0s of len. $k-1$	$\max_{a \in K} f_{\mathbb{K}}(a) = 1$	$\min \lvert S\rvert$ (equiv.: $\min p$)	optimal unitigs [CLJ+14; CLM16] (in theory linear*)	(none)
simplitigs [BBK21] (SPSS [RM21])	(**none**)	runs of 0s of len. $k-1$	$\max_{a \in K} f_{\mathbb{K}}(a) = 1$	$\min \lvert S\rvert$ (equiv.: $\min p$)	eulertigs [SA23] (in theory linear*)	• PropHAsm [BBK21] (linear†) • UST [RM21] (in theory linear*)
matchtigs [SKA+23] (rSPSS)	(none)	runs of 0s of len. $k-1$	(**none**)	$\min \lvert S\rvert$	optimal matchtigs [SKA+23] (polynomial)	• greedy matchtigs [SKA+23] (polynomial)
masked superstring	(none)	(**none**)	(none)	app-specific Table 3.1	brute-force search (exponential)	• BiDir-LocalGreedy (Section 3.1.2) (linear†) • BiDir-GlobalGreedy (Section 3.1.3) (linear†)

The constraints are formulated over the superstring S (and its components $s^{(1)}, \ldots, s^{(p)}$ for (r)SPSS), the k-mer mask M, the multiset $\mathbb{K}$ (see Def. 2) of all represented k-mers with their frequencies (by $f_{\mathbb{K}}(a)$ we denote the frequency of a in $\mathbb{K}$), the optimality criteria, the algorithms for computing the optimal representation, and the associated heuristics, including their time complexities. In bold, we highlight the change of the restrictions from the previous representation.

† Linear: The involved algorithms run in time linear to the total size of k-mers when represented as text, that is $\mathcal{O}(kn)$.

*In theory linear: The involved algorithms could in theory be linear, however their implementations involve the use of BCALM2 for computing unitigs, and as such do not have the guarantee of linearity.

Remark about restrictions on $\mathbb{X}$ and $\mathbb{Y}$ (see Def. 2): None of these representations restricts the multisets $\mathbb{X}$ and $\mathbb{Y}$ of k-mers on OFF positions of the masked superstring in any way (for representations with more components, $\mathbb{X}$ is defined only if those components are concatenated into a superstring). We leave a study of restrictions on $\mathbb{X}$ and $\mathbb{Y}$ to a future work.

Table 2.1: Overview of textual k-mer set representations and their constraints.

3 On Masked Superstrings Computation

As masked superstrings are more general than (r)SPSS representations, they allow for far more optimization objectives than the number of sequences and cumulative length that were used with (r)SPSS. As an upside, the objective can be chosen specifically based on the needs of downstream applications and hence masked superstrings can be tailored specifically to the use-case. To get an idea of objectives which can be used with masked superstrings, see Table 3.1.

The downside of this generality is that finding optimal masked superstrings becomes a more complex problem, which is generally NP-hard, for instance if we consider as an objective the length of the superstring. As a result there is no general polynomial algorithm that would compute masked superstrings optimally for all objectives. Therefore, we propose a simplified two step protocol for optimizing masked superstrings. First, we optimize the superstring with respect to its length (Section 3.1) and then we optimize the mask with objectives depending on the original objective (Section 3.2). In Section 3.3, we show that this protocol in fact yields an approximation algorithm for many objectives.

Minimization objective	**Correspondence**	**Complexity class**	**Obtained polynomial approximability**
$\lvert S\rvert$	• corresponds to the cumulative length of (r)SPSS • storage size if stored as bit vectors	NP-complete (Section 3.1.1)	$\mathcal{O}(1)$ approx. ratio (Section 3.3; by the two-step optimization protocol)
$\texttt{runs}_1(M)$	• corresponds to the number of sequences in (r)SPSS	polynomial (Appendix A)	optimal
$\lvert S\rvert + r \cdot \texttt{runs}_1(M)$	• storage size if mask stored using RLE for $r = \log_2 \lvert S\rvert$	*unknown*	$\mathcal{O}(r)$ approx. ratio (Section 3.3; by the two-step optimization protocol)
$\lvert S\rvert + r \cdot \lvert M\rvert_0$	• storage size if mask stored as positions of zeros for $r = \log_2 \lvert S\rvert$	*unknown*	$\mathcal{O}(r)$ approx. ratio (Section 3.3; by the two-step optimization protocol)

Table 3.1: **Overview of objectives for masked superstrings and their complexity.** The objectives are stated in terms of the length of superstring S and properties of its mask M.

3.1 The First Step: Superstring Optimization

In the first step of the protocol for masked superstring computation, we compute a superstring of the input k-mer set. Unfortunately, this first step, finding the shortest common superstring of a set of k-mers, is NP-hard even for k-mers as we show in Theorem 2.

As the problem is NP-hard, we turn to approximation algorithms. In fact, we remark that any reasonable algorithm for computing textual representations of k-mer sets are approximation algorithms with ratio at most k as representing k-mers as their concatenation yields a superstring of length at most k times the size of the set, which is also a natural lower bound on the size of the superstring. We then propose two different algorithms for superstring computation, one being a generalized version of ProphAsm [BBK21] for computing simplitigs and the

other being a modification of the GREEDY algorithm to the bi-directional model.

3.1.1 NP-Hardness of Finding Minimum Superstrings

Theorem 2. *Given a set of k-mers K, with $k > \log_2 |K|$, finding the shortest superstring for K is NP-complete in both the uni-directional and bi-directional models.*

We show this via a reduction from a specific version of the shortest superstring problem which is known to be NP-complete. Note that this does not immediately yield the NP-hardness for sets of k-mers, as in the bi-directional model we consider a k-mer and its reverse complement as equivalent.

Proposition 3 (Theorem 3 in [GMS80])**.** *The shortest superstring problem is NP-hard even if the input strings are over the binary alphabet and have the same length k, for any $k > \log_2 n$, where n is the number of input strings.*

Proof of Theorem 2. To show NP-hardness, consider an input of SSP under the conditions of Proposition 3. We transform this input to a set of k-mers K only over alphabet $\{\mathtt{A}, \mathtt{C}\}$ by mapping each `1` to `A` and each `0` to `C`, which immediately yields NP-hardness for the uni-directional model. To see that this also holds for the bi-directional model, we note that the overlap between any k-mer in K and any reverse complement of k-mers in K is zero as the reverse complements are over the alphabet $\{\mathtt{T}, \mathtt{G}\}$. Hence any superstring in the bi-directional model can be modified to a superstring in the uni-directional model with the same superstring length by considering reverse complements of maximal segments of `T`'s and `G`'s. Since a k-mer from K could not occur at the boundary of such a segment, this transformed superstring is still a superstring of K, now also in the uni-directional model. This proves the NP-hardness even for the bi-directional model. To see that the decision version of the problem also lies in NP, we remark that the superstring itself can serve as a certificate, since containment of each k-mer can be verified in polynomial time.

□

We further remark that even though typically a small, fixed values of k are used in practice, the proof of NP-hardness requires k to be arbitrarily large. However, k must be at least $\log_4 |K|$ (and even slightly more in the bi-directional model), otherwise we would get a contradiction as $|K| \leq 4^k < 4^{\log_4 |K|} \leq |K|$ which is not possible. Therefore, the provided result is tight up to a constant factor.

3.1.2 Local Greedy Algorithm

We present a novel algorithm which we call BIDIR-LOCALGREEDY for computing a masked superstring of a set of k-mers. The algorithm is a generalization of PROPHASM [BBK21] for computing simplitigs. ProphAsm works as follows. It starts with an arbitrary k-mer and finds a maximal path in the vertex-centric de Bruijn graph containing this k-mer by iteratively extending it to both directions. The k-mers covered by this maximal path are then removed from the set and the process is repeated until the set is empty.

Local greedy generalizes this approach by finding paths not in the de Bruijn graph, but in overlap graphs. Instead of requiring each extending k-mer to have overlap of $k-1$ characters, we allow extending by up to $d_{\max}$ characters, i.e. we require consecutive k-mers to have overlaps only at least $k-d_{\max}$. Therefore, local greedy can be also seen as to work on a subgraph of the overlap graph containing edges of weight at least $k-d_{\max}$.

In more detail, local greedy proceeds as follows. It starts with K being initially the k-mer set we aim to represent. Local greedy then picks an arbitrary k-mer $a \in K$, removes it, and initializes a superstring segment $S_P = a$. It then iteratively extends S_P to the left or to the right. A *left extension* of S_P by d characters is a k-mer b such that the last $k-d$ characters of S_P are equal to the first $k-d$ characters of b. Analogously we define a *right extension* by d characters.

In every step of S_P extension, local greedy picks the extension (either left or right) with the least value of d. If $d \leq d_{\max}$, the extension is added to S_P and the k-mer is removed from K. Otherwise, we have a maximal path in the overlap graph using edges with overlap at least $k-d_{\max}$, we append the string segment to the superstring and repeat the process until we empty the set K.

Note that this algorithm works also in the bi-directional model, we just check whether k-mer or its reverse complement are in K and if we extend, we remove both k-mer and its complement. Furthermore, even though the algorithm as described computes only the superstring, it can be modified to produce also masked superstrings if we put `1` at the starting position of each k-mer considered when extending.

In Sections 3.1.4 and 3.1.5, we provide two implementations of local greedy. First using simple extensive enumeration and k-mer hashing. Second using the Aho-Corasick automaton as an underlying data structure. The Aho-Corasick-automaton-based variant achieves total linear time complexity as compared to the exponential in $d_{\max}$ complexity of the hashing variant. However, for values of $d_{\max}$ which are small enough, the hashing variant is faster in practice.

3.1.3 Global Greedy Algorithm

We modify the global greedy algorithm for the SSP, typically known in the literature as Greedy (see Section 1.4) to work in the bi-directional model. We call the resulting algorithm BiDir-GlobalGreedy. We note that in the uni-directional model Greedy can be used directly to compute a k-mer superstring. It could also be used in the bi-directional model as the computed superstring is also a superstring in the bi-directional model, however, it does not use the overlaps between k-mers and their reverse complements which as a result yields unnecessarily longer superstring.

Therefore, we modify the global greedy algorithm for the bi-directional model so that it utilizes the possibility to merge a k-mer with a reverse complement (RC) of another k-mer and to represent either the original k-mer or its RC but not necessarily both. We then show that the adjusted algorithm can still be implemented in linear time.

For each k-mer present in the k-mer set, we consider both the k-mer and its RC, and we forbid using the two edges between them (forbidding edges between a k-mer and its RC allows us to maintain a certain useful invariant as we show

below). Similarly as in the uni-directional model, our aim is to greedily construct a Hamiltonian path H in the overlap graph. However, in the bi-directional model we ensure that no k-mer and its RC both appear in the Hamiltonian path; the aim is to make the resulting superstring possibly shorter. In other words, we adjust the Hamiltonian requirement so that the resulting path contains each k-mer or its RC but not both[1]. In fact, our algorithm constructs two "reverse complementary" paths P and P' such that if P contains edge (a, b), path P' has edge (b', a'), where b' is the RC of b and a' is the RC of a. In the following, the edge set H will be the union of P and P'.

We therefore modify the global greedy algorithm as follows:

- We maintain a set of chosen edges H in the overlap graph for the k-mer set. The RCs of the k-mers in the original sequence are also included in the set.
- In each step, we choose a largest-overlap edge from k-mer a to k-mer b such that a has outdegree 0 in H, b has indegree 0 in H, the edge does not close a cycle in H, and b is not the RC of a (breaking ties arbitrarily). Letting a' denote the RC of a and b' the RC of b, we add edges (a, b) and (b', a') to H.

Alternatively, BiDir-GlobalGreedy can be described by merging strings: In each step t, there is a set $\mathcal{S}_t$ of strings to merge (initially, this is the set of k-mers and their RCs). In $\mathcal{S}_t$, choose two different strings a and b such that their overlap is the longest (when merging a to b in this order) and b is not an RC of a, breaking ties arbitrarily. Letting a' denote the RC of a and b' the RC of b, we merge a to b and also merge b' to a'. This way, BiDir-GlobalGreedy maintains the following invariant: *In every step t, it holds that for each string $s \in S_t$, the RC of s is also present in $\mathcal{S}_t$* (formally, this can be shown by mathematical induction). This invariant in particular implies that in each step, b' has outdegree 0 and a' has indegree 0.

While global greedy in the uni-directional model outputs a single Hamiltonian path, in the bi-directional model we end up with two disjoint paths of the same length, which correspond to two reverse complementary superstrings; this follows directly from the aforementioned invariant.

The linear-time implementation of Greedy using the Aho-Corasick automaton [Ukk90] can be extended to handle our modification as we show in Section 3.1.5. Furthermore, in Section 3.1.4 we describe a simpler hashing-based implementation.

3.1.4 Hash-Table-Based Implementations

We now describe hash-table-based implementations of BiDir-LocalGreedy and BiDir-GlobalGreedy. The hashing-based variant of global greedy is in direct correspondence to the Aho-Corrasick automaton-based version, whereas the hashing-based version of local greedy can be seen more as an exhaustive enumeration.

From a theoretical point of view, the hashing-based implementations do not achieve worst-case linear-time complexity, unlike the automaton-based imple-

[1]While we restrict the output path in the overlap graph in this way, a similar property does not hold for the resulting superstring S obtained by merging k-mers along the path. Indeed, a k-mer a on the path may appear more times in S or the RC of a may become a substring of S because of merging some other k-mers that are adjacent in the path.

mentations. Specifically, for global greedy, linear-time complexity is achieved in expectation, while the hashing-based implementation of local greedy requires time exponential in the parameter $d_{\max}$. Nevertheless, in practice both versions are more efficient than their automaton-based counterparts (for local greedy, this holds for small-enough values of $d_{\max}$).

A major advantage of hashing-based implementations is that the memory requirements are much lower than of the automaton, since a k-mer for typical values of k can be represented as an integer. However, this requires a limit on the value of k; for example, with 64-bit representations it only works for $k < 32$.

Hashing-based implementation of local greedy. We provide a pseudocode of BiDir-LocalGreedy in Algorithm 1. We implement searching for a left or right extension in a rather straightforward way by exhaustively enumerating all possible length-d strings and checking whether this gives a valid extension for the current path. This enumeration takes time 4^d, but as we first start with $d = 1$ and keep increasing it by one only if no length-d extension is found, we do not get to a large value of d in many steps. Nevertheless, searching for an extension can take time up to $4^{d_{\max}}$, which is tolerable for a small value of $d_{\max}$ only.

Apart from this exhaustive enumeration of extensions, we only need an efficient way to check whether a k-mer a, formed by an enumerated extension and prefix or suffix of the constructed superstring segment, is in the input set of k-mers K (more precisely, to work in the bi-directional model, we take the canonical of a and check its presence in the set of canonical k-mers K). This task can be easily accomplished in $\mathcal{O}(k)$ expected time using a hash table.

Hashing-based implementation of global greedy. We also provide an implementation of BiDir-GlobalGreedy which employs k-mer hashing. Since global greedy looks for the largest overlap in each step, we iterate over overlap lengths d from $k-1$ to 0. For each d, we create a hash table mapping each existing prefix of size d to a list of k-mers with this prefix which so far have indegree 0. Then, for each k-mer a with outdegree 0, we find the first k-mer with length-d prefix equal to the length-d suffix of a such that:

1. The corresponding directed edge does not form a cycle, which we check similarly as in [Ukk90]. In particular, as H is a collection of paths during the computation, for each path P in H we maintain pointers between the endpoints of P (these are arrays `first` and `last` in the pseudocode).
2. The edge does not go to a k-mer with indegree 1 (although we filter out nodes of non-zero indegree when creating the hash table, the indegrees may have changed as we are adding edges between reverse complementary k-mers). Whenever this happens, we erase this k-mer from the prefix hash table.
3. The edge does not go from a string to its reverse complement.

We provide a pseudocode for this hashing-based implementation of BiDir-GlobalGreedy in Algorithm 2.

We argue that this runs in linear time. For any d, the construction of the prefix hash table runs in expected time $\mathcal{O}(n)$ provided that we can compute any prefix (or suffix) of a k-mer with bit operations in constant time; namely, our implementation using 128-bit integers thus requires $k < 64$. Therefore, the only

Algorithm 1: BiDir-LocalGreedy-Hashing – outputs a masked superstring for K. The case $d_{max} = 1$ corresponds to Alg. 1 in [BBK21] (ProphAsm).

input : Length of k-mers k (where $k \geq 2$), set of canonical k-mers K, maximal extension length d_{max} (where $d_{max} < k$)
output : A masked superstring for K

Function `BiDir-LocalGreedy`(K, k, d_{max})**:**
  Superstring $\leftarrow$ `''`; Mask $\leftarrow$ `''`;
  while $|K| > 0$ **do**
    $(K, \mathsf{S}, \mathsf{M}) \leftarrow$ `NextSuperstringSegment`(K, k, d_{max});
    Superstring $\leftarrow$ Superstring + S; Mask $\leftarrow$ Mask + M;
  return (Superstring, Mask);

Function `NextSuperstringSegment`(K, k, d_{max})**:** ; `// Construct a path by locally extending it from an arbitrary` k`-mer`

  S $\leftarrow K$.pop(); M $\leftarrow$ `'1'` ; `// Start with an arbitrary` k`-mer`
  $d_L \leftarrow 1$; $d_R \leftarrow 1$; `// Depth of left/right extension search`
  while $\min\{d_L, d_R\} \leq d_{max}$ **do**
    if $d_R \leq d_L$ **then**
      $ext_R \leftarrow$ `FindExtension`(S, K, d_R, `'R'`);
      if ext_R **then**
        S $\leftarrow$ S + ext_R; M $\leftarrow$ M + `'` $\underbrace{\texttt{0…0}}_{(d_R-1)\times}$ `1'` ; `// right extension of the string and mask`
        $K \leftarrow K - \{$`CanonicalKmer`(suff$_k$(S))$\}$;
        $d_R \leftarrow 1$; `// Reset right extension depth`
      else
        $d_R \leftarrow d_R + 1$; `// No right extension found, increase R-depth`
    else
      $ext_L \leftarrow$ `FindExtension`(S, K, d_L, `'L'`);
      if ext_L **then**
        S $\leftarrow ext_L$ + S; M $\leftarrow$ `'1` $\underbrace{\texttt{0…0}}_{(d_L-1)\times}$ `'` + M ; `// left extension of the string and mask`
        $K \leftarrow K - \{$`CanonicalKmer`(pref$_k$(S))$\}$;
        $d_L \leftarrow 1$;
      else
        $d_L \leftarrow d_L + 1$; `// No left extension found, increase L-depth`
  M $\leftarrow$ M + `'` $\underbrace{\texttt{0…0}}_{(k-1)\times}$ `'` ; `// Make` M `of the same length as` S
  return (K, S, M);

Function `FindExtension`(S, K, d, LR)**:** ; `// Brute-force search for a length-`d `extension; LR specifies whether to extend left or right`

  foreach $ext \in \{$`'A'`, `'C'`, `'G'`, `'T'`$\}^d$ **do**
    switch LR **do**
      case `'L'` **do** $a = ext +$ pref$_{k-d}$(S) ;
      case `'R'` **do** $a =$ suff$_{k-d}$(S) $+ ext$;
    if `CanonicalKmer`(a) $\in K$ **then return** ext ; `// takes` $O(k)$ `expected time if` K `is represented using a hash table`
  return `''`;

Algorithm 2: BiDir-GlobalGreedy-Hashing – a hashing-based implementation of the global greedy algorithm that computes a superstring representation of a set of k-mers K in the bi-directional model.

input : Length of k-mers k (where $k \geq 2$), set of k-mers K (containing a reverse complement for each k-mer in K)
output : A masked superstring for K

Function BiDir-GlobalGreedy-Hashing(K, k):

```
H ← ∅ ;                                   // edges of the contructed Hamiltonian path
for i = 1, ... , |K| do
    first[i], last[i] ← i ;   // First/last vertex of path ending/starting at i, used for determining whether an edge closes a cycle
    prefixForbidden[i], suffixForbidden[i] ← False ;                    // is False iff indegree/outdegree=0
for d = k − 1, ..., 0 do          // Overlap length from the largest to the smallest
    P ←map from prefixes of size d to all k-mers with this prefix and prefixForbidden[i] = False;
    for j = 1, ..., |K| do                                  // iterate all the k-mers
        if suffixForbidden[j] = False then    // the k-mer has outdegree 0 so far
            s ← length-d suffix of k-mer j;
            i ← index of the first k-mer in P[s];
            while prefixForbidden[i] ∨ first[j] = i ∨ i = RC(j) do  // skip k-mer i if edge (j, i) would form a cycle or is between reverse complementary k-mers (RC computes the index of the reverse complement of a given k-mer)
                if prefixForbidden[i] then
                    Remove i from P[s]
                if P[s] does not contain more k-mers then Set i ← −1 and break the while loop;
                i ← next k-mer in P[s]
            if i ≠ −1 then // check if such k-mer exists
                Add edges (j, i) and (RC(i), RC(j)) to H;
                suffixForbidden[j], suffixForbidden[RC(i)] ← True ;   // outdegree of j and RC(i) is 1
                prefixForbidden[i], prefixForbidden[RC(j)] ← True ;   // indegree of j and RC(i) is 1
                // Fix first and last pointers;
                first[last[i]] ← first[j];
                last[first[j]] ← last[i];
                first[last[RC(j)]] ← first[RC(i)];
                last[first[RC(i)]] ← last[RC(j)];
                Remove i from P[s];
H′ ← one of the paths of length |K|/2 in H ;        // for each k-mer a, H′ contains a or RC(a)
S, M ← convert H′ to the masked superstring (by merging k-mers along H′) ;
 // M has the same length as S
return (S, M);
```

potentially expensive part is finding the first k-mer in the prefix table that fulfills the conditions above.

Note, however, that this does not increase the time complexity. For a fixed d, the second case can happen in total at most n times as we can remove each k-mer only once, and similarly, the first case can happen at most once per k-mer [Ukk90]. The same holds for the third case as for each string there is only one complementary string.

Therefore, the algorithm runs in expected linear time with respect to the total length of the k-mers, where the expectation is over the randomness used in the hash table. (More precisely, for a given k such that we can compute any prefix or suffix of a k-mer in constant time and store it in a variable, the expected running time is $\mathcal{O}(k \cdot n)$. Without the assumption on fast prefix/suffix retrieval, the time complexity would increase to $\mathcal{O}(k^2 n)$.) We can achieve $\mathcal{O}(nk)$ running time in the worst case if we use the implementation using the Aho-Corasick automaton which can also work with $k \geq 64$, at the cost of a certain overhead of using a prefix-tree-based data structure as we show in the next subsection.

3.1.5 Aho-Corasick-Based Implementations

We develop worst-case linear-time implementations for both local and global greedy algorithms using the Aho-Corasick (AC) automaton [AC75] (it is also deterministic as we avoid using hash functions).

Aho-Corasick-automaton-based implementation of global greedy. We describe a variant BiDir-GlobalGreedy using Aho-Corasick automaton as its core structure. A linear-time implementation of Greedy for SSP using the AC automaton was already designed by Ukkonen [Ukk90], and we extend it to representing k-mers in the bi-directional model.

We first describe how to use the automaton to implement global greedy for k-mer set representation in the uni-directional model; this is essentially the same implementation as in [Ukk90]. We traverse the automaton in the reverse breadth first search (BFS) order from the root (i.e., starting in leaves) with the aim to construct a Hamiltonian path H in the overlap graph. During this traversal, for every state s in the automaton, we maintain a list $L(s)$ of k-mers which have s as a prefix, and another list $P(s)$ for k-mers which have s as a suffix. For a leaf s, we have $L(s) = P(s) = \{s\}$, while for an internal node s, creating list $L(s)$ upon visiting s is done by just merging lists $L(s')$ over all children s' of s. List $P(s)$ for an internal node s is obtained by merging lists $P(s'')$ for states s'' such that $f(s'') = s$ (i.e., the failure function from s'' leads to s); note that such s'' is visited before s in the traversal and thus, we add strings in $P(s'')$ into $P(s)$ when processing s'', which implies that $P(s)$ is complete when processing s. The order of traversal guarantees that the pairs of k-mers with the highest overlap are merged first.

When we visit a state s, we use lists $L(s)$ and $P(s)$ to find a pair (a, b) of different k-mers such that:

- a has s as its prefix and b has s as its suffix,
- edge (a, b) does not close a cycle in H, and
- a has outdegree 0 in H and b has indegree 0 in H.

Namely, for each $a \in L(s)$, if a has outdegree 0 in H, we iterate $b \in P(s)$ and check the conditions above (that is, while visiting a state we may add more edges, at most one for each $a \in L(s)$). Note that the first condition is ensured by taking $a \in L(s)$ and $b \in P(s)$. We check the second condition in a similar way as in the hashing-based implementation. The second condition may not be satisfied only once for each k-mer a and each of $k-1$ states that contain it in $L(s)$; thus, we reject an edge due to the second condition at most $\mathcal{O}(k \cdot n)$ times in total. For the last condition, we check whether b has indegree 0 in H and if not, we remove b from $P(s)$ (recall that we only consider $a \in L(s)$ with outdegree 0 in H). This ensures that the implementation runs in linear time; see [Ukk90] for details. Finally, having a complete Hamiltonian path H, we merge k-mers in the order given by H. This concludes the description for the uni-directional model.

In the bi-directional model, we use a similar modification as for the hashing-based implementation: When we add an edge (a, b) to H, we also add (b', a') to H, where a' and b' are reverse complements (RCs) of a and b, respectively. Further, we forbid using edges that lead between a k-mer and its RC. This way, we eventually construct two disjoint paths of the same length, each satisfying the adjusted Hamiltonian property. These paths correspond to two reverse complementary superstrings in the bi-directional model.

Aho-Corasick-automaton-based implementation of local greedy. We provide an implementation of BiDir-LocalGreedy using the AC automaton where we use the automaton to decrease the time complexity of finding the left/right extension in the local greedy algorithm. In the version with k-mer hashing, this has exponential-time complexity with respect to the current value of d_L or d_R, which may be up to d_{max} in the worst case. (Recall that d_{max} is a parameter of local greedy, which specifies that the overlap in any step is at least $k - d_{max}$.)

As in global greedy, for each state s of the automaton, we maintain the lists $L(s)$ and $P(s)$ of k-mers which have the string corresponding to s as a prefix or a suffix, respectively. Furthermore, for each prefix and suffix of each k-mer we store the state of the automaton corresponding to the prefix/suffix (the corresponding state may not exist for the suffix, though it always exists for the prefix).

Suppose we are looking for a left extension of length d and let s be the length-$(k-d)$ prefix of the currently constructed string (denoted S in function `NextSuperstringSegment` in Algorithm 1). Since s is a length-$(k-d)$ prefix of some k-mer a, state s is in the automaton and we iterate $P(s)$ to find a k-mer not equal to a or its RC which has not been used as an extension or a starting k-mer yet. To ensure that this runs in linear time, when we find a k-mer b in $P(s)$ that has been used already, we remove b from $P(s)$.

We find a right extension analogously, with s being the length-$(k-d)$ suffix of the currently constructed string S. If s is not a state of the automaton, there is no length-d extension, and otherwise, we iterate list $L(s)$ similarly as we loop over $P(s)$.

We argue that this implementation runs in time $\mathcal{O}(k \cdot n)$ (that is, linear in the total length of the k-mers). The automaton and the mapping from prefixes and suffixes of k-mers to automaton states can be constructed in linear time. Using this mapping, searching for a left or right extension of length d can be

implemented in $\mathcal{O}(1)$ time if we do not count removals from lists $L(s)$ and $P(s)$. Since every k-mer appears in $k-1$ lists $L(s)$ and $k-1$ lists $P(s)$, there are in total at most $\mathcal{O}(k \cdot n)$ removals from these lists (each taking $\mathcal{O}(1)$ time). This concludes the time complexity analysis.

3.2 The Second Step: Mask Optimization

In the second step of masked superstring optimization, the mask is optimized. That is, given a set of k-mers K and a superstring S of K, we want to find a compatible mask M that is optimal under a mask optimization objective. The objective for mask optimization can be either informally derived from the objective for masked superstrings in case the masked superstrings are computed using the two-step optimization protocol, but we might also want to re-optimize masks for already computed masked superstrings.

We propose three objectives for mask optimization – minimizing or maximizing the number of `1`s and minimizing the number of runs of `1`s in the mask. We provide linear algorithms for minimizing and maximizing the number of ones. We prove that minimizing the number of runs of `1`s in NP-hard, but we propose an exact exponential solution based on integer linear programming that is efficient for bacterial genomes and pan-genomes.

We note that minimizing the number of runs of `1`s directly corresponds to minimizing space under compression of the mask with run-length encoding. In Section 6.3.2 we examine the performance of the mask objectives when evaluated on mask compression. Furthermore, in Section 5.3.2 we demonstrate that minimizing and maximizing `1`s can be used for mask recasting used in set operations with k-mer sets.

3.2.1 Maximizing the Number of `1`s

In maximization of the number of `1`s in the mask, for a given superstring S and a given set of k-mers K, we aim to find a mask M such that (S, M) represents K and the number of `1`s is maximal among such masks. We remark that such mask is always unique. Typically, we consider a setting where K is given implicitly via a masked superstring and we only want to optimize the mask.

We provide a very simple two-pass linear-time algorithm for this problem. In the first pass, we retrieve K and in the second pass, we always set $M[i] = \texttt{1}$ if at the position i a k-mer from K starts. The pseudocode is provided in Algorithm 3.

3.2.2 Minimizing the Number of `1`s

Similarly as in maximizing the number of `1`s, given a masked superstring, we aim to find a new mask with minimum number of `1`s that preserves the represented set. Note that unlike in the case of maximizing the number of `1`s, the masks are not unique and different algorithms may produce different results.

We provide a single-pass linear-time algorithm for this problem. We iterate over the masked superstrings and whenever we find a new represented k-mer, we put a `1` to the mask, otherwise we put a `0`. This way only a single occurrence of a represented k-mer is ON. The pseudocode is provided in Algorithm 4. Alternatively,

one might use a similar algorithm as for maximizing the number of 1s and compute K in the first pass. This has a potential advantage that it produces the same result regardless of the original mask.

3.2.3 NP-Hardness of Minimizing the Number of Runs

We show that for a given superstring finding a mask that minimizes the number of runs of ones is NP-hard via reduction from Set Cover. As an overview, to prove this we first restate the problem of minimizing number of runs as covering an edge-centric de Bruijn graph with walks of k-mers. We then map the elements of a given Set Cover instance to selected edges from the de Bruijn graph while modifying the Set Cover instance such that the individual sets correspond to walks in the graph without changing the optima. In this setting, finding the minimum number of runs corresponds to finding the minimum number of walks, which corresponds to finding the minimum number of sets to cover the Set Cover universe.

More formally, we consider the following problem, called MaskMinNumRuns: Given a set of k-mers K (for an arbitrary $k \geq 2$), and their superstring S, find a mask for S (w.r.t. K) that has the minimum number of runs of ones. Recall that a binary string M of the same length as S is a *mask* for S (w.r.t. K) if the following holds:

- every k-mer $a \in K$ has at least one ON occurrence, and
- each k-mer $a \notin K$ has no ON occurrences.

We prove that this optimization problem is hard for a carefully constructed worst-case superstring.

Theorem 4. *MaskMinNumRuns is NP-complete in both the uni-directional and bi-directional models. Furthermore, the problem is NP-hard to $o(\log |K|)$-approximate, i.e., it is NP-hard to find a mask with $o(\log |K|)$ times the optimal number of runs of ones.*

Note that k must not be bounded; this follows from a similar reason as outlined in the discussion of superstring NP-hardness (Section 3.1.1). As the problem is clearly in NP (with mask being a certificate, whose validity can be verified in polynomial time), it is sufficient to show NP-hardness. Strictly speaking, we prove the NP-hardness for the decision version of MaskMinNumRuns, which asks to determine whether there is a mask with at most λ runs of ones, for a given λ.

The approximation hardness uses the following (tight) result about Set Cover:

Theorem 5 (Corollary 4 in [DS14])**.** *For every fixed $\epsilon > 0$, it is NP-hard to approximate Set Cover to within an $(1 - \epsilon) \cdot \ln N$ factor, where N is the size of the instance.*

Proof of Theorem 4. First, we restate the MaskMinNumRuns problem using graph theory. We can take the edge-centric de Bruijn graph of the superstring and color the edges with two colors – blue if it corresponds to a ghost k-mer, i.e., a length-k substring not in K, and red if it appears in K. We can now observe that the superstring corresponds to a walk W in the de Bruijn graph. We can reformulate our problem as selecting the smallest number of subwalks of W consisting only of red edges such that all red edges are covered by one of

Algorithm 3: MaskMaxOnes – two-pass algorithm for maximizing the number of ones in a mask

input : Masked superstring (S, M) implicitly representing a k-mer set K, $k \in \mathbb{N}$
output : A masked superstring for K with the the same superstring and maximum number of ones

Function BiDir-GlobalGreedy-Hashing(S, M, k):

```
K ← ∅ ;                                                  // k-mers already seen
M' = [0] × |M| ;                                                  // new mask
for i = 1, ... , |S| − k do                                      // construct K
    a ← S[i : i + k − 1];
    if M[i] = 1 then
        K ← K ∪ {a};
for i = 1, ... , |S| − k do                                  // construct new mask
    a ← S[i : i + k − 1];
    if a ∈ K then       // can be done also in the bi-directional model
        M'[i] ← 1;
M'[i] ← 1;
return (S, M');
```

Algorithm 4: MaskMinOnes – one-pass algorithm for minimizing the number of ones in a mask

input : Masked superstring (S, M) implicitly representing a k-mer set K, $k \in \mathbb{N}$
output : A masked superstring for K with the the same superstring and minimum number of ones

Function BiDir-GlobalGreedy-Hashing(S, M, k):

```
K ← ∅ ;                                                  // k-mers already seen
M' = [0] × |M| ;                                                  // new mask
for i = 1, ... , |S| − k do
    a ← S[i : i + k − 1];
    if M[i] = 1 then
        if a ∉ K then   // can be done also in the bi-directional model
            K ← K ∪ {a};
            M'[i] ← 1;
return (S, M');
```

the selected subwalks. Observe that whenever we add a red edge into a selected subwalk, we can select all succeeding and preceding red edges in W until we reach a blue edge in W, without having to add another subwalk. Therefore, we can split W by blue edges into maximal red subwalks and find the minimum number of these red subwalks we need to take in order to cover all red edges in the graph.

We now show a reduction from Set Cover. Recall that an instance of Set Cover consists of universe U and a set of m subsets $A_1, \ldots, A_m$ of U, and the goal is to select the minimum number of these subsets A_i that cover U, i.e., whose union equals U.

Given an instance of Set Cover, we choose $n \geq |U| + 1$ and take a complete edge-centric de Bruijn graph G with n vertices corresponding to all strings of length $k - 1 = \log_2 n$ over alphabet {A, C} (we use just A and C to prove it even in the bi-directional model). The k-mers of our instance (including ghost k-mers) will correspond to a subset of the $2n$ edges of G, with each edge representing the length-k merge of its two endpoints. Note that G always contains a Hamiltonian cycle which we denote by H We color the edges of H in dark red. Next, we choose an edge $e_B \notin H$ and color it in blue; this will be the only blue edge. We color all the remaining edges (not in $H \cup \{e_B\}$) in light red. As G contains $2n$ edges in total, out of which we colored n in dark red and one in blue, there are $n - 1 \geq |U|$ light-red edges. We map each element in U to a light-red edge and delete the unmapped light-red edges if any. This way, we get a bijection between light-red edges and U. Furthermore, we modify every set A_i by adding (new elements corresponding to) all the dark-red edges H to obtain new sets denoted A'_i. We also add the dark-red edges into U to get a modified universe U'. Observe that U' consists of exactly (elements corresponding to) all of the red edges and that the solutions for the Set Cover instance $(U, \{A_i\}_i)$ are in one-to-one correspondence to solutions for instance $(U', \{A'_i\}_i)$.

Next, we map each set A'_i to a walk W_i in the graph consisting of red edges only in a way that we can connect the walks W_i into one walk in G just by using the blue edge $e_B = (a, b)$, where a is the tail of e_B and b is its head. For every set A'_i we list all the light-red edges which were mapped to an element in this set. We construct a walk W_i in the following manner: We start W_i at vertex b and then follow the Hamiltonian cycle H up to the tail of the edge corresponding to the first element in A_i, and we append this edge to the path. The iteratively append edges corresponding to other elements of A_i to W_i, connected by a path of dark-red edges in H. Namely, suppose that W_i ends with the j-th element of A_i and we want to add the $(j + 1)$-st. The walk ends at some vertex v_j and the next edge starts at possibly different vertex u_{j+1}. We take the path P_j from v_j to u_{j+1} in the dark-red Hamiltonian cycle H. Then we append path P_j and the $(j + 1)$-st edge (corresponding to the $(j + 1)$-st element of A_i) to W_i, thus extending this walk by one element from the set. At the end, we append the whole dark-red Hamiltonian cycle H to W_i and then finally, a part of H which ends at vertex a, which is the tail of the blue edge e_B. This way we obtain a walk which contains precisely the red edges corresponding to elements in A'_i (the edges in H may be contained more than once). Note also that W_i is polynomially large with respect to the size of the Set Cover instance as it contains at most $(n + 1)|A_i| + 3n$ elements.

We now connect the walks W_i by the blue edge (in an arbitrary order) and get

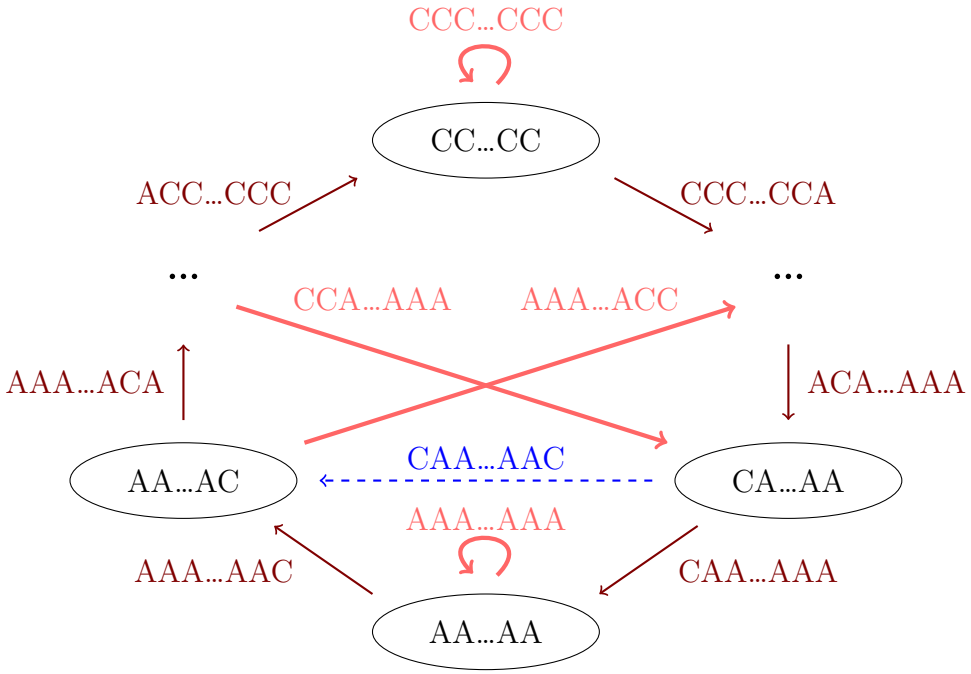


Figure 3.1: **Illustration of the reduction from Set Cover.** The elements from the Set Cover are mapped to the light-red (bold solid) edges. The dark-red (solid) edges which form a Hamiltonian path are added to all sets and can be used to transition between individual light-red edges, hence each set corresponds to a walk in this graph. The blue (dashed) edge is not from the universe and delimits the subwalk corresponding to individual sets.

a walk W such that every maximal red subwalk of W is W_i for some i. Therefore, the reduction preserves Set Cover solutions exactly, with the same objective value. In more detail, every solution for the Set Cover instance corresponds to a solution for MaskMinNumRuns on the instance from the reduction, and vice versa. Moreover, the number of selected subsets A_i equals the number of selected red subwalks, or equivalently, the number of runs of ones.

Finally, since all the k-mers of the superstring (including the only ghost k-mer corresponding to the blue edge) are over alphabet $\{\mathtt{A}, \mathtt{C}\}$ only, there are no reverse complements, so the reduction works in both the uni-directional and bi-directional models. This concludes the polynomial-time reduction from Set Cover to the graph formulation of MaskMinNumRuns, which implies that the NP-hardness of approximation by Theorem 5. □

The main downside of the proof is that the superstring resulting from the reduction would hardly be computed by any reasonable superstring algorithm, even on the same set of k-mers as in the reduction (moreover, a set of k-mers with a Hamiltonian cycle in its de Bruijn graph may not occur in practice). Still, the hardness proof justifies the usage of ILP solvers for MaskMinNumRuns outlined in the next subsection. We leave it as an open question whether or not for a k-mer superstring computed by a particular algorithm, e.g., local or global greedy, it is possible to solve MaskMinNumRuns in polynomial time.

3.2.4 ILP-Based Optimization of the Number of Runs

Despite the NP-hardness of minimizing the number of runs of ones in the mask, we propose an exact algorithm based on integer linear programming (ILP) that is efficient in practice for genomic data in bacterial pan-genomics. The efficiency is due to the fact that ILP solvers are well optimized and we further propose a greedy heuristic that significantly reduces the size of the ILP problem.

We first observe that if a ghost k-mer starts at a position i of the superstring, then in any mask representing the original set, $M[i]$ must be `0`. Therefore, ghost k-mers split the superstring into segments where the mask symbol can be arbitrary (given that every k-mer in K has at least one ON occurrence). Clearly, it gives no benefits to set different mask values for k-mers in the same segment as we can otherwise set all the mask values in the segment to `1` without increasing the number of runs. Therefore, we either set mask symbol for k-mers in a single segment all by `1`s or all by `0`s.

With those observations, we can formulate the problem as an ILP problem. For each segment $[l_j, r_j]$ of the superstring, we introduce a binary variable x_j that is 1 if the segment is masked by `1`s and 0 otherwise. For every k-mer $a \in K$, we add a constraint the sum of x_j's over all segments $[l_j, r_j]$ containing a (i.e. a has an occurrence whose first character is in this segment) is at least 1. The objective of out ILP is to minimize the sum of x_j's. We therefore obtain the following ILP problem::

$$\begin{aligned} \text{minimize} \quad & \sum_{j=1}^{m} x_j & & \\ \text{subject to} \quad & \sum_{j=1,\dots,m:\ a \in \mathbb{K}_j} x_j \geq 1 & & \forall \text{ canonical } k\text{-mer } a \in K \\ & x_j \in \{0,1\} & & \forall j = 1, \dots, m \end{aligned} \tag{MinRunILP}$$

The solution of the original problem can be obtained from the solution of the ILP problem by setting $M[i] = x_j$ for every $i \in [l_j, r_j]$ and `0`s to all occurrences of ghost k-mers. We further remark that this ILP is similar to the ILP for Set Cover. This is not a coincidence as we have seen that minimizing the number of runs yields solutions to Set Cover.

Furthermore, we provide a greedy heuristic to reduce the size of the ILP problem. It works in a two simple steps.

1. For each k-mer appearing only once in the superstring, we set $x_j = 1$ for the segment containing this k-mer.
2. Afterwards, if there is an undecided segment $[l_j, r_j]$ such that all k-mers in K that have an occurrence in this segment have already been masked by `1`s, we set $x_j = 0$.

We are left with segments that are not yet decided and we resolve them using the ILP by introduction variables only for these segments. This heuristic can be ran multiple times, however in practice the first run is usually sufficient to obtain a small-enough ILP instance.

In case the ILP instance is too large even after the reduction using the heuristic and we do not require the optimal solution, either the ILP part can be skipped completely with all remaining variables being assigned with 1s in which case the algorithm falls between maximizing the number of ones and minimizing

the number of runs of ones. Alternatively, an approximation algorithm can be used, e.g., the greedy algorithm for set cover which is known to be a logarithmic approximation [Joh74].

For a full pseudocode of the solution to minimizing the number of runs of ones in the mask, see Algorithm 5.

Algorithm 5: MaskMinRuns – ILP-based algorithm for minimizing the number of runs of ones in a mask

input : Masked superstring (S, M) implicitly representing a k-mer set K, $k \in \mathbb{N}$
output : A masked superstring for K with the the same superstring and the minimum number of runs of ones

```
Function MaskMinRuns(S, M, k):
    K ← dict ;                                  // intervals for represented k-mers
    M' = [0] × |M| ;                                                  // new mask
    for i = 1, ..., |S| − k do                                      // construct K
        a ← S[i : i + k − 1];
        if M[i] = 1 then
            K ← K ∪ {a};
    l ← 0 ;                                                    // current interval
    V ← False ;          // whether the current interval contains a k-mer
    for i = 1, ..., |S| − k do                                 // construct intervals
        a ← S[i : i + k − 1];
        if a ∈ K then       // can be done also in the bi-directional model
            K[a] ← K[a] ∪ {l};
            V ← True;
        else
            if V = True then
                l ← l + 1;
                V ← False;
    for a ∈ K do                                // first step of heuristic reduction
        if |K[a]| = 1 then
            x_{K[a][0]} ← 1;
    for j = 0, ..., l − 1 do                   // second step of heuristic reduction
        if i only has already represented k-mers then
            x_j ← 0;
    Solve ILP for remaining x_j's;
    Construct M' from x_j's;
    return (S, M');
```

3.3 Two-Step Protocol Approximation Guarantees

In the previous sections, we have described the two-step optimization protocol for computing masked superstrings, which solves a reformulated version of the problem instead of computing masked superstrings directly. In this section we show that this protocol in fact computes an approximately optimal solution for many different objectives including those to store masked superstrings in encodings from Figure 2.1.

Towards this, we put several assumptions on the objective function $g(S, M)$. We assume that g can be split into a linear function of the superstring length

and a non-negative function, that can be bounded by a function of mask length. Formally, we assume $g(S, M) = c|S| + g_m(S, M)$, where c is a constant and $0 \leq g_m(S, M) \leq |M| \cdot \hat{g}_m(|M|)$. In the upper bound, we move the factor of $|M|$ outside of $\hat{g}_m$ as this simplifies the math and also most functions naturally grow with increasing size of the mask. For function satisfying these assumptions, we provide approximation ratio guarantees for the two-step optimization protocol in Theorem 6.

Theorem 6. *Given an objective function $g(S, M) = c|S| + g_m(S, M)$, where $0 \leq g_m(S, M) \leq |M| \cdot \hat{g}_m(|M|)$, the two-step optimization protocol has an approximation ratio of*

$$\sigma\left(1 + \frac{\hat{g}_m(|S|)}{c}\right),$$

where σ is the approximation ratio of the algorithm used to compute a superstring in the first step.

Proof. Consider an optimal pair (S_{opt}, M_{opt}) and the value of $g(S_{opt}, M_{opt}) = c|S_{opt}| + g_m(S_{opt}, M_{opt})$, which can be lower-bounded by $c|S_{opt}|$ as g_m is non-negative. On the other hand, the value of the solution (S, M) obtained by the two step optimization protocol can be upper bounded as $c|S| + |M|\hat{g}_m(|M|)$ which is in fact $c|S| + |S|\hat{g}_m(|S|)$ as $|S| = |M|$. Hence, the approximation ratio is at most

$$\frac{c|S| + |S|\hat{g}_m(|S|)}{c|S_{opt}|} \leq \sigma\frac{c|S| + |S|\hat{g}_m(|S|)}{c|S|} \leq \sigma\left(1 + \frac{\hat{g}_m(|S|)}{c}\right)$$

which gives the desired ratio. Clearly, the second step cannot make the result worse. □

We remark that for global greedy in the uni-directional model, σ is about 3.396 [EMV23]. We do not give a proof for the same or similar ratio for the bi-directional global greedy, instead we note that the ratio in the bi-directional model cannot be worse than in the uni-directional case by a factor of more than 2, which is still in $\mathcal{O}(1)$.

We show that this general result applies to all the encodings of masked superstrings shown in Figure 2.1.

- `enc1` or `enc2`: If we store masked superstrings plainly as bit vectors, it requires three bits per superstring character, hence $g(S, M) = 3|S|$ and the ratio is σ.

- `enc3`: If we store the superstring in a vector and masks using its run-length encoding, the objective function is $g = 2|S| + \texttt{runs}(M) \cdot \log|S|$ where the second term can be upper bounded by $|M| \log|S|$, hence we get the ratio $\sigma(1 + \frac{1}{2}\log|S|)$ or at most $\sigma(1 + \frac{1}{2}\log kn)$, where n is the number of k-mers.

- `enc4`: If we store the superstring in a vector and store the positions of zeros in the mask, the objective function is $g = 2|S| + |M|_0 \log|S|$. We can bound the second term as in the previous case and obtain the ratio $\sigma(1 + \frac{1}{2}\log|S|)$ or $\sigma(1 + \frac{1}{2}\log kn)$.

We remark that it is possible to generalize the result for more complex functions of the masked superstring along the same lines, but for most practically interesting cases, this formulation is sufficient.

However, for functions that do not depend on the superstring length directly, this approach gives no approximation guarantees. In fact, as we show in Appendix A, in the case of finding a masked superstring with a globally minimal number of runs of ones, the ratio for the two-step method cannot be bounded. As an upside, in Appendix A, we sketch a polynomial-time algorithm that solves this objective optimally.

Despite the fact that this covers the majority of practically interesting objectives, we note that the logarithmic ratio obtained in the two latter cases is far from ideal. Yet worse, if we do not assume anything about the superstring algorithm except for its ratio, it can be easily seen that the result for each of the encodings is asymptotically tight as we can consider an optimal superstring with a single run of ones, but a computed superstring can have a linear number of runs (and similarly for the number of zeros). We leave it as an open question to design a method for simultaneous optimization of the superstring and mask that would yield a better result.

4 FMS-Index: Efficient Indexing of Masked Superstrings

As masked superstrings form a textual representation k-mer sets, it is natural to index them using full-text indexes (see, e.g., review in [Nav16]). The most natural choice is the FM-index [FM05] (for more details see Section 1.5.5), given its heavy use accross bioinformatics and the availability of high-quality implementations. However, due to the presence of the mask, applying the FM-index directly does not work.

We thus introduce FMS-Index, a modified version of the FM-index, which omits the memory demanding sampled suffix arrays and adds an auxiliary table for the mask, which as a result simplifies membership queries.

Recall that in the FM-index, when indexing several strings at one, the most computationally expensive operation in order to answer membership queries is the location of each of the occurrences in the original strings in order to determine whether the occurrence is inside a queried string or overlaps a boundary. This also requires significant memory overheads as we need to store both the sampled suffix array and the starting position of each segment.

We therefore propose the FMS-index which consists of the following components:

A) The BWT of the superstring.
B) The occurrence function.
C) The SA-transformed mask.

	FM-index [FM05]	**FMS-index (Chapter 4)**
Underlying representations	(r)SPSS [CLJ+14; BBK21; RM21; SKA+23]	Masked superstrings (Section 2)
Required structures	A) BWT of concatenated strings B) Occurrence function C) Starting positions of each string in their concatenation D) Sampled suffix array	A) BWT of the superstring B) Occurrence function C) SA-transformed mask
Membership query	1) Compute the occurrence range using backwards search 2) Locate each occurrence 3) Check if at least one occurrence is not on the boundary	1) Compute the occurrence range using backwards search 2) Check if there is at least one ON occurrence using rank queries

Table 4.1: **Comparison of the FMS-index to the FM-index.** Regarding the required structures, note that the BWTs (A) and occurrence functions (B) directly correspond and that the starting positions (C) can be viewed as a compressed form of an untransformed mask and hence it corresponds to the transformed mask (C). The sampled suffix array (D) is not required in the FMS-index. In the membership query, step (1) is the same in both cases, but for the FMS-index steps (2) and (3) are replaced by a much simpler step of two rank queries on the SA-transformed mask.

The BWT and the occurrence function are the same as in the FM-index. The only new component is the transformed mask, which we define as follows. For the comparison with the FM-index, see Table 4.1.

Definition 3 (SA-transformed Mask)**.** *Let S be a superstring and M be a mask with a special \$ symbol, which is less than all the alphabet letters, appended to*

each. Then the SA-transformed mask is a binary string M' of the same length, such that $M'[i] = x$ if and only if $M[j] = x$ where j is the starting position of the i-th lexicographically smallest suffix of S.

Another way how to view the SA-transformed mask is to consider the Burrows-Wheeler matrix of the superstring and attach each mask symbol to its corresponding superstring letter. The SA-transformed mask is formed by the mask symbols in the first column of the matrix. For a better understanding, see Figure 4.1.

Burrows-Wheeler matrix for masked superstring
(CAGGTAG$, 1011100$)

($, **$**)	(C, 1)	(A, 0)	(G, 1)	(G, 1)	(T, 1)	(A, 0)	(G, 0)
(A, **0**)	(G, 0)	($, $)	(C, 1)	(A, 0)	(G, 1)	(G, 1)	(T, 1)
(A, **0**)	(G, 1)	(G, 1)	(T, 1)	(A, 0)	(G, 0)	($, $)	(C, 1)
(C, **1**)	(A, 0)	(G, 1)	(G, 1)	(T, 1)	(A, 0)	(G, 0)	($, $)
(G, **0**)	($, $)	(C, 1)	(A, 0)	(G, 1)	(G, 1)	(T, 1)	(A, 0)
(G, **1**)	(G, 1)	(T, 1)	(A, 0)	(G, 0)	($, $)	(C, 1)	(A, 0)
(G, **1**)	(T, 1)	(A, 0)	(G, 0)	($, $)	(C, 1)	(A, 0)	(G, 1)
(T, **1**)	(A, 0)	(G, 0)	($, $)	(C, 1)	(A, 0)	(G, 1)	(G, 1)

Figure 4.1: **Illustration of the relation of the SA-transformed mask and the Burrows-Wheeler matrix** The Burrows-Wheeler matrix for an example superstring with the corresponding mask symbols attached to it. The Burrows-Wheeler transform are the superstring symbols in the last column, whereas the SA-transformed mask (Definition 3) are the mask symbols in the first column (in bold).

4.1 Basic Operations with the FMS-Index

In this section, we describe how to efficiently implement three basic operations: construction of the index from a masked superstring, exporting the masked superstring back from the index, and merging two indexes. In Section 4.2, we show how to answer membership queries with the FMS-index.

Construction. The construction of the FMS-index is relatively straightforward. The BWT and the occurrence function are computed as in the FM-index. A possible way this can be done is by first constructing the inverse of the suffix array, from which the BWT can be computed in linear time. This can also be used to directly compute the SA-transformed mask. Therefore, there are no additional overheads needed and the whole construction can be done in linear time.

Masked superstring export. We call the operation of retrieving the original masked superstring from its indexed representation as its export. Getting the

superstring itself is simple, as we only reverse the BWT as in the FM-index. To get the original mask, we realize that the i-th symbol of the SA-transformed mask corresponds to the mask symbol which succeeds in the original superstring the i-th letter of the BWT. Therefore, we reverse the SA-transformed mask by the same operation as the BWT and then rotate it by one position to the right.

Index merging. To merge the FMS-indexes, we can utilize the same approach as with the FM-indexes, we simply glue the i-th SA-transformed mask to the i-th letter of the BWT, merge it as if there was only the BWT and then unglue the mask symbols. The merging of the BWTs can be done as described in [HM14].

Compaction. If the masked superstring contains too many redundant copies of individual k-mers, which may happen if we merge multiple indexed masked superstrings, it may be desirable to *compact* it, i.e., reoptimize its support superstring to reduce memory requirements. This can be performed in linear time, using two different approaches: One option is exporting the f-masked superstring, counting the number of ON and OFF occurrences of each k-mer, constructing the represented k-mer set, and then compute the masked superstring (Chapter 3). Alternatively, one may directly compute an f-masked superstring using the local greedy algorithm (Section 3.1.2) executed directly on the FMS-index, see Appendix C.

4.2 Membership Queries via Simplified FM-Index Search

The process of answering membership queries on indexed masked superstrings can be split into two steps (see also Table 4.1 for comparison to the FM-index search). The first step is the same as in the FM-index. We perform the backwards search for the queried k-mer and obtain the range of occurrences which correspond to prefixes of size k of consecutive rows in the Burrows-Wheeler matrix.

This directly gives us occurrences of the queried k-mer and if at least one of them is ON, the queried k-mer is represented. We can get the number of ON occurrences in this range (which also gives us the number of OFF occurrences) using two rank queries on the SA-transformed mask. If we work in the bi-directional model, we also query the reverse complement of the queried k-mer and sum the numbers of ON occurrences. If the sum is non-zero, the k-mer is represented.

The whole query can be answered in $\mathcal{O}(k)$ time since the most expensive operation is the backwards search. Note that compared to the FM-index, we do not need to perform the costly operation of locating each occurrence in the original string coordinates.

5 Function-Assigned Masked Superstrings for Set Operations

5.1 Function-Assigned Masked Superstrings

Suppose we are given a masked superstring (M, S) and our objective is to determine whether a given k-mer Q is among the represented k-mers. Conceptually, this process consists of two steps: first, identify the occurrences of Q in S and retrieve the corresponding mask symbols; then, verify whether at least one `1` is present. We can formalize this process via a so-called *occurrence function*.

Definition 4. *For a superstring S, a mask M, and a k-mer Q, the* occurrence function $\lambda(S, M, Q) \to \{0,1\}^*$ *is a function returning a finite binary sequence with the mask symbols of the corresponding occurrences, i.e.,*

$$\lambda(S, M, Q) \coloneqq (M_i \,|\, S_i \cdots S_{i+k-1} = Q) \ .$$

In this notation, verifying k-mer presence corresponds to evaluating the composite function '$\mathbf{or} \circ \lambda$'; i.e., k-mer is present if $\lambda(S, M, Q)$ is non-empty and **or** of the values is 1. For instance, in Example 2 for the k-mer $Q = \texttt{GGG}$, it holds that $\lambda(S, M, Q) = (0, 1)$, as the first occurrence is OFF and the second ON, and the **or** of these values is 1; therefore, is `GGG` is represented. The set of all represented k-mers for a masked superstring (S, M) is then

$$K = \{Q \in \{\texttt{A}, \texttt{C}, \texttt{G}, \texttt{T}\}^k \,|\, f(\lambda(S, M, Q)) = 1\} \,,$$

where f is the **or** function.

Nevertheless, **or** is not the only function f that is applicable for such a "demasking" as well; for instance, with **xor**, we consider a k-mer present if and only if there is an odd number of ON occurrences of Q (Figure 5.2).

In fact, k-mer demasking can be done with any Boolean function; see an overview in Table 5.1. Furthermore, it is convenient to allow the function to reject some input sequences as invalid by returning a special value called `invalid`, which can also be viewed as restricting mask domain and thus enforcing certain criteria on mask validity. Finally, we limit ourselves to symmetric functions only, as these will later provide useful guarantees for indexing.

Definition 5. *We call a symmetric function $f : \{0,1\}^* \to \{0, 1, \texttt{invalid}\}$ a* k-mer demasking function.

However, not all demasking functions are practically useful, and we will typically require them to have several natural properties. First, we require the non-appearing k-mers to be treated as non-represented, which is ensured by property (P1) in Definition 6 below. (Naturally, if we want to represent the complement of a set, we treat the non-appearing k-mers as represented.) Second, property (P2) guarantees that for any appearing k-mer Q with any number of occurrences in a given superstring, we can set the mask bits in order to both make Q represented and not represented. Third, even with (P1) and (P2), there is an

Function name	**Definition**	**Comprehensive**	**Use cases**
or	0 if $\|\lambda\|_1 = 0$ 1 if $\|\lambda\|_1 > 0$ `invalid` never	yes	• The default f-masked superstring (Section 5.1) • Generalizes (r)SPSS representations (Section 5.2.2) • Union input and output function (Section 5.2.2)
xor	0 if $\|\lambda\|_1$ is even 1 if $\|\lambda\|_1$ is odd `invalid` never	yes	• Symmetric difference input and output function (Section 5.2.3)
and	0 if $\lambda = \epsilon$ or $\|\lambda\|_0 > 0$ 1 if $\lambda \neq \epsilon$ and $\|\lambda\|_0 = 0$ `invalid` never	yes	• Allows for ON occurrences of ghost k-mers **or**-MS (Appendix B)
$[a,b]$-**threshold** $(1 \leq a \leq b)$	0 if $\|\lambda\|_1 < a$ or $\|\lambda\|_1 > b$ 1 if $a \leq \|\lambda\|_1 \leq b$ `invalid` never	iff $a = 1$	• Intersection and set difference output function (Section 5.2.4)
one-or-nothing	0 if $\|\lambda\|_1 = 0$ 1 if $\|\lambda\|_1 = 1$ `invalid` otherwise	yes	• Union, symmetric difference and intersection input function (Section 5.2.4) • Set difference left input function (Section 5.2.5)
two-or-nothing	0 if $\|\lambda\|_1 = 0$ 1 if $\|\lambda\|_1 = 2$ `invalid` otherwise	no	• Set difference right input function (Section 5.2.5)
all-or-nothing	0 if $\|\lambda\|_1 = 0$ 1 if $\lambda \neq \epsilon$ and $\|\lambda\|_0 = 0$ `invalid` otherwise	yes	• No need for mask rank in queries (Appendix B)

Table 5.1: **Overview of selected demasking functions f for f-masked superstrings.** The table includes those functions that are use for set operations or used in other contexts throughout the paper. In the definitions, we abbreviate $\lambda(f, S, M)$ as λ. Note also that even non-comprehensive functions in this table satisfy properties (P1) and (P4) from Definition 6.

ambiguity in the meaning of `0` and `1` in the mask and thus, in (P3), we require the `1` to have the meaning of a k-mer being represented; namely, if it has a single occurrence masked with `1`, it should be treated as represented. Finally, in (P4), we require the function to be efficiently computable, specifically in $\mathcal{O}(1)$ time from the frequencies of `0`s and `1`s in its input. We call demasking functions satisfying these properties *comprehensive*.

Definition 6. *We say that a demasking function f is* comprehensive *if it satisfies the following three properties:*

(P1) $f(\epsilon) = 0$.

(P2) For every $n > 0$, there exist $x, y \in \{0,1\}^n$ such that $f(x) = 0$ and $f(y) = 1$.

(P3) $f((1)) = 1$ *and* $f((0)) = 0$.

(P4) Given $|x|_0$ and $|x|_1$, one can evaluate $f(x)$ in constant time in the wordRAM model.

With the notion of demasking functions f in hand, we generalize the concept of masked superstrings to so-called f-masked superstrings.

Definition 7. *Given a demasking function f, a superstring S, and a binary mask M with $|M| = |S|$, we call a triplet $\mathcal{S} = (f, S, M)$ a* function-assigned masked superstring *or* f-masked superstrings*, abbreviated as* f-MS*.*

Since we allow the output of f to be `invalid`, it may happen that for some f-masked superstring (f, S, M) and some k-mer Q, the result of f for the occurrences of Q turns out to be `invalid`. We call such f-masked superstring *invalid* and will always ensure validity for all masked superstrings that we will work with.

For a valid f-masked superstring, the set of represented k-mers is

$$K = \{Q \in \{\mathtt{A}, \mathtt{C}, \mathtt{G}, \mathtt{T}\}^k \mid f(\lambda(S, M, Q)) = 1\}.$$

The following observation is a consequence of property (P2) in Definition 6 of comprehensive demasking functions.

Observation 7. *For a comprehensive demasking functions f, it holds that for any k-mer set K and any superstring S of k-mers in K, there exists a mask M such that (f, S, M) represents exactly K.*

This further means that for any f-MS (f, S, M) and any comprehensive demasking function g, it is possible to find a mask M' such that (g, S, M') represents exactly the same set as (f, S, M).

Example. Consider Example 2 with the set of 3-mers $K = \{\mathtt{ACG}, \mathtt{GGG}\}$, a superstring $S = \mathtt{ACGGGG}$, and a mask $M = \mathtt{101100}$. Then the occurrence function for $Q = \mathtt{GGG}$ is $\lambda(S, M, Q) = (1, 1)$. If we choose the f to be **or**, then $f(\lambda(S, M, Q)) = 1$ and thus, `GGG` is represented. However, if we choose **xor** instead, `GGG` is considered as a ghost k-mer.

For **or**, the represented set is $\{\mathtt{ACG}, \mathtt{GGG}\}$ and since **xor** is a comprehensive function, there always exists a mask M' such that $(\mathbf{xor}, S, M')$ represents the same set as $(\mathbf{or}, S, M)$. In our case M' could be `100100`. For functions which are not comprehensive, this is in general impossible. For instance, considering the (impractical) constant-zero function, the represented set will always be empty.

5.2 Function-Assigned Masked Superstrings as an Algebraic Framework

In this section, we describe on the conceptual level how to perform set operations on k-mer sets by simply concatenating f-masked superstrings and choosing suitable demasking functions f. We deal with practical aspects of efficient implementation of this concept in Section 5.3.

5.2.1 Concatenation as an Elementary Low-Level Operation

We define concatenation on f-masked superstrings as concatenating the underlying superstrings and masks for all possible input and output functions f.

Definition 8. *Given a function-assigned masked superstring* (f_1, S_1, M_1) *and* (f_2, S_2, M_2)*, we define* (f_1, f_2, f_o)*-concatenation as the operation taking these two function-assigned masked superstrings and producing the result* $(f_o, S_1 + S_2, M_1 + M_2)$*. We denote this operation by* $+_{f_1,f_2,f_o}$*.*

Note that Definition 8 can be easily extended to more than two input f-masked superstrings. In the case that all the functions are the same, i.e. $f = f_1 = f_2 = f_o$, we call it *f-concatenation* or just *concatenation* if f is obvious from the context.

Definition 9. *We call the set operations that can be performed with* $f_1 = f_2 = f_o$ function-preserving set operations*. The operations that cannot be performed with a single function are called* function-transforming set operations.

Furthermore, note that while the set of appearing k-mers of $S_1 + S_2$ clearly contains the union of appearing k-mers of S_1 and of S_2, additional new occurrences of k-mers may appear at the boundary of the two superstrings. These newly appearing k-mers may not be appearing in any of the superstrings S_1 and S_2. We refer to them as *boundary k-mers* and to the occurrences of appearing k-mers of $S_1 + S_2$ that overlap both input superstrings as *boundary occurrences*.

In the following sections, we demonstrate how concatenation can be used to perform set operations. In Sections 5.2.2 and 5.2.3, we demonstrate that union and symmetric difference are function-preserving set operations and in Sections 5.2.4 and 5.2.5 we show intersection and set difference as function-transforming set operations. See Figure 5.2 for an example of function-preserving set operations.

	k-mer set 1	k-mer set 2	set from concatenation
Example f-MSes	$\mathcal{S}_1 =$ AGc	$\mathcal{S}_2 =$ CgGCg	$\mathcal{S}_1 + \mathcal{S}_2 =$ AGcCgGCg
On occurrences	$1 \times$ AG, $1 \times$ GC	$1 \times$ GC, $2 \times$ CG	$1 \times$ AG, $2 \times$ GC, $2 \times$ CG
Off occurrences	none	$1 \times$ GG	$1 \times$ CC, $1 \times$ GG
f interpreted as **or**			
represented set K	{AC, GC}	{CG, GC}	{AC, CG, GC}
ghost set X	{}	{GG}	{CC, GG}
f interpreted as **xor**			
represented set K	{AC, GC}	{GC}	{AC}
ghost set X	{}	{CG, GG}	{CC, GC, CG, GG}

Table 5.2: **Represented 2-mer sets with or and xor for masked superstrings and their concatenation.** The second and third row depict the On and Off occurrences of 2-mers, respectively, in the uni-directional model for masked superstrings $\mathcal{S}_1 =$ AGc, $\mathcal{S}_2 =$ CgGCg and their concatenation. Note that after the concatenation new Off occurrences of boundary k-mers emerge; in this case, it is just the blue-colored 2-mer CC. The bottom part depicts the represented sets K and ghost sets X for these f-masked superstring when interpreted using **or** and **xor**. Note that $K_{\mathbf{or}} = K_1 \cup K_2$ in the case of **or** and $K_{\mathbf{xor}} = K_1 \Delta K_2$ for **xor**.

5.2.2 Union

We show that concatenating masked superstrings, which are **or**-masked superstrings in our notation, naturally act as union on the represented sets. Hence, union is a function-preserving set operation.

Theorem 8. *For any two sets of k-mers K and K' and any of their **or**-masked superstrings (S, M) and (S', M') respectively, the concatenation of the **or**-masked superstrings is a valid **or**-masked superstring representing the set $K \cup K'$.*

Proof of Theorem 8. Consider a k-mer appearing in K. Then it must have at least one ON occurrence in (S, M) and thus, it will have at least one ON occurrence in the concatenation and will be considered represented. The same line of reasoning works for k-mers in K'. Conversely, consider a k-mer neither in K nor in K'. It has no ON occurrence in any of the original masked superstrings and hence it has no ON occurrence in the result and is correctly not considered as represented. This proves that in the result, exactly the k-mers from $K \cup K'$ are represented. □

This property allows **or**-masked superstrings to generalize (r)SPSS representations, since any set of k-mers in the (r)SPSS representation can be directly viewed as an **or**-masked superstring by concatenating the individual simplitigs/matchtigs.

In fact, we show that **or** is the *only* comprehensive demasking function that acts as union on the represented sets.

Theorem 9. ***or** is the only comprehensive demasking function f such that for any two k-mer sets K and K' and any of their valid f-masked superstrings (S, M) and (S', M') respectively, the concatenation of the f-masked superstrings is a valid f-masked superstring representing the sets $K \cup K'$.*

Proof of Theorem 9. We prove this via contradiction. Assume there is a function f different than **or** that satisfies the above. Consider the smallest n such that there exists an input x of length n such that $f(x)$ is not **or** of x. As f is comprehensive and thus $f((1)) = 1$, $n > 1$. Since f is comprehensive and different than **or**, there must be $x \neq 0$ of length n s.t. $f(x) \neq 1$. Fix a k-mer, for simplicity $\mathtt{A}^k$ (although similar approach works for all k-mers). We take the first f-masked superstring to be the k-mer with mask being $M_0 = x_0$ and $M_i = 0$ for the remaining $k-1$ positions. And the second f-masked superstring to be $\mathtt{CA} \ldots \mathtt{A}$ where $\mathtt{A}$ appears $n + k - 2$ times with the mask being: $M_0 = 0$ and other $M_i = x_i$. At least one of the represented sets contains the k-mer as $x \neq 0$ but the resulting f-masked superstring is either invalid or does not contain the k-mer in the represented set as $f(x) \neq 1$, a contradiction. □

We further demonstrate this uniqueness even on the level of matchtigs and therefore, **or**-masked superstrings are the only f-masked superstrings that generalize (r)SPSS representations.

Theorem 10. ***or** is the only comprehensive demasking function f such that for any sequence of f-masked superstrings, where individual superstrings are matchtigs, the concatenation of all the f-masked superstrings represents the union of represented k-mers.*

Proof of Theorem 10. It is sufficient to find a construction of matchtigs such that we can construct an arbitrary sequence of ones and zeros at the occurrences of a given k-mer and rest follows from the proof of Theorem 9.

We do this with k-mer `CG` and matchtigs `Cg` and `Gc`. Consider the counterexample sequence of occurring ones and zeros from Theorem 9. For every 1 in the sequence, we add the matchtig `Cg` and for each m consecutive zeros, we add $m+1$ times the matchtig `Gc` since at the boundary of two `Gc` matchtigs an OFF occurrence of k-mer `CG` appears. At any other boundaries, the k-mer does not appear, therefore the construction is correct. The rest of the proof follows a similar argument as in Theorem 9. □

Note, however, that the same does not hold if we want to represent simplitigs/SPSS solely. As an individual simplitig cannot appear more than once with an ON occurrence, any comprehensive function generalizes SPSS representations if it satisfies that if there is one ON occurrence of a k-mer, it returns 1, and if there is none, it returns 0.

5.2.3 Symmetric Difference

Next, we observe that **xor** naturally acts as the symmetric difference set operation, i.e., concatenating two **xor**-masked superstring results in a **xor**-MS representing the symmetric difference of the original sets. Indeed, recall that using **xor** implies that a k-mer is represented if and only if there is a odd number of ON occurrences of that k-mer. Observe that the boundary occurrences of k-mers do not affect the resulting represented set as those have zeros in the mask. Thus, if a k-mer is present in both sets, it has an even number of ON occurrences in total and hence, is not represented in the result. Likewise, if a k-mer belongs to exactly one input set, it has an odd number of ON occurrences in this input set and an even number (possibly zero) in the other; thus, it is represented in the result. As any appearing k-mer is either boundary or appears in one of the masked superstrings, the result corresponds to the symmetric difference.

5.2.4 Intersection

After seeing functions for union and symmetric difference operations, it might seem natural that there should be a function for intersection. This is however not the case as there is no comprehensive demasking function that acts as the intersection when concatenating f-masked superstrings.

In a nutshell, this impossibility is caused by the fact that if there is a k-mer Q that occurs exactly once in the input masked superstrings with `1` in the mask, then after concatenation, it will still occur once with `1` in the mask, so under any comprehensive f the k-mer would appear as if it was in the intersection.

Theorem 11. *There is no comprehensive demasking function f with the property that the result of f-concatenation of two f-masked superstrings always represents the intersection of the originally represented k-mer sets.*

Proof of Theorem 11. Let f be any comprehensive demasking function. Consider masked superstrings `A` and `C`, each representing a single 1-mer. Their concatenation is `AC`. Since $f((1)) = 1$ by the comprehensiveness of f, the concatenation represents

both 1-mer `A` and `C`. However, the intersection is empty and thus, f cannot be used to compute the intersection from the concatenation. □

Note that the proof cannot be generally extended to all demasking function as there exist non-comprehensive demasking functions acting as the intersection on the represented sets upon concatenation, for instance the constant zero function. However, since the constant zero function always represents the empty set, it is of no use in practice.

We further remark that although we have for convenience used the property (P3) from the definition of comprehensive functions, the proof in fact relies only on the property (P2) and holds even if consider not only 1-mers but the same counterexample can be built for any k.

We can circumvent the non-existence of a single demasking function acting as intersection by using possibly non-comprehensive demasking functions that are different for the result than for the input. We further show that such schemes have other applications beyond intersection.

To this end, we will need two different types of demasking functions:

- $[a,b]$-**threshold** function (where $0 < a \leq b$) is a demasking function that returns 1 whenever it receives an input of at least a ones and at most b ones and 0 otherwise. Note that unless $a = 1$, $[a,b]$-**threshold** functions are not comprehensive as they do not satisfy properties (P2) and (P3). The corresponding f-masked superstrings are denoted $[a,b]$-**threshold**-masked superstrings.

- The **one-or-nothing** function is a demasking function that returns 1 if there is exactly one 1 in the input, 0 if there are no 1s, and `invalid` if there is more than a single ON occurrence of the k-mer. Note that this function is comprehensive.

We now use these functions to perform any symmetric set operation on any number of input k-mer sets. Given N sets of k-mers, we compute a **one-or-nothing**-masked superstring for each. This is always possible since **one-or-nothing** is a comprehensive demasking function and can be done by directly using the superstrings and masks computed by local or global algorithms (Section 3.1)

We then concatenate the individual **one-or-nothing**-masked superstring. The result is not a valid **one-or-nothing**-masked superstring in general, but it has the special property that each k-mer has as many ON occurrences as the number of sets in which it appears. We can therefore change the demasking function of the resulting f-masked superstring from **one-or-nothing** to an $[a,b]$-**threshold** function. This will result in an $[a,b]$-**threshold**-masked superstring that is always valid and the represented set will be exactly the k-mers that appear in at least a sets and at most b sets. Important $[a,b]$-**threshold**-masked superstrings in this setting include the following:

- The $[N,N]$-**threshold**-masked superstring corresponds to taking the intersection of the represented sets.

- The $[1,N]$-**threshold**-masked superstring is the **or**-masked superstring and corresponds to taking the union.

- The [1,1]-**threshold**-masked superstring corresponds to taking those k-mers that appear in exactly one of the original sets. In case of $N = 2$, this corresponds to the symmetric difference.

It is important to emphasize that we can use different [a,b]-**threshold** functions to alter the resulting k-mer set without changing the superstring or the mask. For instance, we can use the same superstring and mask to consider intersection and union simply by changing the function from [N,N]-**threshold** to [1,N]-**threshold**.

Arbitrary symmetric set operations. The same scheme, with more general demasking functions, can be used to implement *any symmetric set operation* ***op*** *on any number of sets*. Indeed, given N, we again concatenate their **one-or-nothing**-masked superstrings in an arbitrary order. The symmetry of **op** implies that there is a set $S_N \subseteq \{0, 1, ..., N\}$ such that a k-mer belongs to the set resulting from applying **op** if and only if it is in a input sets for some $a \in S$. The sets S_N for $N = 1, 2, ...$ can be directly transferred into a demasking function $f_{\mathbf{op}}$ that models **op**; however, $f_{\mathbf{op}}$ may not satisfy the property (P4) from Definition 6.

5.2.5 Set Difference

Having seen how to perform symmetric set operations, we deal with asymmetric ones, focusing on the set difference of k-mer sets A B. Clearly, we cannot use the same demasking function f to represent both A and B as it would be impossible to distinguish the sets after concatenation. Hence, we use different functions to represent A and B, namely,

- represent A using a $(1, 1)$-masked superstring,
- represent B using a $(2, 2)$-masked superstring, and
- interpret the result as a $(1, 1)$-masked superstring.

This computes the difference correctly as all k-mers represented in B are treated as ghosts in the result, the k-mers from A but not from B still have a single ON occurrence and thus are correctly considered represented, and finally, the ghost k-mers in either of the initial sets or the boundary k-mers have no influence on the result. The same functions can be used if we subtract more than a single set. Furthermore, this scheme can be generalized to *any set operations on any number of sets*, by representing the i-th input set with (i, i)-MS and using a suitable demasking function for the result of the concatenation (constructed similarly as $f_{\mathbf{op}}$ for symmetric operation **op** above).

The downside to this approach is that the $(2, 2)$ function is not comprehensive and we cannot simply use any superstring of k-mers in B, but we need a superstring such that every k-mer of B appears at least twice, which can for instance be achieved by doubling the computed superstring of B.

We remark that this is the best we can do as set difference cannot be achieved with comprehensive functions solely as a result of Theorem 12.

Theorem 12. *There is no demasking function f_o and no comprehensive demasking functions f_1 and f_2, such that the result of (f_1, f_2, f_o)-concatenation would always represent the set difference of the originally represented k-mer sets.*

Proof of Theorem 12. Consider a k-mer appearing only once in both the superstrings. Then we can get the same result as if the k-mer appeared in the first set (and hence should be treated as represented) as if it appeared in the second set as either represented or ghost (in which cases it should be treated as ghost in the first). This however means that no matter the function f_o, it cannot be correctly representing the difference.

□

Other applications. Furthermore, there are many more demasking functions that can be used with f-masked superstrings, although they may not correspond to set operations. In Appendix B, we mention the **and** and **all-or-nothing** demasking functions that could be useful for some applications (see also Table 5.1).

5.3 f-Masked Superstrings as a Data Type

After seeing f-masked superstrings as an algebraic framework in Section 5.2, here we demonstrate how to turn them into a standalone data type for k-mer sets supporting all the key operations. This consists of using the FMS-index (Chapter 4) as the underlying data structure and using its capabilities to perform set operations as well as to use membership queries on f-masked superstrings. One specific prototype implementation is then described and evaluated in Chapter 6.

5.3.1 Using FMS-Index with f-Masked Superstrings

We can utilize the FMS-index to work with general f-masked superstrings and not only with **or**-masked superstrings. The only part which changes is how to determine the presence or absence as now it can be an arbitrary demasking function f. However, as in the membership queries, we already count the number of ON and OFF occurrences of a k-mer, we can simply determine the presence of a k-mer by evaluating f, which for comprehensive functions can be done in constant time.

5.3.2 f-MS Mask Recasting for f Transformation

To change the demasking function f to a different one without altering the represented k-mer set and the underlying superstring, we may need to *recast* the mask.

Although the recasting procedure depends on the specific function f used, for all comprehensive functions mentioned in Table 5.1, this can be done in linear time. For the **and** and **all-or-nothing** functions, recasting can be done via computing masks for **or**-masked superstrings with the maximum number of `1`s in the mask (Section 3.2.1), since those are also compatible masks also in the case of **and** and **all-or-nothing** functions. In the same manner, for the **or**, **xor**, and **one-or-nothing** functions, we can do the recasting by minimizing the number of ones in the mask (Section 3.2.2). Note that for non-comprehensive functions, recasting is in general not possible, since there may be no compatible mask for the new function.

If we deal with indexed f-masked superstrings, we can export the f-MS, then recast the mask, and index the result.

5.3.3 Performing Set Operations on Indexed k-Mer Sets

Using an indexed f-MS, set operations such as union, symmetric difference, intersection or symmetric difference can be performed directly via their associated abstract operations in Section 5.2. Indeed, we implement concatenation of masked superstrings via index merging (see Section 4.1 for details on merging). Prior to concatenating, we only need to ensure that each input set is represented using a correct demasking function as required by the operation (Table 5.1), and to recast the mask if it is not the case.

After the concatenation, depending on our use-case, we recast the mask if we need a different demasking function than the one resulting from individual operations. Finally, it may be desirable to compact the f-masked superstring in case the resulting f-masked superstring is unnecessarily large for the set it represents; more precisely, when many k-mers appear in the f-MS multiple times or there are many ghost k-mers. This can be done by compaction as described in Section 4.1 which can be easily generalized for the case of general demasking functions f.

Given that we either perform a symmetric set operation or the number of input sets is constant, all of these steps can be implemented in linear time and thus, the total time complexity of each set operation is also linear.

6 Experimental Evaluation

In this chapter, we present experimental results, comparing our approach to existing state-of-the-art methods. In Section 6.1.1 we introduce KmerCamel🐫, a tool for efficient masked superstring computation and optimization and in Section 6.1.2 we introduce FMSI, a tool for efficient indexing of masked superstrings and performing set operations on the underlying k-mer sets. In Section 6.2 we describe our experimental setup and benchmarking methodology.

In Section 6.3.1 we compare our algorithms to the existing with respect to the length of the superstring. Then in Section 6.3.2 we compare the proposed mask optimization algorithms in terms of the resulting mask compressibility. In Section 6.3.3 we evaluate both the superstrings and masks on compressibility and in Section 6.3.4 we show that on data sets that do not have the spectrum-like property, our approach outperforms the existing representations. Finally, in Section 6.4.1 we compare the performance of FMSI to existing single k-mer-set indexes and in Section 6.4.2 we demonstrate the feasibility of performing set operations on k-mer sets using f-masked superstrings.

6.1 Implementation

6.1.1 Efficient Computation with KmerCamel🐫

We implemented the two superstring approximation algorithms in a program called KmerCamel🐫, which first reads a user-provided FASTA file with genomic sequences, retrieves the corresponding k-mer set, computes a masked superstring using a user-specified algorithm and core data structure, and prints it in the `enc2` encoding (mask-cased superstring).

KmerCamel🐫 was developed in C++ and is available under the MIT license from Github (`https://github.com/OndrejSladky/kmercamel`). Both the local and global greedy algorithms (Sections 3.1.2 and 3.1.3) were implemented using two distinct data structures: one based on hash tables and k-mer hashing (Section 3.1.4) and the other based on Aho-Corasick automaton of the k-mer set (Section 3.1.5). Note, however, that currently only the hash-table-based versions are well optimized. The automaton-based implementations are currently experimental and their only advantage is that they can work with arbitrarily large k's. However, even the hashing-based versions support k's up to 64 which covers the vast majority of use cases.

Furthermore, KmerCamel🐫 also supports the optimization of the masks. It provides the possibility to minimize and maximize the number of ones in the mask (Sections 3.2.2 and 3.2.1) and to minimize the number of runs in the mask (Section 3.2.4) which uses the GLPK library (`https://www.gnu.org/software/glpk/`) to solve the resulting integer linear program.

6.1.2 Efficient Indexing with FMSI

We implemented FMS-index (Chapter 4) for indexing masked superstrings and their generalization f-masked superstring and the associated k-mer set op-

erations (Section 5.2) in a tool called FMSI (f-Masked Superstring Index). The tool supports membership queries on indexed f-masked superstrings and further provides an implementation of basic building-block operations such as exporting, merging, and compaction (Sections 5.3.1 and 5.3.2) that are used to perform set operations.

Index merging is implemented via export, concatenation of the underlying f-MSes, and then reindexing. Compaction is implemented using k-mer counting and KmerCamel🐫 to construct a superstring. All the demasking functions mentioned in Table 5.1 are supported directly by FMSI, and users can possibly add their custom ones.

FMSI was developed in C++ and is available from GitHub (`https://github.com/OndrejSladky/fmsi`) under the MIT license. The implementation uses the sdsl-lite library [GBMP14], available at `https://github.com/simongog/sdsl-lite/` to perform efficient rank queries, and it also uses KmerCamel🐫.

6.2 Experimental Setup

Experiments with KmerCamel🐫. We evaluated the masked superstrings computed by KmerCamel🐫, specifically, we used hash-table based implementations of both BiDir-LocalGreedy and BiDir-GlobalGreedy . We refer to these as to local and global greedy respectively. As a model species, we used *S. pneumoniae* genome (`NC_011900.1`, $n = 1$, genome length 2.22 Mbp) and pan-genome (computed from 616 assemblies from a study of children in Massachusetts, USA [CFP+15]). To evaluate the behaviour of the representations and the algorithm itself on a variety of different types of de Bruijn graphs, we also used varying values of k to control the amount of branching, as well as shifting towards the pan-genome to increase the amount of branching further. To evaluate the program on larger data sets, we also used the human genome (`GRCh38.p14`, genome length 3.1 Gbp). In Section 6.3.4, we evaluated the performance on sub-sampled k-mer sets. To assess the generality of our findings, we sought to redo the analysis using additional bacterial and viral genomes and pan-genomes, specifically a *S. cerevisiae* genome ($n = 1$, genome length 12.2 Mbp), a *SARS-CoV-2* pan-genome ($n = 590$ k), and an *E. coli* pan-genome (obtained as a union of k-mers of *E. coli* genomes from the 661k collection [BHM+21]). We found exactly the same patterns as with the *S, pneumoniae* data sets (all the data and plots are provided in the supplementary GitHub repository which is available at `https://github.com/OndrejSladky/bc-thesis-supplement`).

Experiments with FMSI. We evaluated the performance of FMSI on bacterial and viral pan-genomes and on a nematode genome. Evaluation on the efficiency of the construction and membership queries (Section 6.4.1) was done using an *E. coli* pan-genome (obtained as a union of k-mers of *E. coli* genomes from the 661k collection [BHM+21]) and *S. pneumoniae* pan-genome (computed from 616 assemblies from a study of children in Massachusetts, USA [CFP+15]) and further verified on a *SARS-CoV-2* pan-genome ($n = 14.7$ M). We measured storage space for each index, both the time and memory requirements for construction and the time and memory requirements for queries with isolated k-mers. We tested both positive and negative dataset, to obtain negative queries,

we took a random subset of 10^5 distinct k-mers from a part of chromosome 1 of the human genome (genome assembly GRCh38.p14), excluding those in the queried dataset. We further evaluated memory-efficiency of FMSI on set operations (Section 6.4.2), namely the computation of unions, intersections and symmetric differences of k-mer sets. The evaluation was performed on two different roundworm genomes, *C. elegans* (`NC_003279.8`, 100M base pairs) and *C. briggsae* (`NC_013489.2`, 108M base pairs) as two genomes which share similarities due to their relatedness but do not share the majority of the genome. All the data and plots are provided in the supplementary GitHub repository, available at `https://github.com/OndrejSladky/bc-thesis-supplement`.

All the algorithms in all the experiments were run on a single thread on a server with AMD EPYC 7302 (G GHz) processor and 251 GB RAM.

6.3 Experiments on Masked Superstrings

6.3.1 Superstring Length

We first sought to evaluate whether the proposed greedy algorithms (Section 3.1) can provide better superstring than the existing state-of-the-art solutions for computing (r)SPSS representations, namely ProphAsm [BBK21], eulertigs [SA23], greedy matchtigs, and optimal matchtigs [SKA+23]. The length of the individual superstrings corresponds to the cumulative length in the case of (r)SPSS representations.

We evaluated the algorithms on a variety of different data sets. In particular, on three genomes with increasing genomes lengths (*S. pneumoniae*, *S. cerevisiae* and a human genome) and on three pan-genomes with increasing number of genomes (*S. pneumoniae*, *E. coli* and *SARS-CoV-2*). Figure 6.1 depicts the results as the number of characters in the superstring per k-mer. As some of the tested algorithms required more computational resources than what we had available, we stopped the execution of the algorithms after a day of computation in the case of the human genome and after 10 hours in the case of all other data sets. We also ceased the execution if the algorithms required more memory than 200GB for the human genome and 60GB for the rest. We excluded the time to compute unitigs, which are required by eulertigs and matchtigs, from the maximum allowed time.

We focused on situations in which simplitigs were unable to approach the lower bound given by the number of k-mers ([BBK21, Figure 2]). For *S. pneumoniae* this corresponds to the range of k between 10 and 15 and with larger genomes the range shifts towards larger k, in general being around $\log_4 |K|$. For smaller values of k, the sets contain almost all k-mers and all the algorithms are able to nearly attain the non-tight lower bound of 1.0 characters per k-mer. Similarly for larger values, in the case of single genomes, the graphs contain a few non-branching paths and thus can be efficiently represented using all representations. For pan-genomes, the sizes of all representations also decrease, but not to 1 character per k-mer due to the branching at polymorphic sites. We remark that when we shifted towards larger k-mer sets, such as those corresponding to the human genome, all eulertigs, greedy matchtigs and optimal matchtigs did not computationally scale well for the branching de Bruijn graphs as we were unable to compute them in 200GB of RAM.

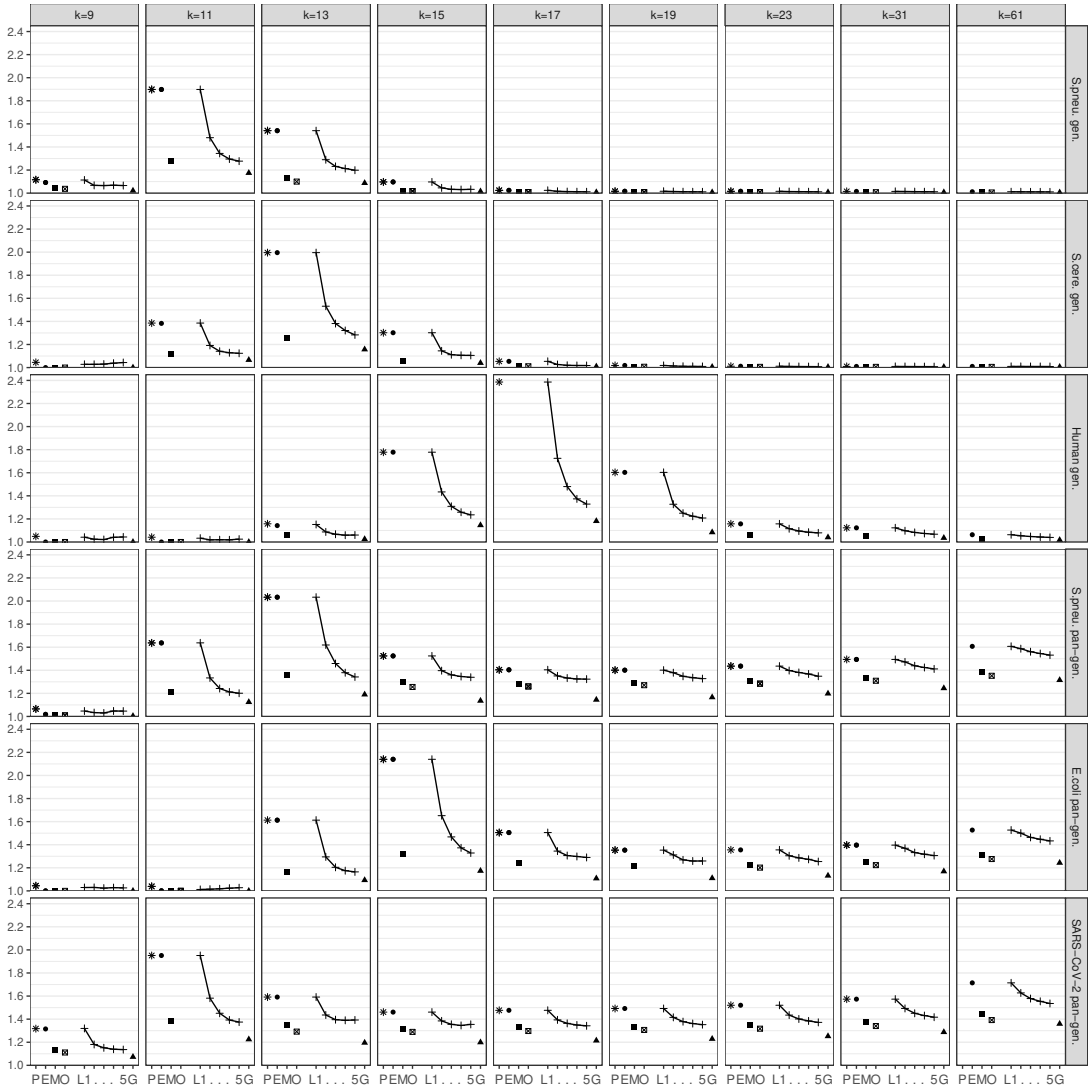


Figure 6.1: **Comparison of algorithms for k-mer superstrings** measured by superstring (or mask) characters per distinct canonical k-mer; 1.0 is a (non-tight) lower bound. The benchmark was performed using a *S. pneumoniae* genome (`NC_011900.1`, $n = 1$, genome length 2.22 Mbp), a *S. cerevisiae* genome ($n = 1$, genome length 12.2 Mbp), a human genome (`GRCh38.p14`, genome length 3.1 Gbp), a *S. pneumoniae* pan-genome (dataset [CFP+15], $n = 616$, total genome length 1.27 Gbp), an *E. coli* pan-genome (union of *E. coli* genomes from the 661k collection [BHM+21]), and a *SARS-CoV-2* pan-genome ($n = 590$ k), and evaluated for a range of k-mer lengths between 9 and 31; for the human genome we also included $k = 61$. We compared state-of-the-art tools PROPHASM (P), eulertigs (E), greedy matchtigs (M), and optimal matchtigs (O) as well as our proposed algorithms local greedy (L, with $d_{\max} \in \{1, \ldots, 5\}$) and global greedy (G), both in their hash-table-based implementations. Note that the optimal matchtigs, greedy matchtigs, and eulertigs are missing for some values of k for the human as the algorithm did not finish in 1 day and 200 GB of memory. Optimal matchtgs are missing some values of k for other datasets as the algorithm did not finish in 10 hours and 60 GB of RAM. We excluded the time for computing unitigs from the maximum time. Furthermore, for the value $k = 61$ the output of PROPHASM is missing as it supports only k-mer sizes up to 32. We remark that local greedy with $d_{\max} = 1$ (L1) is equivalent to PROPHASM (P) as indeed confirmed by the results.

For the situation, where the lower bound of 1.0 character per k-mer is not easily attainable, we identify three levels of performance:

- **Level 1: Global greedy.** The global greedy algorithm consistently outperforms all the other algorithms by at least $\sim 15\%$. On all tested genomes and pan-genomes it has at most 1.4 character per k-mer providing a very good performance even for pan-genomes.
- **Level 2: Matchtigs and local greedy.** Matchtigs (both greedy and optimal) and local greedy with large enough value of $d_{\max}$ (which for $k \leq 20$ is 4 and 5) have a comparable performance, producing representations about 15% longer than those produced by global greedy. For some values of k, matchtigs performed better than local greedy, while for other, especially for values of k corresponding to the most branching de Bruijn graphs, local greedy performed better by a tiny margin. However, once the size of k increases to 61, the values of $d_{\max}$ up to 5 are too low and the performance of local greedy degrades. Note that for many values of k we were not able to compute matchtigs optimally even for smaller data sets in reasonable time limits. For the values where we were able to compute them, they performed similarly to greedy matchtigs; in some cases slightly better.
- **Level 3: Simplitigs/SPSS.** Both simplitigs that were computed heuristically (PropHAsm and local greedy with $d_{\max} = 1$), and optimal simplitigs (eulertigs) produce the longest representations, each requiring up to 2.4 characters per k-mer.

In summary, the global greedy algorithm computed superstrings of smallest length among evaluated algorithms, improving upon matchtigs by about 15%. Viewed from a different perspective, global greedy halved the gap between matchtigs and the non-tight lower bound of 1.0 bits per k-mer.

Based on these results, to simplify further evaluation, we use only a few representations. Since all the SPSS representations (PropHAsm, eulertigs and KmerCamel🐪's local with $d_{\max} = 1$) yield similar results, we use only eulertigs, since these are the optimal form of these representations. For matchtigs, we further use only the greedy matchtigs, as the optimal matchtigs could not be computed for all values of k within 10 hours and for the remaining values their results were comparable. We further use global greedy and local greedy only with $d_{\max} = 5$.

6.3.2 Mask Compressibility

As superstrings are in general not very well compressible, requiring around 2 bits per character, we consider the compressibility of masks. In particular, we first evaluate the compressibility of each mask algorithm for different superstring algorithms used when compressed with different compression algorithms. The considered masks are

- the default mask, which for the KmerCamel🐪's algorithms is the mask produced, for the (r)SPSS representations, it is the mask associated with them if we represent them as masked superstrings (see Theorem 1),
- the mask maximizing the number of ones,

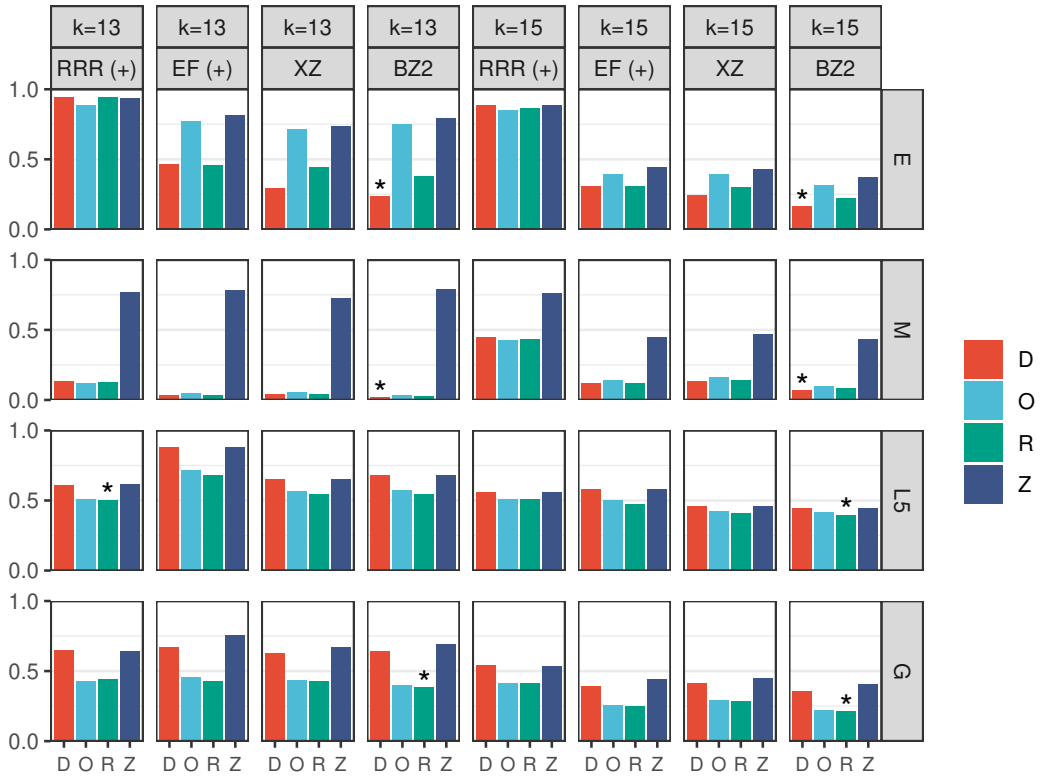


Figure 6.2: **Mask compression sizes for different mask algorithms with different compressors** and superstring algorithms by bits of mask after compression per superstring character; the value of 1.0 corresponds to the size of the mask if stored as a bit vector. The comparison was performed using the k-mer superstrings computed previously for the *S. pneumoniae* pan-genome. Individual heuristics: D – the default mask from the superstring algorithms, O – the maximal number of ones, R – the minimal number of runs, and Z – the minimum number of ones (computed greedily). Evaluated on output superstrings from eulertigs (E), greedy matchtigs (M), local greedy with $d_{\max} = 5$ (L), and global greedy (G) and compressed with different compression algorithms, `RRR` (RRR), `Elias-Fano` (EF), `xz -9` (XZ), and `bzip2 -9` (BZ2). The compressors supporting random access (`RRR` and `Elias-Fano`) are marked with (+). Note that as the values are per superstring characters, the values are not comparable between different superstring algorithms as the superstrings have different lengths. We mark the best result for each value of k and each superstring algorithm with an asterisk.

- a mask minimizing the number of ones (which is done greedily, corresponding to the lexicographically largest among the minimizing masks),
- and a mask minimizing the number of runs in the mask.

For compression, we used `xz` (which is based on the Lempel-Ziv algorithm [ZL77]), `bzip2` (a compressor based on BWT), and two compressors supporting random access: `RRR` [RRS07] and `Elias-Fano` [Eli74; Fan71] (which is based on RLE).

In Figure 6.2 we present the results for the *S. pneumoniae* pan-genome. For simplicity, we focus only on the results for *S. pneumoniae* pan-genome, as the results for other data sets are qualitatively the same (see also the supplementary

repository[1]). Figure 6.2 for simplicity also depicts the results only for k being 13 and 15. Results for k between 11 and 14 are qualitatively similar to $k = 13$ (for general data sets this corresponds to roughly $\log_4 |K|$) as these correspond to the most branching de Bruijn graphs. For larger values, the results resemble those for $k = 15$ with the compression size steadily decreasing with increasing k. For the practically not very interesting cases with very low values of k which correspond to almost all k-mers being present, all the approaches produce very compressible masks.

Regarding different compression algorithms, generally best results were obtained with `bzip2`, which either produced the most compact representation, or in the rare cases such as local greedy with $k = 13$ in Figure 6.2 only by negligible margins worse that the other tools. This trend was consistent across all data sets, even with very large sets, such as the human genome. Therefore, if random access is not required, we suggest using `bzip2`. If random access is desired, in most cases `Elias-Fano` provided better results. However, for $k \sim \log_4 |K|$ and the local or global algorithms, `RRR` was better as those cases contained many smaller runs which were difficult to compress using `Elias-Fano`.

Regarding the mask algorithms, if `RRR` is used, maximizing the number of ones in the mask is the best choice regardless of the superstring algorithm, as `RRR` can use the higher differences between the number of `1`s and `0`s for better compression. For `Elias-Fano`, the best results were obtained with the masks minimizing the number of runs, which is also not surprising as `Elias-Fano` is based on RLE. For `xz` and `bzip2`, the most compressible masks for the (r)SPSS were the default masks since the compressors can take advantage of runs of zeros being long exactly $k-1$ characters. For the global and local greedy, the masks minimizing the number of runs of ones performed the best, with the masks maximizing the number of ones being almost as compressible. Note that maximizing the number of ones can be preferred to minimizing the number of runs as it is computationally less demanding.

6.3.3 Masked Superstrings Compressibility

We further evaluated the compressibility of both superstrings and their masks in encoding `enc1` from Figure 2.1 when compressed with `xz` on *S. pneumoniae* genome and pan-genome, see Figure 6.3. As masks we used the generally most compressible masks for each superstring algorithm when compressed with `xz`. In particular, we used default masks with eulertigs and matchtigs and we used the masks with the minimum number of runs for local and global greedy algorithms. As a result, the global greedy and matchtigs performed the best, both having between 2 and 3 bits per k-mer. Their results are comparable, with matchtigs being slightly better for most values of k (9–15), with the highest difference for the value of 13, and global greedy being slightly better for other values of k (16–18). Local greedy performed worse except for the cases of $k \leq 10$, where it performed the best due to the predictable structure of the superstring. We note, however, that these cases are not very interesting from the practical point of view. Eulertigs were the worst compressible, in the worst cases requiring more than 4 bits per k-mer.

[1] `https://github.com/OndrejSladky/bc-thesis-supplement`

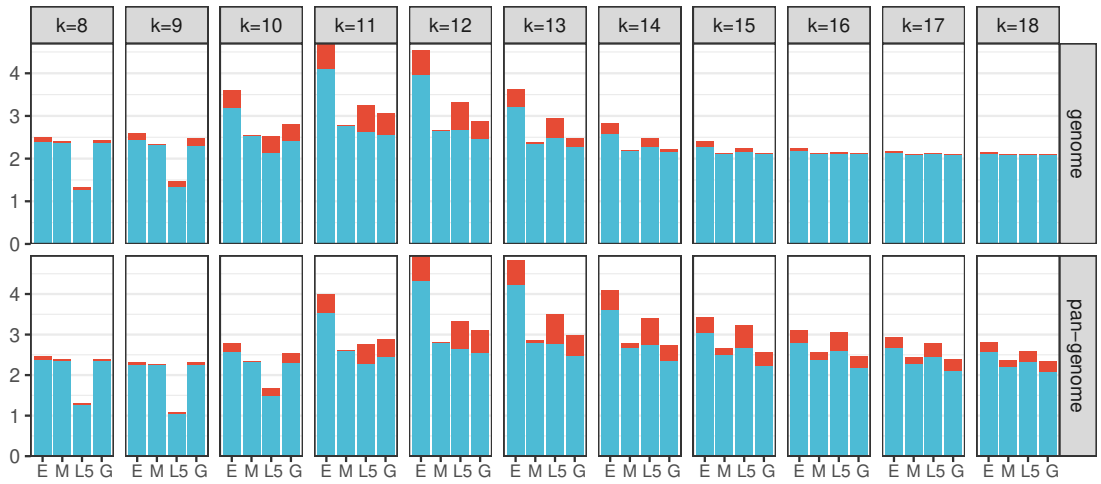


Figure 6.3: **Comparison on masked superstring compressibility** measured by bits per distinct canonical k-mer after superstring (S) and mask compression ('`xz -9`' of `enc1` in Fig. 2.1, with the most compressible masks for each heuristic as in Fig. 6.2). The benchmark was performed using a *S. pneumoniae* genome (`NC_011900.1`, $n = 1$, genome length 2.22 Mbp) and its pan-genome (dataset [CFP+15], $n = 616$, total genome length 1.27 Gbp), and evaluated for a range of k-mer lengths comparing eulertigs (E), greedy matchtigs (M), local greedy (L, with $d_{max} = 5$) and global greedy (G),

6.3.4 Sub-Sampled k-Mer Sets

In order to verify that masked superstrings work well on variety of different datasets, we sought to evaluate it on datasets where the current approaches are not near optima. As an example of such a dataset, we evaluated all the methods on sub-sampled k-mer sets, which has lower spectrum-like property than genomes and pan-genomes. This is due to the fact that removing k-mers may break the large strings into several smaller ones. We considered the k-mer sets of *S. pneumoniae* pan-genome for various k and considered sub-sampling rates r from 10^{-4} to 1. In particular, the sub-sampled k-mer set for a given k and sub-sampling rate $r < 1$ was chosen as a uniformly random subset of $r \cdot n$ distinct k-mers of the pan-genome, where n is their total number in the pan-genome. For all values of k, the observed pattern is very similar (Figure 6.4).

Due to sub-sampling, the compressibility of the masked superstring worsened substantially for all algorithms, especially for local heuristics including simplitigs and matchtigs. Indeed, when the sub-sampling rate is small, namely $r \leq 0.1$, the compressibility in Figure 6.4 is around $2k$ bits per distinct k-mer for all of the local methods that rely on the existence of long paths in the de Bruijn graph. Intuitively, this means that the average size of (weakly) connected components in the de Bruijn graph is a small constant. In the limit of decreasing rate, this value of $2k$ bits per k-mer is attained for all algorithms as in the extreme case there is only a single k-mer.

For higher compression rates, the results of local methods gradually get better, though even for rate $r = 0.9$, the compressibility is substantially worse compared to the not sub-sampled case (Figure 6.3). The global greedy algorithm is now a clear winner, being more than two times better than matchtigs in terms of the

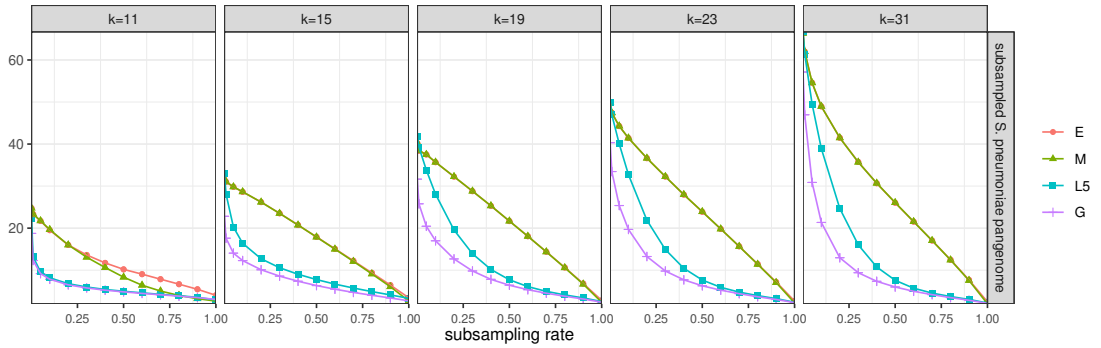


Figure 6.4: **Comparison on masked superstring compressibility on subsampled pan-genome** by bits per distinct canonical subsampled k-mer after superstring and mask compression ('`xz -9`' of `enc1` in Fig. 2.1, with the default masks produced by individual superstring heuristics). The benchmark was performed using a subsampled *S. pneumoniae* pan-genome (dataset [CFP+15], $n = 616$, total genome length 1.27 Gbp), with different values of k and subsampling rates r; specifically we sampled a uniformly random subset with an r-fraction of all distinct k-mers of the pan-genome. Comparison of local greedy (L, with $d_{max} = 5$) and global greedy (G), to eulertigs (E) and greedy matchtigs (M).

final compressed sizes for most rates and values of k. Finally, we observe that local greedy's performance with $d_{max} = 5$ substantially outperforms matchtigs and improves with increasing d_{max}.This is because both global greedy and local greedy with $d_{max} > 1$ take advantage of overlaps shorter than $k-1$, which survive sub-sampling with higher probability.

Finally, we observe that the results of the (r)SPSS representations are in many cases exactly the same. This again follows from that the de Bruijn graph has tiny (weakly) connected components and therefore, there remain only a few choices in terms of the path selection.

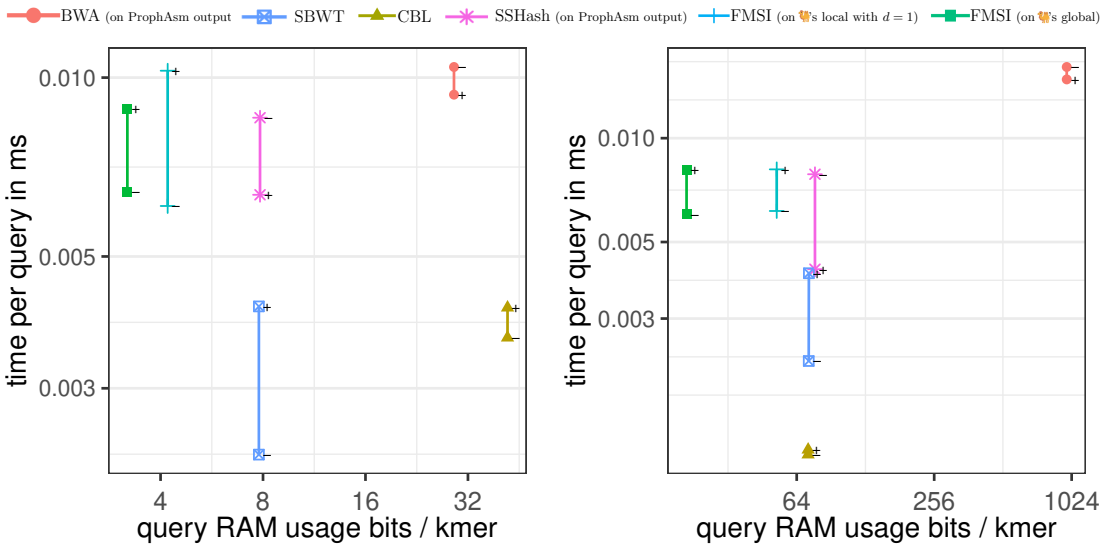
6.4 Experiments with the FMS-Index

6.4.1 Membership Queries

We compared time and memory requirements for processing both positive and negative queries of FMSI to state-of-the-art programs for indexing individual k-mer sets, namely,

- **BWA**[2] [LD09], a state-of-the-art aligner based on the FM index; for processing queries, we used the `fastmap` command [Li12], run with parameter $w = 999999$ on the simplitigs computed by PropHAsm [BBK21],
- **SBWT**[3] [APV23], an index based on the spectral Burrows-Wheeler transform; we used the default plain-matrix variant as it achieves the best query times in [APV23],
- **CBL**[4] [MCLM24], a very recent method based on smallest cyclic rotations of k-mers, and

[2] `https://github.com/lh3/bwa`, commit `139f68f`.
[3] `https://github.com/algbio/SBWT`, commit `c433b53`.
[4] `https://github.com/imartayan/CBL`, commit `8e8f28e`.



(a) $k = 23$, without subsampling k-mers. (b) $k = 23$, subsampled k-mers at rate 0.1.

Figure 6.5: **Query time and memory for *S. pneumoniae* pangenome.** We plot two points, one for positive queries (+) and one for negative (-), connected by a line.

- **SSHash**[5] [Pib22; Pib23], an index based on minimal perfect hashing of k-mers.

We have run FMSI on the masked superstrings computed by KmerCamel🐫, specifically, the global and local greedy algorithms (local is run with $d = 1$).

This experiment was done using an *E. coli* pan-genome [BHM+21] and *S. pneumoniae* pan-genome [CFP+15] and further verified on a *SARS-CoV-2* pan-genome ($n = 14.7\ M$). To verify the behavior across diverse datasets, we also provide experimental results for *subsampled* k-mer sets of these three pan-genomes; we note that after subsampling the spectrum-like property (SLP) no longer hold. Specifically, for a given subsampling rate $r \in [0, 1]$, we selected a uniformly random subset of $r \cdot N$ distinct k-mers of the original pan-genome, where N is the total number of k-mers of the pan-genome.

The results on the *E. coli* pan-genome for $k = 23$ without subsampling and with subsampling at rate 0.1 are presented in Figure 6.5; for further results, we refer to the supplementary repository[6]. Across all of the datasets, values of k, and subsampling rates, FMSI run on the masked superstring computed by KmerCamel🐫's global greedy required 3-10 times less memory for processing queries than all of the other methods, attaining around 3-4 bits per k-mer on non-subsampled *E. coli* pan-genome. However, FMSI was among the slowest from the tested methods for processing queries, performing about the same as SSHash and requiring about twice much time than SBWT. We believe that this result is mainly due to the prototype nature of our implementation, and that the query time of FMSI can still be substantially optimized. SBWT (in the plain-matrix variant) and CBL are generally the fastest algorithms for processing queries. We note that SBWT required substantial disk space during index construction (up to

[5] `https://github.com/jermp/sshash`, commit `5a13d6`.
[6] `https://github.com/OndrejSladky/bc-thesis-supplement`

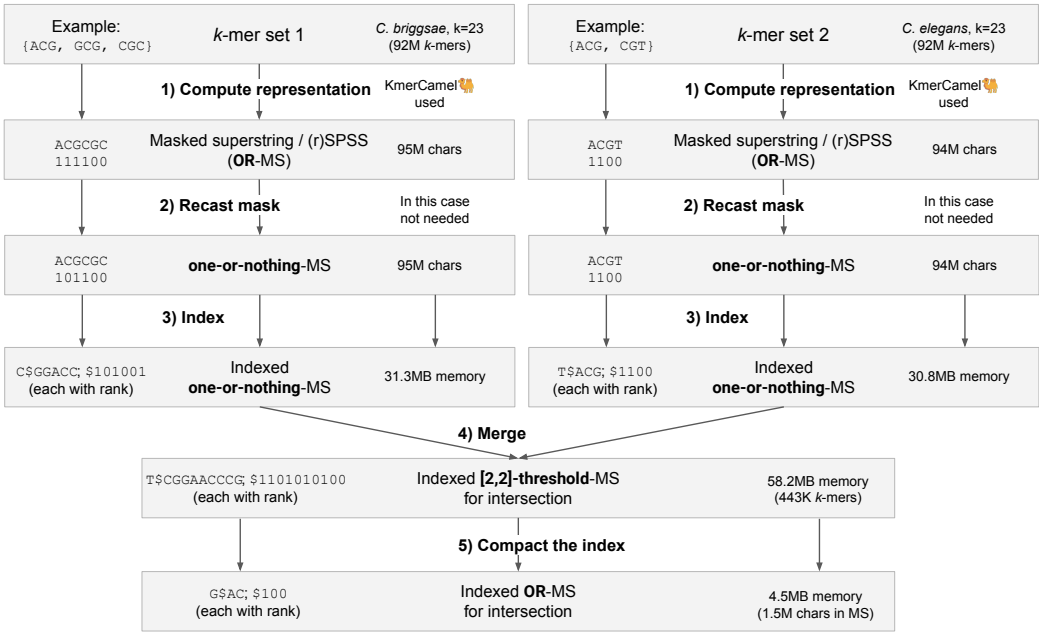


Figure 6.6: **Set operations workflow using f-masked superstrings: example on intersection.** The workflow on intersection also contains an illustrative example on a set of 3-mers as well as time for the operations and experimental data for running the workflow on *C. briggsae* and *C. elegans* genomes with $k = 23$. The experimental data contains the number of MS characters after each change, number of represented k-mers and for each indexed representation the memory required for querying the underlying set.

tens of GBs for the *E. coli* pan-genome). We also remark that while the memory usage per k-mer grows with decreasing subsampling rate, the query time remains roughly the same for all algorithms.

6.4.2 Set Operations

We demonstrate the feasibility of using FMSI to perform set operations on k-mer sets. Our proposed pipeline for set operations, as depicted in Figure 6.6, consists of five steps: First, we compute a textual representation of the k-mer sets interpreted as **or**-masked superstrings. In our experiments, this was done using KmerCamel🐫's global greedy algorithm. Second, we recast the mask to the desired demasking function, specifically we keep **or** for union and change to **one-or-nothing** for intersection and to **xor** for symmetric difference. In the case of **or**-MS computed by KmerCamel🐫, mask recasting is actually not needed as the output already minimizes the number of 1s in the mask. Then we index the f-masked superstrings using FMSI (this can be done even before mask recasting). The last two steps are concatenating the two FMS-indexes by index merging and compacting the resulting FMS-index if needed. Note that once indexed, we can ask membership queries on the resulting k-mer sets.

For this experiment, we used genomes of *C. elegans* and *C. briggsae*. We evaluated the superstring length of each computed f-masked superstring, and the memory requirements to perform queries on the indexed individual and

concatenated f-masked superstrings, both before and after compaction. The results for intersection of the two roundworm genomes for $k = 23$ are depicted in Figure 6.6. Overall, at every step, the memory required to query the indexed f-masked superstring was around 3 bits per superstring character, which in case of the roundworms was almost the same as the number of k-mers. This trend continues even for the merged FMS-index, albeit for the compacted concatenated result the per-k-mer memory was higher as it was as low as latent memory required to run FMSI. Note also that the fact that compaction significantly reduces the masked superstring length highly depends on the particular use case, namely on the proportion of represented k-mers in the result. For union and symmetric difference for the same data, the compaction led to almost negligible length reduction. For data about symmetric difference and union as well as for other values of k, see the supplementary repository[7].

[7] `https://github.com/OndrejSladky/bc-thesis-supplement`

Conclusion

We have developed a new concept for text-based representations of sets of k-mers, which we call the masked superstring of k-mers. We have shown that masked superstrings unify the theory and even generalize the existing (r)SPSS representations, which provides additional flexibility to optimize masked superstrings for specific applications.

We have also demonstrated that optimizing masked superstrings is in general an NP-hard problem, but we have developed efficient local and global approximation algorithms and implemented them in a tool called KmerCamel🐫. Both algorithms come in two variants: first, highly optimized based on k-mer hashing and second, a prototype implementation of the asymptotically optimal solution using Aho-Corasick automaton. Moreover, we have studied the optimization of masks, designed algorithms for it and implemented them in KmerCamel🐫 as well.

We have shown that using masked superstrings can lead to simplifications of data-structures for k-mer sets; we have developed the FMS-index, a simplified version of the FM-index based on masked superstrings which eliminates the necessity of storing the sampled suffix array and to locate original coordinates for each found occurrence.

Furthermore, we have generalized the framework of masked superstrings to function-assigned masked superstrings, where we interpret which k-mers are represented based on a demasking function f. We have studied several natural demasking functions and we have shown that with f-masked superstrings, we can perform set operations on k-mer sets via a simple concatenation of f-masked superstrings. This renders f-masked superstrings as an abstract data type for k-mer sets. When this approach is combined with the FMS-index, it provides a full-text k-mer index with support for set operations, which we implemented in a prototype called FMSI.

To showcase the practical usefulness of masked superstrings and their generalization, f-masked superstrings, we evaluated the proposed algorithms using viral, bacterial, and eukaryotic genomes and pan-genomes. We demonstrated that masked superstrings provide better compression characteristics than simplitigs and can provide an improvement of up to several factors over the (r)SPSS, including matchtigs, for data that do not satisfy the spectrum-like property. Furthermore, we have shown that using the FMS-index with masked superstrings can provide significant memory savings compared to state-of-the-art single set indexes, while still providing a competitive query time.

This work opens several directions for future research:

- Regarding the optimization of masked superstrings, although we proved that the problem is NP-hard in case our objective is the length of the superstring, for other practically interesting objectives, this is not known (Table 3.1). Even if other objectives turn out to be NP-hard, it would still be desirable to develop algorithms that simultaneously optimize both the superstring and the mask unlike in our two-step method where we first optimize the superstring and then the mask with the superstring already fixed. We believe that with a suitable objective, this could provide even better results for compressibility than our two-step method, even in cases where our approach

was comparable to matchtigs.

- We noted that masked superstrings were able to halve the gap of superstring length to the non-tight lower bound of 1 character per k-mer compared to matchtigs in the case of branching de Bruijn graphs. Although our algorithms do not compute the optimal solution, we believe that the output of the global greedy algorithm is already very close to the shortest superstring. We leave the study of stronger lower bounds and of the limits of textual representations to a future work.
- We have only considered indexing a single k-mer set and another significant direction is to extend the FMS-index for indexing large collections of many k-mer sets. Furthermore, our index currently supports only isolated queried k-mers, but additional support for streaming queries is desired. Although our index supports set operations, efficient support of single additions and deletions is also left to future work.
- Although to perform symmetric set operations, only a single concatenation is needed, to perform asymmetric operations, we need to copy some inputs, which is undesirable. Since this is not possible to resolve with the current f-MS framework, perhaps a generalization of masks to more different characters will be needed to resolve this.
- We remark the current software implementations can be improved. While we believe that the hashing-based implementations in KmerCamel🐫 are well optimized, the AC-automaton-based have a room for improvement, especially in the design of memory and time-efficient implementations of the AC automaton. Regarding FMSI, we believe it can be further optimized, both in terms of time and memory requirements. Especially, index merging, mask recasting, and compaction can be optimized significantly.

In conclusion, we see masked superstrings as a unifying and generalizing concept that enables to better mathematically study and optimize k-mer set representations. We further envision our research on f-masked superstrings as a first step towards a space- and time-efficient library for analyzing k-mer sets which includes all the aforementioned features.

Bibliography

[AC75] Aho, Alfred V.; Corasick, Margaret J. Efficient String Matching: An Aid to Bibliographic Search. *Commun. ACM*. 1975, vol. 18, no. 6, pp. 333–340. ISSN 0001-0782. Available from DOI: 10.1145/360825.360855.

[APV23] Alanko, Jarno N; Puglisi, Simon J; Vuohtoniemi, Jaakko. Small Searchable κ-Spectra via Subset Rank Queries on the Spectral Burrows-Wheeler Transform. In: *SIAM Conference on Applied and Computational Discrete Algorithms (ACDA23)*. SIAM, 2023, pp. 225–236. Available from DOI: 10.1137/1.9781611977714.20.

[AVMP23] Alanko, Jarno N; Vuohtoniemi, Jaakko; Mäklin, Tommi; Puglisi, Simon J. Themisto: a scalable colored k-mer index for sensitive pseudoalignment against hundreds of thousands of bacterial genomes. *Bioinformatics*. 2023, vol. 39, no. Supplement_1, pp. i260–i269. Available from DOI: 10.1093/bioinformatics/btad233.

[AKM+22] Almodaresi, Fatemeh; Khan, Jamshed; Madaminov, Sergey; Ferdman, Michael; Johnson, Rob; Pandey, Prashant; Patro, Rob. An incrementally updatable and scalable system for large-scale sequence search using the Bentley-Saxe transformation. *Bioinformatics*. 2022, vol. 38, no. 12, pp. 3155–3163. Available from DOI: 10.1093/bioinformatics/btac142.

[ASSP18] Almodaresi, Fatemeh; Sarkar, Hirak; Srivastava, Avi; Patro, Rob. A space and time-efficient index for the compacted colored de Bruijn graph. *Bioinformatics*. 2018, vol. 34, no. 13, pp. i169–i177. ISSN 1367-4811. Available from DOI: 10.1093/bioinformatics/bty292.

[ASD+20] Amarasinghe, Shanika L.; Su, Shian; Dong, Xueyi; Zappia, Luke; Ritchie, Matthew E.; Gouil, Quentin. Opportunities and challenges in long-read sequencing data analysis. *Genome Biology*. 2020, vol. 21, no. 1. ISSN 1474-760X. Available from DOI: 10.1186/s13059-020-1935-5.

[BBGI19] Bingmann, Timo; Bradley, Phelim; Gauger, Florian; Iqbal, Zamin. COBS: a compact bit-sliced signature index. In: *String Processing and Information Retrieval: 26th International Symposium, SPIRE 2019, Segovia, Spain, October 7–9, 2019, Proceedings 26*. Springer, 2019, pp. 285–303. Available from DOI: 10.1007/978-3-030-32686-9_21.

[BHM+21] Blackwell, Grace A.; Hunt, Martin; Malone, Kerri M.; Lima, Leandro; Horesh, Gal; Alako, Blaise T. F.; Thomson, Nicholas R.; Iqbal, Zamin. Exploring bacterial diversity via a curated and searchable snapshot of archived DNA sequences. *PLOS Biology*. 2021, vol. 19, no. 11, pp. 1–16. Available from DOI: 10.1371/journal.pbio.3001421.

[BJL+94] Blum, Avrim; Jiang, Tao; Li, Ming; Tromp, John; Yannakakis, Mihalis. Linear Approximation of Shortest Superstrings. 1994, vol. 41, no. 4, pp. 630–647. Available from doi: 10.1145/179812.179818.

[BOSS12] Bowe, Alexander; Onodera, Taku; Sadakane, Kunihiko; Shibuya, Tetsuo. Succinct de Bruijn Graphs. In: *Algorithms in Bioinformatics - 12th International Workshop, WABI 2012, Ljubljana, Slovenia, September 10-12, 2012. Proceedings*. Springer, 2012, vol. 7534, pp. 225–235. Lecture Notes in Computer Science. Available from doi: 10.1007/978-3-642-33122-0_18.

[BDR+19] Bradley, Phelim; Den Bakker, Henk C; Rocha, Eduardo PC; McVean, Gil; Iqbal, Zamin. Ultrafast search of all deposited bacterial and viral genomic data. *Nature Biotechnology*. 2019, vol. 37, no. 2, pp. 152–159. Available from doi: 10.1038/s41587-018-0010-1.

[BGW+15] Bradley, Phelim; Gordon, N Claire; Walker, Timothy M; Dunn, Laura; Heys, Simon; Huang, Bill; Earle, Sarah; Pankhurst, Louise J; Anson, Luke; De Cesare, Mariateresa, et al. Rapid antibiotic-resistance predictions from genome sequence data for Staphylococcus aureus and Mycobacterium tuberculosis. *Nature Communications*. 2015, vol. 6, no. 1, p. 10063. Available from doi: 10.1038/ncomms10063.

[BPMP16] Bray, Nicolas L; Pimentel, Harold; Melsted, Páll; Pachter, Lior. Near-optimal probabilistic RNA-seq quantification. *Nature Biotechnology*. 2016, vol. 34, no. 5, pp. 525–527. Available from doi: 10.1038/nbt.3519.

[BJJ97] Breslauer, Dany; Jiang, Tao; Jiang, Zhigen. Rotations of Periodic Strings and Short Superstrings. *J. Algorithms*. 1997, vol. 24, no. 2, pp. 340–353. Available from doi: 10.1006/jagm.1997.0861.

[Bři16] Břinda, Karel. *Novel computational techniques for mapping and classification of Next-Generation Sequencing data*. Université Paris-Est, 2016. Available from doi: 10.5281/zenodo.1045317. PhD thesis.

[BBK21] Břinda, Karel; Baym, Michael; Kucherov, Gregory. Simplitigs as an efficient and scalable representation of de Bruijn graphs. *Genome Biology*. 2021, vol. 22, no. 96. Available from doi: 10.1186/s13059-021-02297-z.

[BCM+20] Břinda, Karel; Callendrello, Alanna; Ma, Kevin C; MacFadden, Derek R; Charalampous, Themoula; Lee, Robyn S; Cowley, Lauren; Wadsworth, Crista B; Grad, Yonatan H; Kucherov, Gregory, et al. Rapid inference of antibiotic resistance and susceptibility by genomic neighbour typing. *Nature Microbiology*. 2020, vol. 5, no. 3, pp. 455–464. Available from doi: 10.1038/s41564-019-0656-6.

[BLP+23] Břinda, Karel; Lima, Leandro; Pignotti, Simone; Quinones-Olvera, Natalia; Salikhov, Kamil; Chikhi, Rayan; Kucherov, Gregory; Iqbal, Zamin; Baym, Michael. Efficient and Robust Search of Microbial Genomes via Phylogenetic Compression. *bioRxiv*. 2023. Available from doi: 10.1101/2023.04.15.536996.

[BSPK17] Břinda, Karel; Salikhov, Kamil; Pignotti, Simone; Kucherov, Gregory. ProPhyle 0.3.1.0. *Zenodo*. 2017, vol. 5281. Available from doi: 10.5281/zenodo.5237391.

[BW94] Burrows, Michael; Wheeler, David. *A block-sorting lossless data compression algorithm*. Technical Report 124, Digital Equipment Corporation, 1994.

[CR20] Cazaux, Bastien; Rivals, Eric. Hierarchical Overlap Graph. *Inf. Process. Lett.* 2020, vol. 155. Available from doi: 10.1016/j.ipl.2019.105862.

[CSR16] Cazaux, Bastien; Sacomoto, Gustavo; Rivals, Eric. Superstring Graph: A New Approach for Genome Assembly. In: *Algorithmic Aspects in Information and Management*. Springer International Publishing, 2016, pp. 39–52. isbn 978-3-319-41168-2. Available from doi: 10.1007/978-3-319-41168-2_4.

[Chi21] Chikhi, Rayan. *K-mer Data Structures in Sequence Bioinformatics*. HDR thesis, Institut Pasteur Ecole Doctorale "EDITE", 2021.

[CHM21] Chikhi, Rayan; Holub, Jan; Medvedev, Paul. Data structures to represent a set of k-long DNA sequences. *ACM Computing Surveys (CSUR)*. 2021, vol. 54, no. 1, pp. 1–22. Available from doi: 10.1145/3445967.

[CLJ+14] Chikhi, Rayan; Limasset, Antoine; Jackman, Shaun; Simpson, Jared T.; Medvedev, Paul. On the Representation of de Bruijn Graphs. In: *Research in Computational Molecular Biology*. Cham: Springer International Publishing, 2014, pp. 35–55. isbn 978-3-319-05269-4. Available from doi: 10.1007/978-3-319-05269-4_4.

[CLM16] Chikhi, Rayan; Limasset, Antoine; Medvedev, Paul. Compacting de Bruijn graphs from sequencing data quickly and in low memory. *Bioinformatics*. 2016, vol. 32, no. 12, pp. i201–i208. issn 1367-4803. Available from doi: 10.1093/bioinformatics/btw279.

[Cla97] Clark, David. *Compact PAT trees*. UWSpace, 1997. PhD thesis.

[CB11] Conway, Thomas C; Bromage, Andrew J. Succinct data structures for assembling large genomes. *Bioinformatics*. 2011, vol. 27, no. 4, pp. 479–486. issn 1367-4803. Available from doi: 10.1093/bioinformatics/btq697.

[CLRS22] Cormen, Thomas H; Leiserson, Charles E; Rivest, Ronald L; Stein, Clifford. *Introduction to algorithms*. MIT press, 2022.

[CT23] Cracco, Andrea; Tomescu, Alexandru I. Extremely fast construction and querying of compacted and colored de Bruijn graphs with GGCAT. *Genome Research*. 2023. issn 1549-5469. Available from doi: 10.1101/gr.277615.122.

[CKB+18] Crawford, Victoria G.; Kuhnle, Alan; Boucher, Christina; Chikhi, Rayan; Gagie, Travis. Practical dynamic de Bruijn graphs. *Bioinformatics*. 2018, vol. 34, no. 24, pp. 4189–4195. Available from doi: 10.1093/bioinformatics/bty500.

[CFP+15] Croucher, Nicholas J; Finkelstein, Jonathan A.; Pelton, Stephen I; Parkhill, Julian; Bentley, Stephen D; Lipsitch, Marc; Hanage, William P. Population genomic datasets describing the post-vaccine evolutionary epidemiology of Streptococcus pneumoniae. *Scientific Data*. 2015, vol. 2. Available from doi: 10.1038/sdata.2015.58.

[Dil50] Dilworth, Robert P. A Decomposition Theorem for Partially Ordered Sets. *Annals of Mathematics*. 1950, vol. 51, no. 1, pp. 161–166. issn 0003486X.

[DS14] Dinur, Irit; Steurer, David. Analytical Approach to Parallel Repetition. In: *Proceedings of the Forty-Sixth Annual ACM Symposium on Theory of Computing*. New York, New York: Association for Computing Machinery, 2014, pp. 624–633. STOC '14. isbn 9781450327107. Available from doi: 10.1145/2591796.2591884.

[Eli74] Elias, Peter. Efficient Storage and Retrieval by Content and Address of Static Files. *J. ACM*. 1974, vol. 21, no. 2, pp. 246–260. issn 0004-5411. Available from doi: 10.1145/321812.321820.

[EMV23] Englert, Matthias; Matsakis, Nicolaos; Veselý, Pavel. Approximation guarantees for shortest superstrings: simpler and better. In: *Proceedings of the 34th International Symposium on Algorithms and Computation (ISAAC 2023)*. Leibniz International Proceedings in Informatics (LIPIcs), 2023. to appear.

[FKPP23] Fan, Jason; Khan, Jamshed; Pibiri, Giulio Ermanno; Patro, Rob. Spectrum Preserving Tilings Enable Sparse and Modular Reference Indexing. In: *Research in Computational Molecular Biology - 27th Annual International Conference, RECOMB 2023, Istanbul, Turkey, April 16-19, 2023, Proceedings*. Springer, 2023, vol. 13976, pp. 21–40. Lecture Notes in Computer Science. Available from doi: 10.1007/978-3-031-29119-7_2.

[FKS+24] Fan, Jason; Khan, Jamshed; Singh, Noor Pratap; Pibiri, Giulio Ermanno; Patro, Rob. Fulgor: a fast and compact k-mer index for large-scale matching and color queries. *Algorithms for Molecular Biology*. 2024, vol. 19, no. 1, p. 3. Available from doi: 10.1186/S13015-024-00251-9.

[Fan71] Fano, Robert Mario. *On the number of bits required to implement an associative memory.* Memorandum 61, Computer Structures Group, MIT, Cambridge, MA, 1971.

[FM05] Ferragina, Paolo; Manzini, Giovanni. Indexing Compressed Text. *J. ACM*. 2005, vol. 52, no. 4, pp. 552–581. issn 0004-5411. Available from doi: 10.1145/1082036.1082039.

[Ful56] Fulkerson, Delbert R. *Note on Dilworth's decomposition theorem for partially ordered sets*. Vol. 7. American Mathematical Society (AMS), 1956. No. 4. Available from doi: 10.1090/S0002-9939-1956-0078334-6.

[GMS80] Gallant, John; Maier, David; Storer, James A. On Finding Minimal Length Superstrings. *J. Comput. Syst. Sci.* 1980, vol. 20, pp. 50–58. Available from doi: 10.1016/0022-0000(80)90004-5.

[GJ79] Garey, Michael R.; Johnson, David S. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman, 1979.

[GBMP14] Gog, Simon; Beller, Timo; Moffat, Alistair; Petri, Matthias. From Theory to Practice: Plug and Play with Succinct Data Structures. In: *13th International Symposium on Experimental Algorithms, (SEA 2014)*. 2014, pp. 326–337.

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

[GMM16] Goodwin, Sara; McPherson, John D.; McCombie, W. Richard. Coming of age: ten years of next-generation sequencing technologies. *Nature Reviews Genetics*. 2016, vol. 17, no. 6, pp. 333–351. issn 1471-0064. Available from doi: 10.1038/nrg.2016.49.

[GGV03] Grossi, Roberto; Gupta, Ankur; Vitter, Jeffrey Scott. High-order entropy-compressed text indexes. In: *Proceedings of the Fourteenth Annual ACM-SIAM Symposium on Discrete Algorithms*. Baltimore, Maryland: Society for Industrial and Applied Mathematics, 2003, pp. 841–850. SODA '03. isbn 0898715385.

[GYC+21] Gupta, Gaurav; Yan, Minghao; Coleman, Benjamin; Kille, Bryce; Elworth, R. A. Leo; Medini, Tharun; Treangen, Todd; Shrivastava, Anshumali. Fast Processing and Querying of 170TB of Genomics Data via a Repeated And Merged BloOm Filter (RAMBO). In: *Proceedings of the 2021 International Conference on Management of Data*. Virtual Event, China: Association for Computing Machinery, 2021, pp. 2226–2234. SIGMOD '21. isbn 9781450383431. Available from doi: 10.1145/3448016.3457333.

[HM20] Holley, Guillaume; Melsted, Páll. Bifrost: highly parallel construction and indexing of colored and compacted de Bruijn graphs. *Genome Biology*. 2020, vol. 21, no. 1, pp. 1–20. Available from doi: 10.1186/s13059-020-02135-8.

[HM14] Holt, James; McMillan, Leonard. Merging of multi-string BWTs with applications. *Bioinformatics*. 2014, vol. 30, no. 24, pp. 3524–3531. issn 1367-4803. Available from doi: 10.1093/bioinformatics/btu584.

[HL10] Huang, Chi-Ruei; Lo, Szecheng J. Evolution and Diversity of the Human Hepatitis D Virus Genome. *Advances in Bioinformatics*. 2010, vol. 2010, pp. 1–9. ISSN 1687-8035. Available from DOI: 10.1155/2010/323654.

[Jac89] Jacobson, G. Space-efficient static trees and graphs. In: *2013 IEEE 54th Annual Symposium on Foundations of Computer Science*. Los Alamitos, CA, USA: IEEE Computer Society, 1989, pp. 549–554. Available from DOI: 10.1109/SFCS.1989.63533.

[Jac88] Jacobson, Guy Joseph. *Succinct static data structures*. USA: Carnegie Mellon University, 1988. PhD thesis. AAI8918056.

[Joh74] Johnson, David S. Approximation algorithms for combinatorial problems. *Journal of Computer and System Sciences*. 1974, vol. 9, no. 3, pp. 256–278. ISSN 0022-0000. Available from DOI: 10.1016/S0022-0000(74)80044-9.

[KLR15] Kaplinski, Lauris; Lepamets, Maarja; Remm, Maido. GenomeTester4: a toolkit for performing basic set operations - union, intersection and complement on k-mer lists. *GigaScience*. 2015, vol. 4, no. 1, s13742-015-0097–y. ISSN 2047-217X. Available from DOI: 10.1186/s13742-015-0097-y.

[KMD+20] Karasikov, Mikhail; Mustafa, Harun; Danciu, Daniel; Barber, Christopher; Zimmermann, Marc; Rätsch, Gunnar; Kahles, André. Metagraph: Indexing and analysing nucleotide archives at petabase-scale. *BioRxiv*. 2020, pp. 2020–10. Available from DOI: 10.1101/2020.10.01.322164.

[KS13] Karpinski, Marek; Schmied, Richard. Improved inapproximability results for the shortest superstring and related problems. In: *Proceedings of the Nineteenth Computing: The Australasian Theory Symposium - Volume 141*. Adelaide, Australia: Australian Computer Society, Inc., 2013, pp. 27–36. CATS '13. ISBN 9781921770265. Available from DOI: 10.5555/2525519.2525523.

[KLA+01] Kasai, Toru; Lee, Gunho; Arimura, Hiroki; Arikawa, Setsuo; Park, Kunsoo. Linear-Time Longest-Common-Prefix Computation in Suffix Arrays and Its Applications. In: *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2001, pp. 181–192. ISBN 9783540481942. ISSN 1611-3349. Available from DOI: 10.1007/3-540-48194-x_17.

[KM95] Kececioglu, J. D.; Myers, E. W. Combinatorial algorithms for DNA sequence assembly. *Algorithmica*. 1995, vol. 13, no. 1–2, pp. 7–51. ISSN 1432-0541. Available from DOI: 10.1007/bf01188580.

[KKDP22] Khan, Jamshed; Kokot, Marek; Deorowicz, Sebastian; Patro, Rob. Scalable, ultra-fast, and low-memory construction of compacted de Bruijn graphs with Cuttlefish 2. *Genome Biology*. 2022, vol. 23, no. 1. ISSN 1474-760X. Available from DOI: 10.1186/s13059-022-02743-6.

[Kha21] Khan, Shahbaz. Optimal Construction of Hierarchical Overlap Graphs. In: Gawrychowski, Pawel; Starikovskaya, Tatiana (eds.). *32nd Annual Symposium on Combinatorial Pattern Matching, CPM 2021, July 5-7, 2021, Wrocław, Poland*. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021, vol. 191, 17:1–17:11. LIPIcs. Available from DOI: 10.4230/LIPIcs.CPM.2021.17.

[KA03] Ko, Pang; Aluru, Srinivas. Space efficient linear time construction of suffix arrays. *J. Discrete Algorithms*. 2003, vol. 3, pp. 143–156. Available also from: https://api.semanticscholar.org/CorpusID:14008339.

[KDD17] Kokot, Marek; Długosz, Maciej; Deorowicz, Sebastian. KMC 3: counting and manipulating k-mer statistics. *Bioinformatics*. 2017, vol. 33, no. 17, pp. 2759–2761. ISSN 1367-4803. Available from DOI: 10.1093/bioinformatics/btx304.

[KPS94] Kosaraju, S. Rao; Park, James K.; Stein, Clifford. Long Tours and Short Superstrings. In: *35th*. 1994, pp. 166–177. Available from DOI: 10.1109/SFCS.1994.365696.

[LLT+09] Lam, T. W.; Li, Ruiqiang; Tam, Alan; Wong, Simon; Wu, Edward; Yiu, S. M. High Throughput Short Read Alignment via Bi-directional BWT. In: *2009 IEEE International Conference on Bioinformatics and Biomedicine*. IEEE, 2009. Available from DOI: 10.1109/bibm.2009.42.

[LGB+18] Lees, John A; Galardini, Marco; Bentley, Stephen D; Weiser, Jeffrey N; Corander, Jukka. pyseer: a comprehensive tool for microbial pangenome-wide association studies. *Bioinformatics*. 2018, vol. 34, no. 24, pp. 4310–4312. Available from DOI: 10.1093/bioinformatics/bty539.

[LMCP22] Lemane, Téo; Medvedev, Paul; Chikhi, Rayan; Peterlongo, Pierre. kmtricks: efficient and flexible construction of Bloom filters for large sequencing data collections. *Bioinformatics Advances*. 2022, vol. 2, no. 1, vbac029. ISSN 2635-0041. Available from DOI: 10.1093/bioadv/vbac029.

[Li12] Li, Heng. Exploring single-sample SNP and INDEL calling with whole-genome de novo assembly. *Bioinformatics*. 2012, vol. 28, no. 14, pp. 1838–1844. ISSN 1367-4803. Available from DOI: 10.1093/bioinformatics/bts280.

[Li13] Li, Heng. Aligning sequence reads, clone sequences and assembly contigs with BWA-MEM. *arXiv preprint arXiv:1303.3997*. 2013. Available from DOI: 10.48550/arXiv.1303.3997.

[LD09] Li, Heng; Durbin, Richard. Fast and accurate short read alignment with Burrows-Wheeler transform. *Bioinformatics*. 2009, vol. 25, no. 14, pp. 1754–1760. Available from DOI: 10.1093/bioinformatics/btp324.

[LD10] Li, Heng; Durbin, Richard. Fast and accurate long-read alignment with Burrows-Wheeler transform. *Bioinformatics*. 2010, vol. 26, no. 5, pp. 589–595. Available from doi: 10.1093/bioinformatics/btp698.

[LRCP17] Limasset, Antoine; Rizk, Guillaume; Chikhi, Rayan; Peterlongo, Pierre. Fast and Scalable Minimal Perfect Hashing for Massive Key Sets. In: *16th International Symposium on Experimental Algorithms, SEA 2017, June 21-23, 2017, London, UK*. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2017, vol. 75, 25:1–25:16. LIPIcs. Available from doi: 10.4230/lipics.sea.2017.25.

[LLL+12] Liu, Lin; Li, Yinhu; Li, Siliang; Hu, Ni; He, Yimin; Pong, Ray; Lin, Danni; Lu, Lihua; Law, Maggie. Comparison of Next-Generation Sequencing Systems. *Journal of Biomedicine and Biotechnology*. 2012, vol. 2012, pp. 1–11. issn 1110-7251. Available from doi: 10.1155/2012/251364.

[MM90] Manber, Udi; Myers, Gene. Suffix Arrays: A New Method for on-Line String Searches. In: *Proceedings of the First Annual ACM-SIAM Symposium on Discrete Algorithms*. San Francisco, California, USA: Society for Industrial and Applied Mathematics, 1990, pp. 319–327. SODA '90. isbn 0898712513.

[MBP+21] Marchet, Camille; Boucher, Christina; Puglisi, Simon J; Medvedev, Paul; Salson, Mikaël; Chikhi, Rayan. Data structures based on k-mers for querying large collections of sequencing data sets. *Genome Research*. 2021, vol. 31, no. 1, pp. 1–12. Available from doi: 10.1101/gr.260604.119.

[MIG+20] Marchet, Camille; Iqbal, Zamin; Gautheret, Daniel; Salson, Mikaël; Chikhi, Rayan. REINDEER: efficient indexing of k-mer presence and abundance in sequencing datasets. *Bioinformatics*. 2020, vol. 36, no. Supplement-1, pp. i177–i185. Available from doi: 10.1093/bioinformatics/btaa487.

[MKL21] Marchet, Camille; Kerbiriou, Maël; Limasset, Antoine. BLight: efficient exact associative structure for k-mers. *Bioinformatics*. 2021, vol. 37, no. 18, pp. 2858–2865. Available from doi: 10.1093/bioinformatics/btab217.

[MCLM24] Martayan, Igor; Cazaux, Bastien; Limasset, Antoine; Marchet, Camille. Conway-Bromage-Lyndon (CBL): an exact, dynamic representation of k-mer sets. *bioRxiv*. 2024. Available from doi: 10.1101/2024.01.29.577700.

[MPM17] Minkin, Ilia; Pham, Son; Medvedev, Paul. TwoPaCo: an efficient algorithm to build the compacted de Bruijn graph from many complete genomes. *Bioinformatics*. 2017, vol. 33, no. 24, pp. 4024–4032. issn 1367-4803. Available from doi: 10.1093/bioinformatics/btw609.

[MAB19] Muggli, Martin D.; Alipanahi, Bahar; Boucher, Christina. Building large updatable colored de Bruijn graphs via merging. *Bioinformatics*. 2019, vol. 35, no. 14, pp. i51–i60. Available from doi: 10.1093/bioinformatics/btz350.

[MBN+17] Muggli, Martin D.; Bowe, Alexander; Noyes, Noelle R.; Morley, Paul S.; Belk, Keith E.; Raymond, Robert; Gagie, Travis; Puglisi, Simon J; Boucher, Christina. Succinct colored de Bruijn graphs. *Bioinformatics*. 2017, vol. 33, no. 20, pp. 3181–3187. Available from doi: 10.1093/bioinformatics/btx067.

[Nav16] Navarro, Gonzalo. Texts. In: *Compact Data Structures: A Practical Approach*. Cambridge University Press, 2016, pp. 395–449.

[NP12] Navarro, Gonzalo; Providel, Eliana. Fast, Small, Simple Rank/Select on Bitmaps. In: *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2012, pp. 295–306. isbn 9783642308505. issn 1611-3349. Available from doi: 10.1007/978-3-642-30850-5_26.

[PBJP18] Pandey, Prashant; Bender, Michael A.; Johnson, Rob; Patro, Rob. Squeakr: an exact and approximate k-mer counting system. *Bioinformormatics*. 2018, vol. 34, no. 4, pp. 568–575. Available from doi: 10.1093/bioinformatics/btx636.

[PPC+21] Park, Sangsoo; Park, Sung Gwan; Cazaux, Bastien; Park, Kunsoo; Rivals, Eric. A Linear Time Algorithm for Constructing Hierarchical Overlap Graphs. In: *32nd Annual Symposium on Combinatorial Pattern Matching, CPM 2021, July 5-7, 2021, Wrocław, Poland*. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021, vol. 191, 22:1–22:9. LIPIcs. Available from doi: 10.4230/LIPIcs.CPM.2021.22.

[PDL+17] Patro, Rob; Duggal, Geet; Love, Michael I; Irizarry, Rafael A; Kingsford, Carl. Salmon provides fast and bias-aware quantification of transcript expression. *Nature Methods*. 2017, vol. 14, no. 4, pp. 417–419. Available from doi: 10.1038/nmeth.4197.

[PFL10] Pellicer, Jaume; Fay, Michael F; Leitch, Ilia J. The largest eukaryotic genome of them all? *Botanical Journal of the Linnean Society*. 2010, vol. 164, no. 1, pp. 10–15. issn 0024-4074. Available from doi: 10.1111/j.1095-8339.2010.01072.x.

[PSTU83] Peltola, Hannu; Söderlund, Hans; Tarhio, Jorma; Ukkonen, Esko. Algorithms for Some String Matching Problems Arising in Molecular Genetics. In: *IFIP Congress*. 1983. Available also from: https://api.semanticscholar.org/CorpusID:37787264.

[Pib22] Pibiri, Giulio Ermanno. Sparse and skew hashing of K-mers. *Bioinformatics*. 2022, vol. 38, no. Supplement_1, pp. i185–i194. issn 1367-4803. Available from doi: 10.1093/bioinformatics/btac245.

[Pib23] Pibiri, Giulio Ermanno. On weighted k-mer dictionaries. *Algorithms for Molecular Biology*. 2023, vol. 18, no. 1. issn 1748-7188. Available from doi: 10.1186/s13015-023-00226-2.

[Rah23] Rahman, Amatur. *Compression Algorithms for de Bruijn Graphs and Uncovering Hidden Assembly Artifacts*. The Pennsylvania State University, 2023. PhD thesis.

[RCM21] Rahman, Amatur; Chikhi, Rayan; Medvedev, Paul. Disk compression of k-mer sets. *Algorithms for Molecular Biology*. 2021, vol. 16, no. 1. ISSN 1748-7188. Available from DOI: 10.1186/s13015-021-00192-7.

[RM21] Rahman, Amatur; Medvedev, Paul. Representation of k-Mer Sets Using Spectrum-Preserving String Sets. *Journal of Computational Biology*. 2021, vol. 28, no. 4, pp. 381–394. Available from DOI: 10.1089/cmb.2020.0431. PMID: 33290137.

[RRS07] Raman, Rajeev; Raman, Venkatesh; Satti, Srinivasa Rao. Succinct indexable dictionaries with applications to encoding k-ary trees, prefix sums and multisets. *ACM Trans. Algorithms*. 2007, vol. 3, no. 4, 43–es. ISSN 1549-6325. Available from DOI: 10.1145/1290672.1290680.

[RA15] Rhoads, Anthony; Au, Kin Fai. PacBio Sequencing and Its Applications. *Genomics, Proteomics & Bioinformatics*. 2015, vol. 13, no. 5, pp. 278–289. ISSN 1672-0229. Available from DOI: 10.1016/j.gpb.2015.08.002.

[RG19] Rice, Edward S.; Green, Richard E. New Approaches for Genome Assembly and Scaffolding. *Annual Review of Animal Biosciences*. 2019, vol. 7, no. 1, pp. 17–40. ISSN 2165-8110. Available from DOI: 10.1146/annurev-animal-020518-115344.

[Sal17] Salikhov, Kamil. *Efficient algorithms and data structures for indexing DNA sequence data.* [Université Paris-Est]. 2017. PhD thesis.

[SBPK] Salikhov, Kamil; Břinda, Karel; Pignotti, Simone; Kucherov, Gregory. *ProPhex* [https://github.com/prophyle/prophex]. GitHub, [n.d.]. Available from DOI: 10.5281/zenodo.1247432.

[SDI+16] Schirmer, Melanie; D'Amore, Rosalinda; Ijaz, Umer Z.; Hall, Neil; Quince, Christopher. Illumina error profiles: resolving fine-scale variation in metagenomic sequencing data. *BMC Bioinformatics*. 2016, vol. 17, no. 1. ISSN 1471-2105. Available from DOI: 10.1186/s12859-016-0976-y.

[Sch23] Schmidt, Sebastian. *Unitigs Are Not Enough: the Advantages of Superunitig-Based Algorithms in Bioinformatics*. University of Helsinki, 2023. PhD thesis.

[SA23] Schmidt, Sebastian; Alanko, Jarno N. Eulertigs: minimum plain text representation of k-mer sets without repetitions in linear time. *Algorithms for Molecular Biology*. 2023, vol. 18, no. 1, p. 5. ISBN 1748-7188. Available from DOI: 10.1186/s13015-023-00227-1.

[SKA+23] Schmidt, Sebastian; Khan, Shahbaz; Alanko, Jarno N.; Pibiri, Giulio E.; Tomescu, Alexandru I. Matchtigs: minimum plain text representation of k-mer sets. *Genome Biology*. 2023, vol. 24, no. 1, p. 136. ISSN 1474-760X. Available from DOI: 10.1186/s13059-023-02968-z.

[SDK23] Schulte, Sara C.; Dilthey, Alexander T.; Klau, Gunnar W. HOGVAX: Exploiting Peptide Overlaps to Maximize Population Coverage in Vaccine Design with Application to SARS-CoV-2. 2023. Available from DOI: 10.1101/2023.01.09.523288.

[SRM23] Shaw, Liam P; Rocha, Eduardo P C; MacLean, R Craig. Restriction-modification systems have shaped the evolution and distribution of plasmids across bacteria. *Nucleic Acids Research*. 2023, vol. 51, no. 13, pp. 6806–6818. Available from DOI: 10.1093/nar/gkad452.

[SBK22] Shibuya, Yoshihiro; Belazzougui, Djamal; Kucherov, Gregory. Efficient Reconciliation of Genomic Datasets of High Similarity. In: Boucher, Christina; Rahmann, Sven (eds.). *22nd International Workshop on Algorithms in Bioinformatics (WABI 2022)*. Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2022, vol. 242, 14:1–14:14. Leibniz International Proceedings in Informatics (LIPIcs). ISBN 978-3-95977-243-3. ISSN 1868-8969. Available from DOI: 10.4230/LIPIcs.WABI.2022.14.

[SVB23] Sladký, Ondřej; Veselý, Pavel; Břinda, Karel. Masked superstrings as a unified framework for textual k-mer set representations. *bioRxiv*. 2023. Available from DOI: 10.1101/2023.02.01.526717.

[SVB24] Sladký, Ondřej; Veselý, Pavel; Břinda, Karel. Function-Assigned Masked Superstrings as a Versatile and Compact Data Type for k-Mer Sets. *bioRxiv*. 2024. Available from DOI: 10.1101/2024.03.06.583483.

[SLF+15] Stephens, Zachary D; Lee, Skylar Y; Faghri, Faraz; Campbell, Roy H; Zhai, Chengxiang; Efron, Miles J; Iyer, Ravishankar; Schatz, Michael C; Sinha, Saurabh; Robinson, Gene E. Big Data: Astronomical or Genomical? *PLoS Biology*. 2015, vol. 13, no. 7, e1002195. Available from DOI: 10.1371/journal.pbio.1002195.

[TU88] Tarhio, Jorma; Ukkonen, Esko. A greedy approximation algorithm for constructing shortest common superstrings. *Theoretical Computer Science*. 1988, vol. 57, no. 1, pp. 131–145. ISSN 0304-3975. Available from DOI: 10.1016/0304-3975(88)90167-3.

[Ukk90] Ukkonen, Esko. A Linear-Time Algorithm for Finding Approximate Shortest Common Superstrings. *Algorithmica*. 1990, vol. 5, no. 3, pp. 313–323. Available from DOI: 10.1007/BF01840391.

[WS14] Wood, Derrick E; Salzberg, Steven L. Kraken: ultrafast metagenomic sequence classification using exact alignments. *Genome Biology*. 2014, vol. 15, no. 3, pp. 1–12. Available from DOI: 10.1186/gb-2014-15-3-r46.

[ZL77] Ziv, J.; Lempel, A. A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory*. 1977, vol. 23, no. 3, pp. 337–343. ISSN 0018-9448. Available from DOI: 10.1109/tit.1977.1055714.

List of Figures

List of Tables

List of Abbreviations

AC	Aho-Corasick (automaton)
ATSP	Asymmetric Traveling Salesman Problem
BWT	Burrows-Wheeler Transform
***f*-MS**	*f*-Masked Superstring / Function-Assigned Masked Superstring
EOL	End of Line
HOG	Hierarchical Overlap Graph
ILP	Integer Linear Programming
LCP	Longest Common Prefix
MS	Masked Superstring
RLE	Run-Length Encoding
rSPSS	Repetitive Spectrum-Preserving String Sets (matchtigs)
(r)SPSS	Both SPSS and rSPSS
SA	Suffix Array
SLP	Spectrum-Like Property
SPSS	Spectrum-Preserving String Sets (simplitigs)
SSP	Shortest Superstring Problem

A A Polynomial-Time Algorithm for Masked Superstrings with Globally Minimum Number of Runs

In Section 3.3 we provide approximation guarantees for objective functions which depend on the superstring length if the two-step optimization protocol (Chapter 3) is used. For other functions, there are no general guarantees for this protocol. In this appendix, we consider such a function, in particular $g(S, M) = runs_1(M)$ (see also Table 3.1), for which the two-step protocol does not produce nearly optimal result. Here, we however demonstrate that this objective function can be optimized in polynomial time.

We show that the ratio of optimal solution and the result of the two-step optimization cannot be bounded. Let a block B_i consist of k-mers AX_iCG, GCY_iA, TX_iCG, GCY_iA and S being a simplitig transitioning between the first two k-mers i.e., $X_iCG^kCY_i$. If X_i and Y_i are chosen in a way that the k-mers do not have an overlap larger than G, then the shortest superstring has two runs of ones, whereas optimum is 1. If we connect r blocks with additional transitions, the number of runs in the shortest superstring is at least r, whereas the optimum remains 1.

The minimization of the number of runs can be however solved optimally in polynomial time, employing ideas similar as in [SA23; SKA+23] First, we observe that if we split the solution on zeros, the runs of ones directly correspond to individual matchtigs. Hence, we are trying to compute matchtigs with the fewest number of sequences. An alternative view is that we aim to find the fewest walks that cover the whole de Bruijn graph. To do this, we first contract the strongly connected components. The result is then a DAG and we can find the minimum path cover in polynomial time [Dil50; Ful56].

We note that this result also corresponds to finding the minimum number of matchtigs covering the de Bruijn graph. However, despite having the least number of sequences, the cumulative length might in theory be quite large and thus, the resulting masked superstrings might not be suitable for usage. Nevertheless, this algorithm demonstrates that there are natural objective functions for masked superstrings which are polynomial-time solvable.

B Alternative Demasking Functions

In this section we provide two other demasking functions that can be useful in some situations.

B.1 The all-or-nothing-Masked Superstrings

Perhaps the simplest approach to representing a set of k-mers is to mark all occurrences of represented k-mers with one, all ghost k-mers with zero, and treat all other masks as invalid. This corresponds to a function that returns 1 if it receives a list of ones, 0 if a list of zeros (or an empty list), and $undefined$ otherwise.

This representation has its clear benefits. Most importantly, one can determine the presence or absence of a k-mer by looking at the mask at any occurrence of the k-mer. For example, this makes indexing of **all-or-nothing**-masked superstrings easier than indexing general f-masked superstrings, as we do not need to query the rank to determine the number of ON occurrences. Instead, we can simply determine the presence or absence of a k-mer based on any of its occurrences which is simpler than having to perform two rank queries on the SA-transformed mask as discussed in Chapter 4.

We could potentially achieve higher compressibility of the mask by realizing that we can infer the presence or absence of a k-mer from its first occurrence, which comes from the fact that a mask for a given set is unique. Thus, we can omit all symbols in the mask corresponding to any further occurrences of the k-mer, making the mask shorter and easier to store, while it be easily reconstructed afterwards.

We further note that **all-or-nothing**-masked superstrings can be viewed as or-masked superstrings that maximize the number of ones in the mask.

B.2 The and-Masked Superstrings

We could easily replace the **or** function with **and**. That is, we could consider a k-mer present if it is marked as present at *all* its occurrences, with the small difference that we consider a k-mer not represented if it does not appear, i.e. we consider the **and** of an empty list returning 0. This ensures that the **and** function is comprehensive.

The potential advantage of **and**-masked superstrings over **or**-masked superstrings is that we can mark ghost k-mers with ones at some occurrences and therefore obtain masks with more ones in them, which could be beneficial, for instance for potential additional improvements in compressibility.

C Local Greedy Directly on the FMS-Index

In this appendix we describe how is it possible to compute masked superstrings with the local greedy (Section 3.1.2) executed directly on the FMS-index (Chapter 4) which can be used for FMS-index compaction. We describe a version that works with general f-masked superstrings (Section 5.1) in the uni-directional model. At its core, this uses a bi-directional variant of the FMS-index based on the bi-directional FM-index [LLT+09] as the underlying data structure alongside with the SA-transformed mask. Note that the bi-directionality of the FM-index is not related to the bi-directional model, but rather that it is possible to extend the searched pattern to both directions.

Additionally to the bi-directional FM-index and the SA-transformed mask, we require a $kLCP_{0-1}$ [Sal17] data structure that for each position determines whether the longest common prefix of the two neighboring suffixes is of length at least k. The $kLCP_{0-1}$ array can be computed trivially from the LCP array which can be computed in $\mathcal{O}(|S|)$ time during the construction of the FM-index [KLA+01].

Recall that the local greedy algorithm (with a parameter $d_{\max}$) proceeds as follows. It first chooses an arbitrary k-mer that has not been represented yet. Then it tries to extend it to both sides via extensions of length d starting with $d = 1$ and going up to $d = d_{\max}$. Regarding the implementation of local greedy, there are two questions. First, how to maintain the k-mers that have not been represented yet and second, how to quickly check whether a k-mer exists.

We start with the lexicographically smallest k-mer, which is the one that appears first in the suffix coordinates, and based on the $kLCP_{0-1}$ array, we find the last occurrence of this k-mer. From the SA-transformed mask, we determine whether the k-mer is represented. If not we continue to the next k-mer.

Otherwise, we delete the k-mer Q. This can be done by finding the number m of `1`s such that for input x of size as long as the number of occurrences Q with m `1`s, $f(x) = 0$ and then setting the first m symbols in the SA-transformed mask to `1` and other to `0`. Then we try to extend it to both directions. Adding the extension characters to either direction can be done directly using the bi-directional FM-index and removing the characters in order to keep the string a k-mer can be done using the $kLCP_{0-1}$ array as we can extend to both directions as long as the value of the $kLCP_{0-1}$ is 1 [Sal17]. We check whether this k-mer is represented, if so, delete the k-mer, extend the string, and continue. If no extension, we proceed with the next not-yet-represented k-mer in the SA.

As each position is visited at most once per deletion and once when scanning for k-mers, the complexity of the algorithm is $\mathcal{O}(|S| + N4^{d_{\max}})$ where N is the number of represented k-mers. This is linear for constant values of $d_{\max}$.

In the bi-directional model, we would need to locate the reverse complement of each k-mer, which would worsen the time complexity by a factor of k. We leave it as an open question whether the same time complexity as in the uni-directional can be obtained in the bi-directional model as well. However, we note that for practical usage, the uni-directional algorithm is usable also in the bi-directional model as only the canonical k-mer from the pair can be stored and queried.