Master's Degree programme
in **Computer Science and Information Technology**

Final Thesis

Meta-Differential compression for colored de Bruijn graphs

Supervisor
Prof. Giulio Ermanno Pibiri

Graduand
Alessio Campanelli
Matriculation Number 878170

Academic Year
2023-2024

To my parents, Tatiana and Cesare, who showed me what true love is, and have always helped me and supported all my choices. To Silvia, Mario, and all my friends and fellow university colleagues, who have accompanied me through this journey, making it the best experience of my life. To Pietro, Gianluca, Alberto, and all other professors and tutors who taught me the art of programming. To Professor Pibiri, for his dedication with which he followed me during the realization of this thesis, and thanks to whom I embarked on my research path.

Abstract

The problem of sequence identification or matching (i.e. the process of determining the origin, function, and mutations of a DNA sequence) is relevant for many important tasks in Computational Biology, such as metagenomics and pangenome analysis. Due to the complex nature of such analyses and the large scale of the reference collections, a resource-efficient solution is critical. To solve this problem, we propose a lossless compressed data structure for colored de Bruijn graphs, which can be regarded as maps from k-mers to their color sets. The color set of a k-mer is the collection of all the identifiers of the references in which that k-mer can be found. The solutions developed in this thesis exploit the repetitiveness of the color sets when indexing large collections of related genomes, extracting repeating patterns and encoding them once, instead of redundantly replicating their representation. Experimental results show that these representations substantially improve over the space effectiveness of the best previous solutions while impacting only marginally the efficiency of the queries.

The work carried out during this thesis led to a publication in the *Joural of Computational Biology* [1].

Contents

Chapter 1

Introduction

Bioinformatics is an interdisciplinary field of science that aims to develop software tools for understanding biological data, particularly when the datasets are massive. It combines not only Biology and Computer Science but also Mathematics, Statistics, Physics, and Chemistry to analyze and interpret this type of information. Thanks to the advancements made in DNA sequencing technology, this field of research has gained a lot of interest in recent years, finding an increasing number of applications as time passes, such as sequence analysis [2, 3], proteomics (the study of proteins) [4, 5], drug discovery and development [6, 7], personalized medicine [8, 9], only to name a few. All of these work with many different kinds of data, like sequences of nucleotides that form DNA and RNA strands, sequences of amino acids making up proteins, as well as full biochemical pathways.

In this thesis, we will focus on the applications working with genomic data, specifically with **pangenomes**: collections of DNA sequences belonging to a particular species, population, or closely related group [10]. When dealing with this kind of data, a key operation is to determine the set of references — also called **color set** — in which individual nucleotide sequences[1] appear. Since these target sequences are set to have a fixed length k, they are called ***k*-mers**. Such operation can be formalized into the following problem:

> **Problem 1** (Colored k-mer indexing)**.** Let $\mathcal{R} = \{R_1, ..., R_N\}$ be a collection of DNA sequences, called **references**. Each reference R_i is a string over the DNA alphabet $\Sigma = \{\mathsf{A}, \mathsf{C}, \mathsf{G}, \mathsf{T}\}$. We want to build a data structure that

[1]Nucleotides are the basic building blocks of DNA. They are composed of a sugar molecule, a phosphate group, and a nucleobase, which can be one of *adenine*, *cytosine*, *guanine*, and *thymine*. In a DNA molecule, they are paired as A-T and C-G. Since individual reads come from both strands when sequencing genomes, we consider a nucleotide sequence and its **reverse complement** equal. The reverse complement is the inverted sequence, with the bases swapped with its pair complement. For example CACTCG is the reverse complement of CGAGTG.

allows us to retrieve the set $\textsc{ColorSet}(x) = \{i | x \in R_i\}$ as efficiently as possible for any k-mer $x \in \Sigma^k$. If the k-mer x does not occur in any reference, $\textsc{ColorSet}(x) = \varnothing$.

To solve this problem efficiently, it is essential to rely on the specific properties of Problem 1. We know that consecutive k-mers share $k - 1$ long overlaps, obtained by removing the first or the last character of the k-mer, and that k-mers that co-occur in the same set of references have the same color sets. A combinatorial object that elegantly captures these properties is the **colored de Bruijn Graph**, which will be the main topic of this document.

1.1 Contributions

The goal of this thesis is to introduce two novel compressed representations for the colored de Bruijn Graph, as well as provide a comprehensive overview of the most recent and influential solutions for Problem 1, with particular interest on the specific case in which $\mathcal{R}$ is a pangenome.

To take advantage of the properties of the problem, we exploit recent developments in indexing de Bruijn graphs that split the data structure into two collaborating entities: a k-mer dictionary and the collection of the color sets. In this work, we will focus on the representation of the second one. The two new representations mentioned above are based on partitioning the color sets into patterns that repeat across the collection of all color sets. These patterns are encoded only once in the data structure, avoiding unnecessary redundancy for their representation. This is a completely new approach compared to previous methods in the literature, as many of them consider color sets individually, or encode such repeating patterns in different ways that significantly harm query speed.

1.2 Structure

We will look at the basic concepts regarding de Bruijn graphs in Chapter 3: how they are defined, built, and applied to computational biology problems. Then we will review related work and the state-of-the-art in Chapter 4. In Chapter 5 we describe our new representations for the color sets, compare them to the state-of-the-art, and present a simple framework to build these new data structures. Finally in Chapter 6 we will draw our conclusions, demonstrating the power of exploiting the repetitiveness of shared patterns, and discuss some promising future work.

Chapter 2

Preliminaries

Before starting the conversation on de Bruijn graphs and their implementations, it is important to introduce some important tools that will be used across the document.

2.1 Elias Gamma and Delta Codes

Elias δ codes [11] is a variable-length prefix code that maps positive integers into bit sequences, such that no sequence is a prefix of another.

To define them it is necessary to introduce first **unary** codes: prefix codes computed as

$$u(x) = \mathtt{0}^{x-1}.\mathtt{1}$$

for any $x > 0$. The notation $\mathtt{0}^n$ means that the character $\mathtt{0}$ repeats n times. For example, $u(1) = \mathtt{1}$, $u(2) = \mathtt{0.1}$, $u(3) = \mathtt{00.1}$, $u(4) = \mathtt{000.1}$, and so on. It is easy to see that these codes take $|u(x)| = x$ bits.

Unary codes can be used as prefixes for **gamma** (γ) codes, that are more useful for larger values of x. The γ-code of x is

$$\gamma(x) = u(|x|).[x]_{|x|-1}$$

where $[x]_\ell$ are the ℓ least significant bits of x. This means that the binary length $|x| = \lfloor \log_2(x) \rfloor$ of x is encoded in unary, followed by the binary representation of x without its highest bit. For example, $\gamma(1) = \gamma(\mathtt{1}) = \mathtt{1}$, $\gamma(2) = \gamma(\mathtt{10}) = \mathtt{01.0}$, $\gamma(2) = \gamma(\mathtt{11}) = \mathtt{01.1}$, $\gamma(4) = \gamma(\mathtt{100}) = \mathtt{001.00}$, and so on. Gamma codes take $|\gamma(x)| = 2\lfloor \log_2 x \rfloor - 1 = O(\log_2 x)$ bits.

Finally, **delta** (δ) codes can be obtained similarly to gamma codes, by using $\gamma(|x|)$ as the prefix of the representation:

$$\delta(x) = \gamma(|x|).[x]_{|x|-1} \ .$$

For example $\delta(1) = \delta(1) = 1$, $\delta(2) = \delta(10) = 010.0$, $\delta(2) = \delta(11) = 010.1$, $\delta(4) = \delta(100) = 011.00$, and so on. Delta codes are shorter than gamma codes for $x \geq 32$ and take $|\delta(x)| = |x| + 2\lfloor \log_2 |x| \rfloor = \log_2 x + O(\log_2 \log_2 x)$ bits.

2.2 Constant Time Rank and Select Queries over Bitvectors

Let $B[1..n]$ be a bitvector of length n. When working with bitvectors two of the most important operations are:

- $\textsc{Rank}_1(B, i)$, that returns the number of 1 bits in $B[1..i]$.
- $\textsc{Select}_1(B, i)$, that returns the position of the i-th 1 bit in B.

With a space overhead of just $o(n)$ bits, it is possible to execute both operations in constant time [12, 13, 14].

2.2.1 Rank

A naïve solution to perform $\textsc{Rank}$ in constant time is to simply store a vector R, such that $R[i] = \textsc{Rank}_1(B, i)$. This solution however requires $n \log_2 n$ additional bits, much more than $o(n)$.

The same approach can be improved by precomputing some tables that store parts of the answer to every query, in a way that these parts can be extracted in $O(1)$ time and the total size of the tables is $o(n)$ bits. First, we divide B into *superblocks* of size $s = (\log^2 n)/2$ and build a table $R_s[0, \lfloor n/s \rfloor]$ where $R_s[i] = \textsc{Rank}_1(B, i \cdot s)$. Then, each superblock is divided into *blocks* of size $b = (\log n)/2$ and a new table $R_b[0, \lfloor n/b \rfloor]$ is created, where $R_b[i] = \textsc{Rank}_1(B, i \cdot b) - R_s[\lfloor i \cdot b/s \rfloor]$. In other words, $R_b[i]$ is the number of 1s up to the i-th block, from the beginning of its superblock. To compute $\textsc{Rank}_1(B, i)$ we add up:

- $R_s[\lfloor i/s \rfloor]$, the number of 1s up to the start of the superblock.
- $R_b[\lfloor i/b \rfloor]$, the number of 1s from the start of the superblock up to the start of the block.
- The number of 1s from the start of the block to i.

The last value can be computed in constant time by precomputing a third table R_p with the ranks of all possible blocks of b bits. Since the size of all three tables is $o(n)$, we need $R_s + R_b + R_p = o(n)$ extra bits to perform $\textsc{Rank}$ queries in constant time.

2.2.2 Select

A similar partitioning approach can be applied to SELECT operations to reach constant time execution with $o(n)$ extra bits.

First, note that RANK can be used to perform a binary search of i: if $\text{RANK}_1(B, n/2) > i$, then the result of the query is in the first half of the bitvector, otherwise, it is in the second half.

As for RANK, faster queries can be obtained by precomputing some of the answers. This can be done by splitting the bitvector into blocks of $s = \log_2^2 n$ bits; if a block is larger than $s \log_2^2 n$ bits then it is considered *long* and the result for each SELECT inside the block is stored in an array, otherwise, it is *short* and the result can be found using binary search. Note that if a block is long then the space needed to store the SELECT answers is small compared to its length, whereas if it is short the binary search will be fast. Short blocks can be further partitioned into *miniblocks* that in turn can be short or long. The answers to long miniblocks can be stored as before, while a precomputed table of results can be used for short ones.

2.3 Elias-Fano Codes

The Elias-Fano [15, 16] representation is a simple way to represent monotone non-decreasing sequences in a very compact way, supporting constant time random access and search operations.

Let A be a non-decreasing sequence of n integers from the universe $[0, ..., m)$. Each integer is first encoded in binary using $\lceil \log_2 m \rceil$ bits. Then, the binary representation is split into two parts: the higher part consisting of the first $\lceil \log_2 n \rceil$ bits and the lower part with the remaining $\ell = \lceil \log_2 m \rceil - \lceil \log_2 n \rceil \leq \lceil \log_2(m/n) \rceil$ bits. The lower parts of all elements are concatenated in the order they appear and explicitly stored in a bitvector of ℓn bits. The higher part is constructed starting from a bitvector containing n 0s, one for each integer that can be represented with $\lceil \log_2 n \rceil$ bits. Then, before the i-th 0 are added as many 0s as the number of times i appears as the high part of an integer. Since there are n 0s at the start and n 1s are added (one per element in the list), the high part takes $2n$ bits, for a total space of $2n + \ell n$ bits.

Example Let $A = \{2, 3, 5, 7, 11, 13, 24\}$, so $n = |A| = 7$ and $m = 24 + 1 = 25$ (as m is not included in the universe). The binary representation of each integer is $\lceil \log_2 25 \rceil = 5$ bits long, of which the high part takes $\lceil \log_2 7 \rceil = 3$ bits. Thus, A can be represented as

A	High bits	Low bits
2	000	10
3	000	11
5	001	01
7	001	11
11	010	11
13	011	01
24	110	00

The low bits are concatenated into

10.11.01.11.11.01.00

and the high bits are stored as the following bitvector:

110.110.10.10.0.0.10.0

To query the i-th number, where $i \in [1, n]$ the high and low parts must be retrieved separately. Getting the low part is trivial, as it consists of reading ℓ bits starting from the bit in position $(i-1) \cdot \ell$, as all the elements have the same length. The high part can be computed by counting the number of 0s before the i-th 1 in the high part bitvector. This reduces to executing $j = \textsc{Select}_1(i) - i$. The high part is thus the binary representation of j using $\lceil \log_2 n \rceil$ bits.

Since both operations can be executed in constant time, as explained in the previous section, we can state that random access using Elias-Fano encoding takes $O(1)$ time.

Example To read the number at position $i = 5$, first we take the low part

10.11.01.11.**11**.01.00

then we compute the high part as:

110.110.**1**0.10.0.0.10.0

$$\textsc{Select}_1(5) - 5 = 7 - 5 = 2.$$

Since 2 can be expressed in binary as 010 using $\lceil \log_2 7 \rceil = 3$ bits, the result is

$$010.11_2 = 11_{10}$$

Chapter 3

Colored de Bruijn Graphs

The process of determining the exact sequence of nucleotides in a DNA molecule is called *sequencing*. Sanger et al. [17] developed the first widespread sequencing technique in 1977. Said technique works by generating multiple copies of the target DNA, terminating the duplication reaction with special markers at random points to create strands of different lengths. These are then sorted by their length and their tail marker is identified using radioactive or fluorescent labeling. In this way, it is possible to decode very long DNA sequences with an exceptional accuracy of 99.99%, thus not requiring complex data analysis.

Despite its advantages, Sanger sequencing has two main drawbacks: it is a slow and costly process. For this reason, Next Generation Sequencing (NGS) technologies, such as Illumina [18], PacBio [19], and Nanopore [20], have been developed in recent years. These adopt a very different design with respect to Sanger method, based on massive parallel processing techniques: instead of processing one DNA fragment at a time, NGS technologies read millions of DNA fragments simultaneously, offering a much higher throughput at the cost of worse accuracy and shorter sequences. Because of this, NGS methods require highly efficient assembly algorithms to reconstruct the full DNA sequence and correct any errors the process might introduce, as well as compact data structures to store the results.

These motivations led to the rise in popularity of the de Bruijn Graph data structure, which is now widely used to address the problems of genome assembly, correction, and storage [21, 22, 23, 24]. In the following sections, we will see why de Bruijn Graphs became so important in the field of computational biology. We will focus on how their definition, properties, applications, and which software tools can be used to build them.

3.1 Definitions

A de Bruijn Graph (dBG for short) is a special type of directed graph, named after the mathematician Nicolaas Govert de Bruijn, who first described it.

Definition 1 (de Bruijn Graph)**.** Let Σ be an alphabet, and $\mathcal{K} = \left\{x \in \Sigma^k\right\}$ be a set of k-mers. The **de Bruijn Graph** of $\mathcal{K}$ is a directed graph $G(\mathcal{K}, E)$ whose nodes are the k-mers in $\mathcal{K}$ and edges $(u, v) \in E$ connect two nodes if and only if the last $k-1$ symbols of the source u are equal to the first $k-1$ symbols of the destination v (Figures 1a, 1b).

Note that the edge set E is implicitly defined by the set of nodes, and can therefore be omitted from subsequent definitions. Likewise, a path in the dBG spells the string obtained by concatenating all k-mers along said path, without repeating the overlaps. In particular, unary (i.e. non-branching) paths can be collapsed into single nodes spelling strings referred to as **unitigs**, as it can be seen in Figure 1c. The dBG arising from this step is called **compacted** dBG and is indicated with $G(\mathcal{U})$, where $\mathcal{U} = \{u_1, \ldots, u_m\}$ is the set of unitigs. Henceforth, whenever a dBG is mentioned, we assume it is already compacted unless otherwise specified. Note that, if $|\mathcal{K}| = n$, it holds that $m \leq n$. In practice, the number of unitigs is much smaller than the number of k-mers.

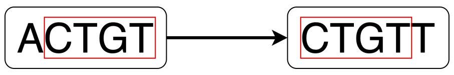
It's easy to see that, by setting $\Sigma = \{\mathsf{A}, \mathsf{C}, \mathsf{G}, \mathsf{T}\}$ and building the sets $\mathcal{K}$ and $\mathcal{U}$ starting from a DNA reference R, the dBG is a very suited data structure for performing sequence analysis operations and a variety of other applications.

De Bruijn Graphs can also be augmented by storing various types of additional data, to perform more complex tasks or to produce better results on simpler operations. Some types of metadata are:

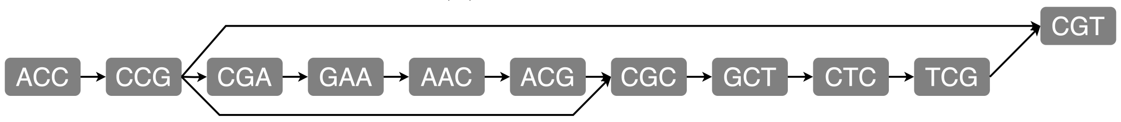
- **abundances**: the number of times k-mers appear in one or multiple genomes
- **positions**: where each k-mer appears in the genomes
- **colors**: given a set of references, in which subset of them each k-mer appears

Among these, we will focus our attention on the last modification.

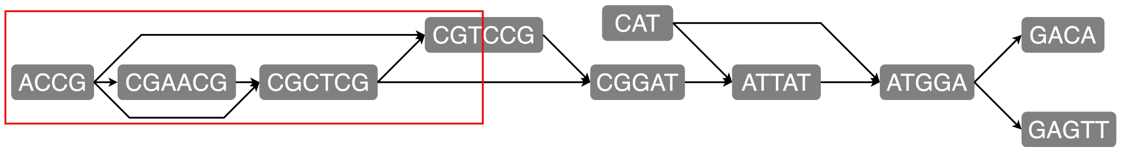
Definition 2 (colored de Bruijn Graph)**.** Let $\mathcal{R}$ be the collection of the references $\{R_1, ..., R_N\}$. We will use the terms reference and **color** interchangeably. A colored de Bruijn Graph (c-dBG) is a dBG that stores, for each k-mer x, the value $\textsc{ColorSet}(x)$, which is the set $\{i | x \in R_i\}$ of all references where the k-mer x is present.



(a) Edge in a dBG



(b) An example dBG



(c) A compacted dBG

Figure 1: In panel (a), an edge in a dBG for $k = 5$. Notice that the end of the source node is equal to the start of the destination node. In panel (b), an example dBG for $k = 3$. Panel (c) represents a dBG collapsed by using unitigs. The part highlighted in red is the graph in panel (b).

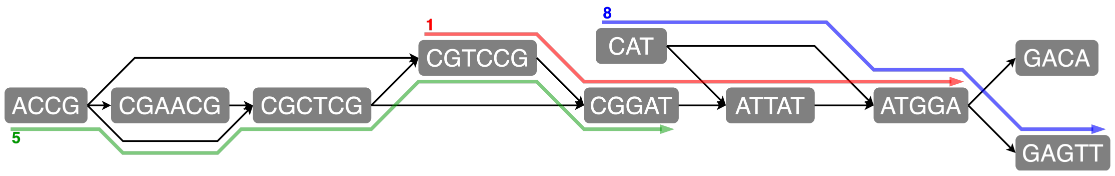


Figure 2: An example c-dBG for $k = 3$, with three colors highlighted (1, 5, 8)

The compaction step performed on the classical dBG can also be applied to the c-dBG. This time, however, only non-branching paths with nodes having the same color set are collapsed into unitigs. We refer to a compacted c-dBG as $G(\mathcal{U}, \mathcal{C})$. Figure 2 shows the same dBG as Figure 1b with some of the color sets highlighted.

From now on, we will focus only on the compacted variants of the dBG.

Table 1 summarizes all the notation introduced in this Section that will be used throughout this document.

3.2 Properties

Unitigs in a c-dBG have the following properties:

1. **Unitigs spell references in $\mathcal{R}$.** Each distinct k-mer of $\mathcal{R}$ appears once, as a substring of some unitig. By construction, each reference $R_i \in \mathcal{R}$ can be spelled out by some concatenation of the unitigs (accounting for the

Symbol	Meaning
$\mathcal{K}$	set of all k-mers
$\mathcal{U}$	set of all unitigs
$\mathcal{R}$	set of all references
$\mathcal{C}$	set of all color sets
$G(\mathcal{U})$	compacted de Bruijn Graph
$G(\mathcal{U}, \mathcal{C})$	compacted colored de Bruijn Graph
n	number of distinct k-mers
m	number of unitigs
N	number of colors
z	number of distinct color sets

Table 1: dBG essential notation

overlaps). Joining k-mers into unitigs reduces their storage requirements, as fewer overlaps need to be saved, and accelerates the lookup of k-mers in consecutive order.

2. **Unitigs are monochromatic**. The k-mers belonging to the same unitig share the same color set. We write $x \in u_i$ to indicate that k-mer x is a substring of unitig u_i. This means that $\textsc{ColorSet}(u_i)$ denotes the color set of each k-mer $x \in u_i$, thus a single color set should be represented for each unitig rather than for each k-mer, further reducing the c-dBG size.

3. **Unitigs co-occur**. Distinct unitigs often have the same color set, meaning that they appear in the same set of references. We indicate with z the number of distinct color sets $\mathcal{C} = \{C_1, ..., C_z\}$. Note that $z \leq m$ and that, in practice, there are always many more unitigs than color sets.

It follows that exploiting all three properties could be highly beneficial to query efficiency and space effectiveness.

3.3 Construction

In the last decade, many algorithms for the construction of compacted dBGs were developed. Computing the unitigs might seem trivial, as it can be performed with a linear time algorithm: first, all k-mers are inserted in a hash table, where they are mapped to all of their successor and predecessor characters. Then, each k-mer is extended left or right if and only if there is exactly one predecessor or successor, respectively. Despite the algorithm's simplicity, the problem's practical difficulty comes from the potential size of the input reads, which may not fit in the machine's main memory. For this reason, efficient dBG compaction algorithms are heavily engineered to use external memory and multithreading.

We will now present some of the most recent and influential solutions to this problem.

3.3.1 TwoPaCO

TwoPaCo [25] is a scalable, low-memory algorithm for the direct construction of compacted dBG from a set of complete genomes.

It works by splitting the k-mers into ℓ partitions and searching for *junctions*, nodes in the graph that have an in-degree or an out-degree greater than one, or k-mers that are the start or the end of an input string. Each round considers one of the partitions and proceeds as follows. It starts by marking each node as a junction, using a bit-vector, and then performs two passes to unmark non-junction nodes. The first pass uses a Bloom filter [26] to store the edges of candidate junction k-mers and unmarks nodes that do not satisfy the definition of junction.

Definition 3 (Bloom filter)**.** A Bloom filter (BF) is an approximate data structure representing a set $\mathcal{S}$, supporting insertion and membership operations. It guarantees a bounded one-sided error on membership queries: if $x \in \mathcal{S}$, the query returns YES with probability 1, otherwise if $x \notin \mathcal{S}$ it returns NO with probability $1 - p$, for some user-defined parameter $p \in [0, 1]$.

It is implemented as a bitvector $B[0, w]$ of length w, together with t independent and uniform hash functions $h_1, ..., h_t$, mapping values to $[0..w]$. The length w is computed based on the maximum capacity of the filter and false positive rate p. To insert a value x, we set $B[h_i(x)] \leftarrow 1$ for all $i = 1, ..., t$. To answer a membership query for the value x, we return YES if and only if $\bigwedge_{i=1}^{k} B[h_i(x)]$ is 1.

Bloom filters take very little space, but can generate false positives on membership queries. This does not affect the correctness of the TwoPaCo algorithm, as all correct junctions will never be unmarked, but leaves some non-junction nodes marked. A second pass is thus performed using a hash table to eliminate the excess junctions, which will take considerably less space than it would have done in a single pass, thanks to the filtering of the first one.

This algorithm is highly parallelizable, as each split can be processed simultaneously, and each pass can be made multithreaded by using data structures that support concurrent writes.

3.3.2 BCALM2

BCALM2 [27] is a parallel algorithm that distributes the input based on a minimizer hashing technique. **Minimizers** [28, 29] are substrings of k-mers of fixed length $g < k$ — essentially g-mers of k-mers — that are increasingly being used in various fields of Information Retrieval, such as text or genomic data search. These g-mers are used as a compacted representation of sequences, and they are chosen so that they *minimize* a certain value, be it the lexicographic order or the result of a hash function.

BCALM2 takes in input a set of k-mers and compacts them in three stages. First, k-mers are distributed into one or two buckets each, then each bucket is compacted separately, and finally k-mers that were put into two buckets are glued together. In the distribution phase, the left and right minimizers of each k-mer are computed using multiple threads. The left minimizer is the minimizer of the first $k-1$ characters of the k-mer; conversely, the right minimizer only considers its last $k-1$ characters. Then, k-mers are put inside the buckets (files on the disk) corresponding to the minimizers' identifiers; k-mers having the same left and right minimizers are written on a single bucket. The second step follows, where buckets are concurrently compacted, such that if two k-mers share exactly one edge, they are joined in a single string. In the last step, strings in different buckets that share a k-mer on opposite ends are glued together using a union-find set. The final result corresponds to the set of unitigs of the dBG.

3.3.3 Cuttlefish2

Cuttlefish2 [30] is a dBG construction algorithm applicable on raw sequencing of short reads and assembled references, based on the modeling of vertices as Deterministic Finite Automata (DFA).

It works by first enumerating the set of edges of the graph, from which the set of vertices is extracted. Next, a Minimal Perfect Hash Function (MPHF) [31] over the vertices is constructed, mapping vertices to $[1, n]$. The MPHF is a space-efficient way to associate information to the vertices. Each vertex is then modeled as a DFA, where its state is the number of left/right neighbors (zero, one, or more than one). In this way, just enough information to build the unitigs is stored per vertex. With a final traversal on the vertices and the DFA states, k-mers are stitched together if the current k-mer does not branch forward and if the following k-mer does not branch backward, constructing the maximal unitigs.

Cuttlefish2 performs many I/O operations to store the intermediate results to disk to keep the working memory low. Like BCALM2 it does not compress data on disk, further increasing the time required by writing and reading from the disk.

3.3.4 GGCAT

GGCAT [32] is the latest and fastest algorithm for compacting dBGs, achieving great speedups compared to BCALM2 and Cuttlefish2.

Contrary to its predecessors, GGCAT merges the k-mer counting step with unitig construction, by computing them inside each bucket. In this way, not all k-mers have to be stored, as only the unitigs are written in a compressed format to disk, further reducing the space requirements. It works by splitting references into substrings of consecutive k-mers that share the same minimizer and keeping track of the characters immediately preceding and succeeding the substring (called linking characters). Then, each substring is stored in a *group* based on its minimizer. For each group, a list of unique k-mers is computed, and for each unused k-mer x a new string $z := x$ is initialized. z will be extended left or right as long as it is a unitig: for the first and last $(k-1)$-mer each possible extension (either A, C, G or T) is looked for inside the group, using a hashmap. If exactly one match y is found, y is marked as used, z is extended left or right, respectively, and the process is repeated until a match is not unique or a linking character is found, in both directions. In the former case, the unitig is considered complete, otherwise in the latter, intermediate unitigs from different groups are merged together into maximal unitigs, based on their left and right ending k-mers.

In this way, GGCAT manages to outspeed all other compaction algorithms being $3\times$ to $21\times$ faster than Cuttlefish2.

3.4 Layout

To solve Problem 1, the goal is to implement the map $x \rightarrow \textsc{ColorSet}(x)$ as efficiently as possible for any k-mer x, in terms of both memory usage and query time. The efficiency of any solution using a c-dBG is directly related to its encoding.

As many information retrieval problems, also Problem 1 can be solved using an **inverted index** [33], a data structure that stores the association between terms (k-mers) and the sorted lists of the identifiers of the documents that contain such terms (color sets). These sorted lists are called **inverted list** and in the context of this problem, each $\textsc{ColorSet}(x) \in \mathcal{C}$ is the inverted list of the k-mer x.

Let $\mathcal{L}$ be the inverted index for $\mathcal{R}$. $\mathcal{L}$ stores the $\textsc{ColorSet}(x)$ for each k-mer $x \in \mathcal{R}$. In order to implement the map from k-mers to color sets efficiently, all the distinct k-mers are stored losslessly in a dictionary $\mathcal{D}$. To be useful to this problem, $\mathcal{D}$ must be *associative*, meaning that it has to support the operation $\textsc{Lookup}(x)$, which returns $\perp$ if k-mer $x \notin \mathcal{D}$ or a unique identifier in $[n] = \{1, ..., n\}$ otherwise, where $n = |\mathcal{K}|$ is the number of distinct k-mers in $\mathcal{R}$.

Problem 1 can thus be solved thanks to the interplay between $\textsc{Lookup}(x)$ and

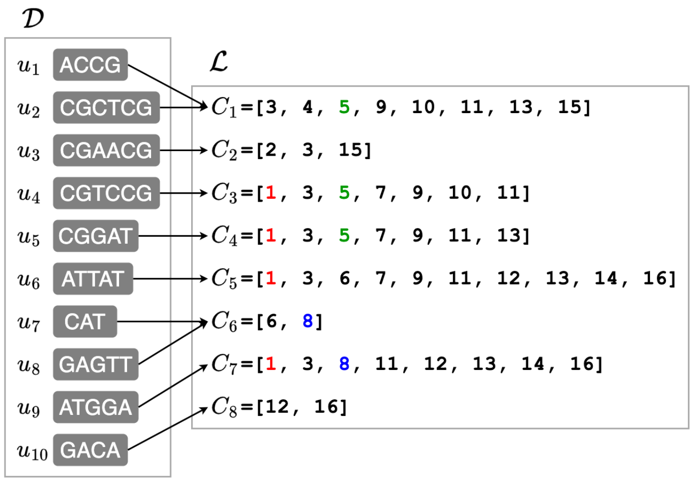


Figure 3: A representation of the modular index layout, assuming the c-dBG was built from $N = 16$ references, showing the dictionary $\mathcal{D}$ and the inverted index $\mathcal{L}$. Note that unitigs mapping to the same color set are consecutive in the dictionary.

ColorSet(x): the index stores the color sets in some compressed form, sorted by the value returned by Lookup(x).

See Figure 3 for a schematic index representation.

3.5 Applications

With the advancements in NGS techniques, **genome assembly** [21] became one of the most important tasks in genome biology. Its objective is to obtain a complete genome sequence given a set of DNA fragments, called *reads*, generated by the sequencing technologies. The algorithm used in conjunction with Sanger sequencing, the Overlap-Layout-Consensus, is too complex, as it requires solving a Hamiltonian path problem — known to be NP-hard — followed by an alignment step. On the other hand, when using a dBG, the same problem becomes much easier. The first step is splitting the reads into k-mers and building a dBG with them. After that, assembling a complete genome requires finding a path that traverses each edge exactly once, thus considering all possible overlaps. This is the well-known Eulerian path problem, solvable with a polynomial time algorithm. Moreover, given the dBG overlapping properties, the alignment step can be skipped completely, as it is intrinsic in the graph construction.

Another operation that can be easily performed using a dBG is **read correction** [34]. Sequencing technology is not perfect, so it is possible for DNA reads to have some errors, like missing or misread nucleotides. To prevent these inaccuracies from impacting other operations, many dBG construction algorithms filter out or correct k-mers that are considered wrong. This is mostly done using a consensus method: all k-mers that appear less than a set threshold are removed or corrected, as k-mers generated by sequencing errors appear much rarer than correct ones.

A fundamental challenge of NGS techniques is the alignment of sequencing reads to one or more reference genomes. **Read alignment** is among the most resource-consuming steps of high throughput sequencing analysis, and thanks to the rapid increase of available full DNA sequences, many new applications require aligning reads to one or more reference genomes. For example, species and pathogens can be identified or characterized by aligning reads to some reference genomes [35]. Similarly, it can be used in microbiome research to determine the composition of microbial communities [36]. If the reference genome is missing, the alignment is called **de novo assembly** [37, 38], a process in which k-mers are joined without prior knowledge of the correct sequence or order. Most state-of-the-art generic aligners are based on a *seed-and-extend* approach, which works in two steps:

1. Seeding: the k-mers of the reads to align are chosen as starting points of the research.

2. Extension: the reads are aligned to the regions surrounding each seed, to determine the most likely read positions.

Seeding, despite accelerating the process, induces a trade-off between speed and accuracy: the generation of a large number of seeds will most likely yield the best alignment, but using an exact alignment algorithm will take a significant amount of time to compute. On the other hand, using fewer seeds, the alignment process will be faster, but with a higher probability of missing the target [39].

To avoid this compromise, newer approaches focus on the **pseudoalignment** of reads. Contrary to classical alignment, the result of a pseudoalignment operation is a color set *without* specific coordinates mapping each base in the read to particular positions in each of the transcripts [40]. In other words, it reports only whether a read matches a reference sequence or not, without necessarily returning the genomic coordinates of the match [41]. Pseudoalignment algorithms mainly fall into two categories: exhaustive methods, that retrieve the color set of every k-mer on a given read, or skipping heuristics, that can jump over k-mers that are likely to be uninformative.
As the definition of pseudoalignment is essentially the same as Problem 1, when discussing and evaluating the querying capabilities of the indexes described in Chapters 4 and 5, we will focus on this operation.

One important use of alignment techniques is **pangenome analysis**, the study of the complete set of genes belonging to the same species, rather than focusing on a single individual. Pangenomes have revolutionized DNA analysis by providing a more comprehensive understanding of genetic diversity within a species. This is particularly valuable as it can express a wide range of genetic variations, including rare and unique sequences that may be absent from a particular reference genome.

When performing this kind of analysis, the first step is aligning all genes with each other. Then, the aligned sequences are compared to find similarities and discrepancies. In this way, it is possible to find useful genomic data of a species, like *core genomes* — the set of genes that are present in all individuals of the species — and *dispensable genomes* — made of genes shared by only a subset of the strains. While the first ones are typically involved in essential cellular functions, the others often contribute to multiple variations, such as antibiotic resistance, environment adaptations, etc. To perform pangenome analysis, it is not sufficient to rely on color sets alone, but it is also necessary to store the position of each k-mer relative to its reference. Since the main topic of this work are colored de Bruijn graphs, we leave the discussion on positional dBGs as a possibility for future work.

Chapter 4

Related Work

In the last few years, many solutions based on c-dBGs have been proposed to solve the colored k-mer indexing problem (Problem 1). While all of them share the same dictionary and inverted lists structure, their approaches to encoding these two data structures are very different. However, in the majority of the cases, none exploits the unitig properties described in Section 3.2

In this chapter, we will describe and compare some of these solutions, with particular attention to whether or not they follow the three c-dBG properties, their space effectiveness, and query efficiency.

4.1 Mantis

Mantis [42, 43] is a de Bruijn Graph index based on Counting Quotient Filters [44] (CQF). A CQF is a compact representation of a multiset — in the same way a Bloom Filter is a compact representation of a set — that can answer counting queries with a one-sided error, i.e. the count returned by a CQF is never smaller than the correct one.

Mantis follows the modular layout described in Section 3.4, where the CQF is the dictionary mapping k-mers to color sets, and the inverted index is a table of bitvectors. In particular, the CQF is repurposed to store not how many times a k-mer is repeated inside the input, but to store its color set identifier. In other words, to map a k-mer x to the color set with id c, x has to be inserted c times in the CQF (this operation can be done with a single instruction).
In its first implementation, the bitvectors encoding the color sets were compressed using RRR compression [45], but in its most space-efficient variant, the color sets are expressed differentially as edits performed on the branches of an approximate minimum spanning tree over the color sets.

Regarding the properties of the c-dBG, Properties 1 and 2 are not exploited, as the individual k-mers are stored and mapped to their color sets through the CQF. Property 3, is followed, as color sets are stored only once in the inverted index. However, looking up consecutive k-mers has no locality due to the employment of hash functions in the CQF.

4.2 COBS

COBS [46] is an inverted index designed for compressing the k-mers of DNA samples. Additionally, it works with any text document and allows for approximate pattern-matching queries, achieved through the usage of Bloom filters.

Internally, COBS is just a collection of $|\mathcal{R}| = N$ Bloom filters, where the i-th one represents the approximate membership of the k-mers to R_i. However, since the false positive probability depends on the reference size, where larger references lead to denser — and thus more error-prone — bitvectors, the COBS index adapts the size of the BFs to the references to keep the false positive rate constant. This is obtained by sorting and partitioning the input collections into shards of approximately the same size to build sub-indices of different sizes.

COBS does not exploit any specific property among the ones stated in the previous section, so it is not optimized for any particular dataset. Unitigs are broken into their constituent k-mers and indexed separately, and because of the usage of Bloom filters, COBS suffers the same locality problem of Mantis.

4.3 Bifrost

Bifrost [47] is a c-dBG implementation that offers a broad range of functions, such as indexing, editing, and querying. The dictionary $\mathcal{D}$ comprises a unitig array and a dynamic hash table, mapping the minimizers of the k-mers to their positions in the unitigs. Color sets are stored for each unitig u in a $|u| \times N$ binary matrix, where rows represent k-mers and columns represent colors. To limit their memory usage, these matrices are stored in different compressed indexes based on their sparseness: 64-bit words for tuples or very small matrices, Roaring bitmaps [48] otherwise.

Only the first property is utilized among the previously mentioned ones, as unitigs are stored in the unitig array. The other two are ignored because of the way colors are stored: the color set of each k-mer is stored individually in the bit matrix, possibly being repeated multiple times. It is also immediate to see that, if unitigs were monochromatic, a simple array would suffice to store the colors.

4.4 MetaGraph

MetaGraph [49] provides several data structures for storing k-mer sets ($\mathcal{D}$) and many general schemes to compress metadata associated with k-mers ($\mathcal{L}$). Among the different dictionary structures, the default one is the BOSS table [50], a data structure very similar to an XBWT [51], that is an extension of the Burrows-Wheeler transform [52]. Colors are represented in an $n \times N$ matrix, that can be compressed in multiple ways, row or column-wise. The best-performing inverted index uses a *Multi-BRWT* representation, an n-ary Wavelet Tree [53] that allows vertical splitting of matrices into more than two sub-matrices.

Being based on a BWT, the BOSS data structure does not exploit Property 1, since the k-mers are arranged in *colexicographical* order and not in the order of appearance in the unitigs.

4.5 Themisto

Themisto [41] is one of the latest proposed indices to represent c-dBGs.

The index structure is divided into two parts, as depicted in Figure 3. The set of k-mers $\mathcal{D}$ is stored in a Spectral Burrows-Wheeler Transform [54] (SBWT), a variant of the BOSS data structure, while the inverted index $\mathcal{L}$ stores the color sets of some “key k-mers” using different encodings based on the sparseness of the color sets (i.e. the ratio $|C_i|/N$). In particular:

- *sparse* sets are stored as sorted sequences of color identifiers, where each identifier uses $\lceil \log_2 z \rceil$ bits;
- *dense* sets are encoded with bit maps of length z, where a 1 in position i denoted the presence of color i, or its absence otherwise.

As for the BOSS data structure, the SBWT disregards Property 1. This translates to an overhead of $\log_2(z)$ bits per key k-mer to associate its color set. Note however that this requires dedicated storage per-k-mer, thus also failing to exploit Property 2.

Table 2 visually summarizes the properties of the indexes discussed in this section.

4.6 State of the Art

To the best of our knowledge, the only solution that exploits all three properties is the Fulgor index. It first maps k-mers to unitigs in a dictionary, implemented

Index	Property 1	Property 2	Property 3
Mantis			✓
COBS			
Bifrost	✓		
MetaGraph			✓
Themisto			✓
Fulgor	✓	✓	✓

Table 2: Comparison of the c-dBG implementations properties. Fulgor will be described in Section 4.6

using SSHash, then maps unitigs to their color sets, compressing them using a density-based approach.

4.6.1 Dictionary: Sparse and Skew Hashing (SSHash)

SSHash [55] is a dictionary specifically tailored for k-mers, based on minimizers. In particular, it exploits two statistical properties of k-mer minimizers, precisely those of being **sparse** and **skewly distributed** (hence the name).

Given a set $\mathcal{K}$ of all distinct k-mers from a large DNA string (or collection of strings), it supports two operations:

- $\textsc{Lookup}(x)$, that returns an unique integer $0 \le i < n$ if $x \in \mathcal{K}$ or $i = -1$ otherwise;
- $\textsc{Access}(i)$, that extracts the k-mer x for which $\textsc{Lookup}(x) = i$.

SSHash is also optimized for streaming queries, which are $\textsc{Lookup}$ queries for multiple consecutive k-mers. As stated in Section 3.1, the set of $\mathcal{K}$ can be compacted into a set of $\mathcal{U}$, from which it is possible to construct the compacted cdBG $G(\mathcal{U})$. The strings in $\mathcal{U}$ form the natural basis for a space-efficient dictionary, as $|\mathcal{U}| \le |\mathcal{K}|$, and because we are guaranteed that there are no duplicate k-mers in $\mathcal{U}$.

Given a k-mer x, an integer $g \le k$, and a total order relation on all g-mers, the first step is to compute the minimizer of x. Typically — for efficiency reasons — the total order is given by a (pseudo) random hash function h. In other words, the minimizer of x is its substring of length g that minimizes the value of h.

The popularity of minimizers in sequence analysis is given by the fact that consecutive k-mers tend to have the same minimizer. This means there are far fewer minimizers than k-mers if g is not too small. Given a string of any length, be it a path in the dBG or a query sequence, a **super-k-mer** is a maximal sequence of consecutive k-mers sharing the same minimizer. There are approximately $(k -$

$g + 2)/2$ times less super-k-mers than k-mers. This means that a super-k-mer of length K is a space-efficient representation of its constituent $K - k + 1$ k-mers since it takes $2K/(K - k + 1)$ bits/k-mer.

The dictionary data structure has the following layout:

- The input strings (the unitigs) are written one after the other in a vector, using $\lceil \log_2 \Sigma \rceil = 2$ bits per base.

- We also store the endpoints of the strings to avoid the detection of alien k-mers in a sorted integer sequence of length m. This sequence is compressed using Elias-Fano and takes $m \lceil \log_2 (\sum_i^m |u_i|/m) \rceil + 2m + o(m)$ bits.

- Let $\mathcal{Z}$ be the set of all minimizers and K the number of super-k-mers. Since a minimizer can appear more than once, we have that $|\mathcal{Z}| \leq K$. Given a minimizer r, B_r is the **bucket** of the minimizer r, i.e. the set of all super-k-mers with the same minimizer r.

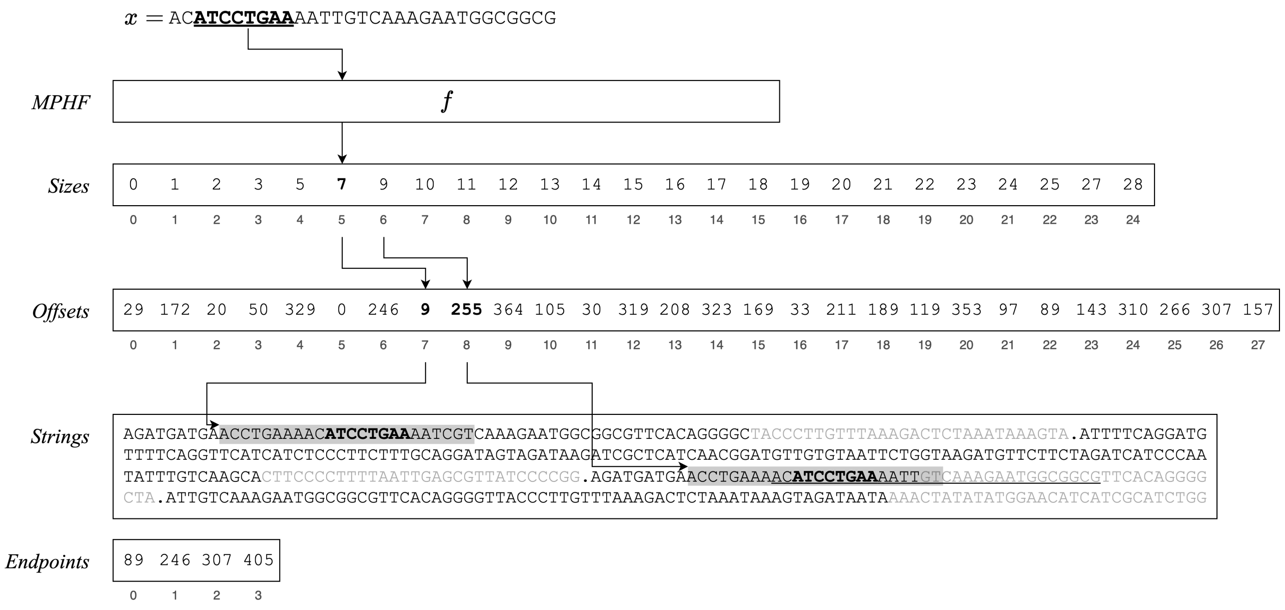
  We build a **minimal perfect hash function** (MPHF) f for $\mathcal{Z}$, for which $f(r) \in [0, |\mathcal{Z}|)$: given a minimizer r, the function returns a unique integer, representing its 'bucket identifier'. We then keep an array $\mathit{Sizes}[0, |\mathcal{Z}| + 1)$, where $\mathit{Sizes}[f(r)]$ is the number of super-k-mers before bucket B_r (i.e. the array contains the prefix-sums of the sizes of the buckets). The *Sizes* array is also compressed using Elias-Fano.

- The absolute offsets of the super-k-mers, used to identify them, are stored in an array $\mathit{Offsets}[0, K)$, in the order given by f. For a minimizer r such that $\mathit{Sizes}[f(r)] = s$, its $|B_r|$ offsets are written consecutively, in order, in $\mathit{Offsets}[s, s + |B_r|)$.

Figure 4 shows the different components just described.

Remember that a k-mer and its reverse complement are considered to be identical. This means that if a k-mer x is not found by the LOOKUP algorithm, there is still the possibility to find its reverse complement $\hat{x}$. Therefore, the algorithm should search first for x and then for $\hat{x}$, doubling the query time in the worst case. To overcome this issue, a different minimizer computation is used to make sure that only one bucket is inspected: select the minimum between the minimizer of x and $\hat{x}$. In this way, it is guaranteed that a k-mer and its reverse complement belong to the same bucket. This solution leads to a greater number of distinct minimizers used, thus a higher space usage but faster query time.

Example (Figure 4) In this example, the input of the dictionary is made up of $N = 4$ strings, visually separated by a '.' for the sake of clarity. In total, there are



x = ACATCCTGAAAATTGTCAAAGAATGGCGGCG
MPHF
f
Sizes
0 1 2 3 5 7 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 27 28
Offsets
29 172 20 50 329 0 246 9 255 364 105 30 319 208 323 169 33 211 189 119 353 97 89 143 310 266 307 157
Strings
AGATGATGAACCTGAAAACATCCTGAAAATCGTCAAAGAATGGCGGCGTTCACAGGGGCTACCCTTGTTTAAAGACTCTAAATAAAGTA.ATTTTCAGGATG
TTTTCAGGTTCATCATCTCCCTTCTTTGCAGGATAGTAGATAAGATCGCTCATCAACGGATGTTGTGTAATTCTGGTAAGATGTTCTTCTAGATCATCCCAA
TATTTGTCAAGCACTTCCCCTTTTAATTGAGCGTTATCCCCGG.AGATGATGAACCTGAAAACATCCTGAAAATTGTCAAAGAATGGCGGCGTTCACAGGGG
CTA.ATTGTCAAAGAATGGCGGCGTTCACAGGGGTTACCCTTGTTTAAAGACTCTAAATAAAGTAGATAATAAAACTATATATGGAACATCATCGCATCTGG
Endpoints
89 246 307 405

Figure 4: A schematic representation of an SSHash dictionary, for $k = 31$ and $g = 8$. Reproduced, with permission, from [55].

405 bases and $405 - N(k-1) = 405 - 4 \cdot (31-1) = 285$ k-mers for $k = 31$, since the last $k-1$ bases of a string (grey characters) are not addressable as they are too short to form a k-mer. There are $|\mathcal{Z}| = 24$ minimizers for $g = 8$ and $K = 28$ super-k-mers, thus the *Sizes* array has a length of $|\mathcal{Z}| + 1 = 25$ and *Offsets* has a size of $K = 28$.

The figure also shows the lookup for a k-mer x, whose minimizer r is highlighted in bold font. The function f returns the identifier of r as $f(r) = 5$. Knowing this, the bucket size is computed: $|B_r| = e - s = \mathit{Sizes}[5+1] - \mathit{Sizes}[5] = 9 - 7 = 2$, meaning that there are two super-k-mers to consider. The offsets of those two super-k-mers are $\mathit{Offsets}[s] = \mathit{Offsets}[7] = 9$ and $\mathit{Offsets}[s+1] = \mathit{Offsets}[8] = 255$ respectively. To find the k-mer x we must scan the 2 super-k-mers starting at $\mathit{Strings}[9]$ and $\mathit{Strings}[255]$. The k-mer x is ultimately found in the second super-k-mer at position $w = 8$, so starting at the character with offset $255+8 = 263$. Since we are in the third string, we have to remove the non-addressable characters of the first two strings to obtain the identifier of the k-mer, that is $263-(31-1)\cdot(3-1) = 263 - 60 = 203$.

The efficiency of the Lookup query depends on the size $|B_r|$ of the bucket of a minimizer. For this reason, it would be best if the number of super-k-mers inspected was a constant amount. This can be done by exploiting the **skew** property of k-mers: most minimizers appear just once, and relatively few repeat many times.

To do so, two fixed quantities are introduced, ℓ and L, with $0 \leq \ell < L$. We have that the number of buckets whose size is larger than 2^ℓ is small for a proper choice of ℓ. Also, assume, without loss of generality, that 2^L is smaller or equal to the largest bucket size. For each $i \in [\ell, L]$, let $\mathcal{S}_i$ be the set of all k-mers belonging to buckets of size $s \in [2^i, 2^{i+1})$. For each set $\mathcal{S}_i$ we build an MPHF f_i. Given a k-mer $x \in \mathcal{S}_i$, we can store the identifier of the super-k-mer containing it in an integer vector $V[0, |\mathcal{S}_i|)$ — where $|\mathcal{S}_i| \leq 2^{i+1}$ — at position $f_i(x)$. Note that thanks to the skew distribution, these additional components take very little space, while granting constant time searches. In this way, LOOKUP(x) works as follows:

1. Compute the minimizer r of x and the values of $s = \mathit{Sizes}[f(r)]$ and $e = \mathit{Size}[f(r)+1]$. Bucket r has thus size $|B_r| = e - s$.

2. Let $b = \lceil \log_2(e-s) \rceil$. If $b \leq \ell$ the bucket is small, so the process follows as explained earlier. Otherwise, we know that x (if present) belongs to some partition i.

3. In the second case, retrieve the super-k-mer identifier $q = V_i[f_i(g)]$ and search for x in the super-k-mer whose offset is $\mathit{Offsets}[s+q]$.

In this way, the number of accesses made to *Strings* is limited to 2^ℓ.

4.6.2 Fulgor

Fulgor [56] is a colored de Bruijn graph index powered by SSHash and GGCAT. It follows the same structure described in Section 3.1: a k-mer dictionary $\mathcal{D}$ and an inverted index $\mathcal{L}$ for the colors. The dictionary is implemented using SSHash for two main reasons: firstly, it exploits both unitig Properties 1 and 2, and secondly, because a query returns (among other values) the UNITIGID(x) $= i$ of the queried k-mer x.

The map from unitigs to color sets exploits another key property of SSHash: the unitigs it stores internally can be permuted in any order, without impacting the dictionary's correctness or efficiency. In particular, unitigs are sorted by their COLORSETID(u_i), so that all unitigs having the same color set are stored consecutively in SSHash. To compute the COLORSETID(u_i) all that is required is a RANK$_1$ query over a bitvector $B[1..m]$, where:

$$B[i] = \begin{cases} 1 & \text{if } i < m \land \textsc{ColorSetID}(u_i) \neq \textsc{ColorSetID}(u_{i+1}) \\ 1 & \text{if } i = m \\ 0 & \text{otherwise} \end{cases} .$$

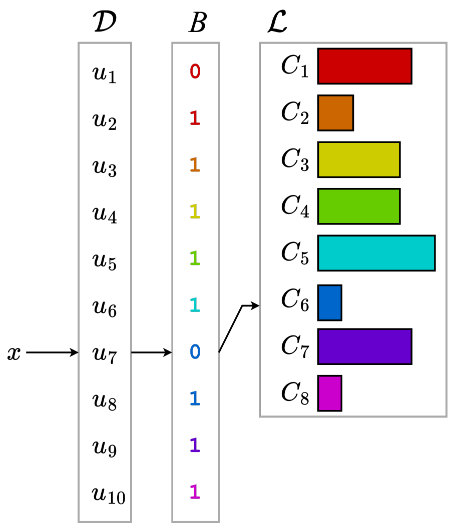


Figure 5: Schematic Fulgor representation of the cdBG in Figure 3. Reproduced, with permission, from [56]

In other words, the bitvector has $|\mathcal{C}| = z$ bits set, each on the last unitig belonging to the same color set. Since the Rank_1 operation can be implemented in $O(1)$ time (see Section 2.2.1), we can compute the color id of the unitig as $\text{ColorSetID}(u_i) = \text{Rank}_1(i, B) + 1$ in constant time (see Section 2.2). The process of finding the color set of a k-mer x is represented in Figure 5 and explained in the following example.

Example (Figure 5). First, the query $\text{Lookup}(x)$ is performed on SSHash, which returns the id of the k-mer together with $\text{UnitigID}(x) = 7$. Then, $\text{Rank}_1(7, B) + 1$ is computed to find the id of the color set. There are five 1s in $B[1, 7)$, so $\text{ColorSetID}(x) = 5 + 1 = 6$.

The inverted index $\mathcal{L}$ is a collection of sorted integer sequences $\{C_1, ..., C_z\}$. These are compressed using different strategies based on the density of the sequence C_i to be compressed, similarly to how it was done for Themisto (Section 4.5):

1. *Sparse* color set ($|C_i|/N < 1/4$): the differences between consecutive integers are computed and represented via the universal Elias δ code (see Section 2.1).

2. *Very Dense* color set ($|C_i|/N > 3/4$): the complementary set of C_i, $\overline{C_i} = \{j \in [1..N] \mid j \notin C_i\}$ is computed. The density of $\overline{C_i}$ is certainly less than $1/4$, so it can be compressed as a sparse color set.

3. *Dense* color sets, that do not belong to the previous categories, are stored as a bitvector $b[1..N]$, where $b[j] = 1$ if $j \in C_i$, and $b[j] = 0$ otherwise.

***Theorem* 1:** The selected thresholds for the density-based encoding strategy described above minimize the number of bits required to store the sets using the chosen encodings.

Proof. We start by calculating the upper limit of the sizes of the different encodings. Let $\textsc{Cost}_e(C_i)$ be the number of bits of the color set C_i, with the subscript e being either S or VD, for sparse and very dense sets respectively.

The size taken by dense sets is trivial, as there are exactly N bits in the bitvector, one for each color.

Let $G = \{g_1, ..., g_{|C_i|}\}$ be the the gaps of the sparse set $C_i = \{c_1, ..., c_{|C_i|}\}$, with $g_i = c_i - c_{i-1}$ and $g_1 = c_1$. Using Elias δ-codes is possible to store each gap g_i using $\lceil \log_2(g_i + 1)\rceil + 2\lceil \log_2 \log_2(g_i + 1)\rceil$ bits, so

$$\textsc{Cost}_{\mathrm{S}}(C_i) = \sum_{i=1}^{|C_i|} \left(\lceil \log_2(g_i + 1)\rceil + 2\lceil \log_2 \log_2(g_i + 1)\rceil\right).$$

We know that — using Jensen's inequality [57] — for a concave function f and $x_1 + \ldots + x_n = x$, the sum $\sum_i f(x_i)$ is maximized when each $x_i = x/n$. Since the sum of all gaps is at most N and the logarithm function is concave, we can apply Jensen's inequality to both left and right operands of the sum, obtaining

$$\begin{aligned}\textsc{Cost}_{\mathrm{S}}(C_i) &< \sum_{i=1}^{|C_i|} \left(\lceil \log_2(N/|C_i|)\rceil + 2\lceil \log_2 \log_2(N/|C_i|)\rceil\right)\\ &= |C_i| \log_2\left(\frac{N}{|C_i|}\right) + 2|C_i| \log_2 \log_2\left(\frac{N}{|C_i|}\right).\end{aligned}$$

The same reasoning can be made for very dense sets, changing the number of elements in the set to $N - |C_i|$:

$$\textsc{Cost}_{\mathrm{VD}}(C_i) < (N - |C_i|) \log_2\left(\frac{N}{N - |C_i|}\right) + 2(N - |C_i|) \log_2 \log_2\left(\frac{N}{N - |C_i|}\right).$$

Since $|C_i|/N < 1/4$ for a sparse set, its maximum cost becomes:

$$\begin{aligned}\textsc{Cost}_{\mathrm{S}}(C_i) &= |C_i| \log_2\left(\frac{N}{|C_i|}\right) + 2|C_i| \log_2 \log_2\left(\frac{N}{|C_i|}\right)\\ &< \frac{N}{4} \log_2(4) + \frac{N}{2} \log_2 \log_2(4)\\ &= \frac{N}{2} + \frac{N}{2}\\ &= N.\end{aligned}$$

Thus the cost of a sparse color set encoded with δ-codes is strictly smaller than N, so it is better than just using a bitvector.

The same process can be applied to very dense sets to obtain the same result, by noticing that if $|C_i|/N > 3/4$, then $N - |C_i| < N/4$. $\square$

The compressed representations are then concatenated into a single bitvector, and an additional sequence, *offsets*$[1..z]$, is used to store where each sequence begins. This sequence is compressed using Elias-Fano encoding (Section 2.3).

Up to date, Fulgor is the only cdBG index that takes advantage of all unitig properties (Section 3.2). k-mers are stored in the order they appear in the unitigs inside SSHash, with k-mers having different colors belonging to different unitigs (Properties 1 and 2). Color sets are stored only once, without ever repeating (Property 3), and are mapped to their relative unitigs through the bitvector B.

As we will see in the following section, this really makes the difference in experimental results, both in terms of space efficiency and query time.

4.7 Experimental Results

In this section, we will show and compare the experimental results of COBS, MetaGraph, Themisto, and Fulgor. We fixed the k-mer length to $k = 31$. All experiments were run on a machine equipped with Intel Xeon Platinum 8276L CPUs (2.20GHz), 500 GB of RAM, and running on Linux 4.15.0.

Both Themisto and COBS were built using the default parameters as suggested by their authors: with option `-d 20` for Themisto for better space effectiveness, using shards of at most 1024 references having Bloom filters with a false positive rate of 0.3 and one hash function to accelerate lookup for COBS. MetaGraph indexes were built with the relaxed row-diff BRWT data structure.

Datasets Experiments were run using the following pangenomes of bacteria:

- 3,682 *Escherichia Coli* (`EC`) genomes from NCBI [58]
- different collections of *Salmonella Enterica* (`SE`) from the "661K" collection by Blackwell et al. [59]
- 30,691 *Gut Bacteria* (`GB`) genomes, assembled from human gut samples, published by Hiseni et al. [60]

Table 3 outlines some statistics about these collections.

Note how the `GB` dataset has the highest number of k-mers and the smallest average color set size ($\approx$ 44 integers). This is because the dataset is much more diverse than the others, being composed of different bacterial species.

Genomes		EC	SE					GB
Number of colors (N)		3,682	5,000	10,000	50,000	100,000	150,000	30,691
Distinct color sets (z)	$\times 10^6$	5.59	2.69	4.24	13.92	19.36	23.61	227.80
Integers in color sets	$\times 10^9$	5.74	5.77	15.68	133.49	303.53	490.04	10.04
Average color set size	$\times 10^3$	1.03	2.14	3.70	9.69	15.68	20.76	0.04
k-mers in dBG (n)	$\times 10^6$	170.65	104.69	239.88	806.23	1,018.69	1,194.44	13,936.86
Unitigs in dBG (m)	$\times 10^6$	9.31	4.95	8.24	30.64	41.16	49.60	566.39

Table 3: Statistics for the tested collections, with $k = 31$

Dataset	COBS	MetaGraph			Themisto			Fulgor		
	Total	dBG	Color sets	Total	dBG	Color sets	Total	dBG	Color sets	Total
EC	**7.53**	0.10	0.23	**0.33**	0.22	1.85	**2.08**	0.29	1.36	**1.65**
SE-5k	**9.11**	0.07	0.19	**0.26**	0.14	1.29	**1.43**	0.16	0.59	**0.75**
SE-10k	**18.68**	0.13	0.38	**0.51**	0.32	3.50	**3.81**	0.35	1.66	**2.01**
SE-50k	**88.61**	0.36	1.95	**2.31**	1.07	32.42	**33.48**	1.25	17.03	**18.29**
SE-100k	**173.58**	0.45	3.50	**3.95**	1.35	75.94	**77.28**	1.71	40.71	**42.43**
SE-150k	**265.49**	*NA*	*NA*	***NA***	1.58	125.16	**126.74**	2.02	68.61	**70.65**
GB	**21.23**	5.23	4.77	**10.00**	18.33	30.88	**49.21**	21.29	15.54	**36.83**

Table 4: Index spaces in GB, broken down to space required for indexing the k-mers in the dBG (in order, BOSS for MetaGraph, SBWT for Themisto, and SSHash for Fulgor), and the data structures needed to encode the color sets and map them to the k-mers.

4.7.1 Index size

Table 4 reports the total *on disk* index size of the different methods evaluated.

COBS, despite being approximate, performs considerably worse than all of the others, except for the GB collection. This is likely because of the great diversity of that data, which causes the exact indexes to spend a considerable fraction of their total size on the representation of the k-mer dictionary. On the other hand, COBS does not utilize a dictionary structure, so, in this particular case, it can outperform Themisto and Fulgor.

Metagraph is consistently the smallest index, most of the time taking more than an order of magnitude less space than its competitors. Despite this remarkable achievement, its on-disk size does not reflect the working memory required for performing queries, as it will be shown in Section 4.7.2. Notice also how we were not able to construct the index for SE-150k since the construction required more resources than the ones offered by the machine it was run on.

Finally, we can see that Fulgor improves on the space usage of Themisto in all the inputs considered, bringing a 1.3× reduction on the GB case and a 1.8× reduction on the SE pangenomes.

Dataset	Hit rate	COBS		MetaGraph-B		MetaGraph-NB		Themisto		Fulgor	
		h:mm:ss	GB	mm:ss	GB	h:mm:ss	GB	h:mm:ss	GB	mm:ss	GB
EC	98.99%	45:11	34.93	22:00	30.44	1:05:41	0.40	3:40	2.46	2:10	1.67
SE-5k	89.49%	38:34	41.93	14:14	36.54	20:32	0.33	3:50	1.82	1:10	0.80
SE-10k	89.71%	1:01:14	84.20	28:15	92.18	43:40	0.61	7:35	4.16	2:20	2.06
SE-50k	91.25%	3:54:18	408.82	*NA*	*NA*	4:40:03	2.72	42:02	33.14	12:00	18.24
SE-100k	91.41%	8:07:29	522.56	*NA*	*NA*	9:40:06	4.82	1:22:00	75.93	24:00	42.20
SE-150k	91.52%	7:47:14	522.63	*NA*	*NA*	*NA*	*NA*	2:00:13	124.27	37:00	70.55
GB	92.91%	34:45	225.57	28:55	15.86	22:05	9.91	1:20	48.47	1:10	36.01

Table 5: Total query time and memory used by the process (maximum RSS) as reported by `/usr/bin/time -v`, using 16 threads. For this experiment, the output is written to `/dev/null`, to avoid recording I/O overhead.
The "B" query mode of MetaGraph corresponds to the batch mode (with default batch size), while "NB" corresponds to the non-batch query mode.

4.7.2 Query efficiency

Table 5 reports the query times of the indexes on a high-hit workload, i.e. when more than 90% of the queries have a non-empty result. Specifically, the measured times are for performing a pseudoalignment operation and are relative to a second run of each experiment, where indexes are loaded from the disk cache. There are various pseudoalignment algorithms, such as full-intersection or threshold-union. This experiment will focus on the former: given a query string Q, we consider it as a set of k-mers. Let $\mathcal{K}(Q) = \{x \in Q | \textsc{ColorSet}(x) \neq \varnothing\}$ be the set of all query k-mers also present in the index. The full intersection method computes the intersection between the color sets of all the k-mers in $\mathcal{K}(Q)$, i.e. $\bigcap_{x \in \mathcal{K}(Q)} \textsc{ColorSet}(x)$.

The queries consist of all FASTQ records in the first read file of the following accessions: SRR1928200 for EC, SRR801268 for SE, and ERR321482 for GB, each file containing many millions of reads.

COBS is generally much slower than the other indexes, except for MetaGraph in non-batch mode. This is likely because the input collections are partitioned into shards of references, in order to build Bloom filters of different sizes and thus save space. At query time, however, a k-mer lookup has to be resolved by every shard and individual results combined.

As stated in the previous section, it is not the case with all indexes that the size of the index *on disk* is a good metric for the memory required to actually query the index. Specifically, for MetaGraph in batch mode (B), the required memory can exceed the index size by up to 2 orders of magnitude, resulting in the exhaustion of available memory in several tests and the inability to complete the queries. On the other hand, Fulgor, Themisto, and MetaGraph when not executed in batch mode (NB) require only a small constant amount of working memory beyond the size of the index present on the disk.

Dataset	COBS		MetaGraph		Themisto		Fulgor	
	h:mm	GB	h:mm	GB	h:mm	GB	h:mm	GB
EC	0:03	6	0:46	149	0:19	17	0:06	17
SE-5k	0:09	8	0:47	191	0:11	13	0:04	13
SE-10k	0:17	16	1:50	219	0:25	24	0:09	24
SE-50k	1:41	82	14:16	120	2:32	96	1:13	44
SE-100k	2:37	84	26:40	104	6:25	202	2:56	74
SE-150k	4:54	159	*NA*	*NA*	10:00	323	4:36	137
GB	0:22	17	10:50	100	6:21	184	2:27	115

Table 6: Total index construction time and GB of memory used during construction (maximum RSS), as reported by `/usr/bin/time -v`, using 48 threads. The times include the time to serialize the index on the disk.
In red font: the constructions exceeding the available memory (> 500GB) for which it was necessary to cap the maximum memory usage to 100GB.

Finally, we note that Fulgor is the fastest index after Themisto, being almost 4× faster in all SE datasets.

4.7.3 Construction time and space

In Table 6 we consider the resources needed to build the indexes.

The fastest indexes to build are Fulgor and COBS, the latter being even faster on the GB collection, since — as already explained — it does not build an exact dictionary for the k-mers. The tested MetaGraph configuration is significantly slower. We were unable to build the index for SE-150K within 3 days and using 48 parallel threads. Also, its construction produced more than 1TB of intermediate files.

Chapter 5

Repetition Aware Compression

When indexing large pangenomes, the space taken by the color sets $\mathcal{C} = \{C_1, ..., C_z\}$ dominates the index space (Table 4, "Color sets" column). As a consequence, improving the memory usage of c-dBGs translates to devising better compression algorithms for the color sets. In the following sections, we will focus on exploiting the following key property:

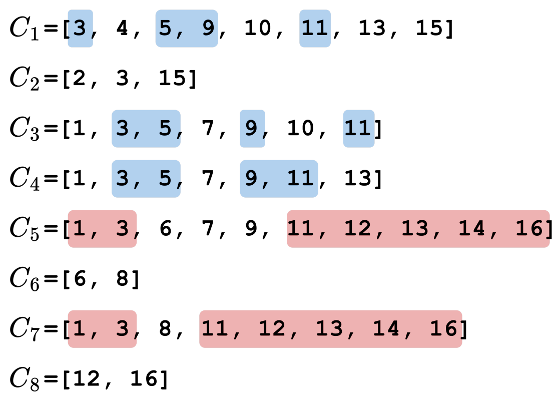
"The genomes in a pangenome are very similar"

which in turn implies that the color sets are also very similar. This property, closely related to the setting of the problem, enables substantially better compression effectiveness. Color sets are similar in the sense that they share many, potentially very long, identical integer sub-sequences, that we will call **patterns**.

Consider Figure 6 for some examples. The pattern $[3, 5, 9, 11]$ repeats in C_1, C_3, C_4, hence it is represented redundantly three times. Similarly, the longer pattern $[1, 3, 11, 12, 13, 14, 16]$ is repeated twice. These simple examples suggest that such patterns can repeat in many sets, hence increasing the redundancy and aggravating the memory usage of the index.

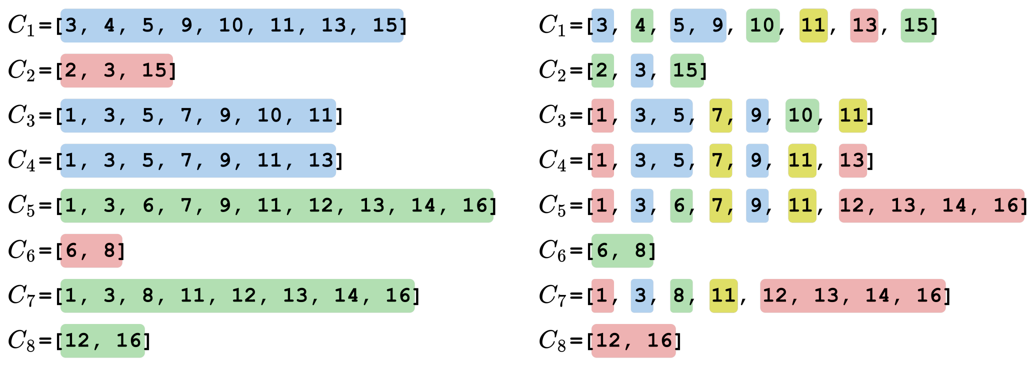
To address this issue, we developed two solutions based on **partitioning** the color sets to factor out repetitive patterns. Encoding repetitive patterns once clearly reduces the amount of redundancy in the inverted index, improving the space of data structures. In particular, we investigate how two different partitioning paradigms affect the compression and query speed of the color sets. We refer to them as **horizontal** partitioning (Section 5.2) and **vertical** partitioning (Section 5.3), based on whether the data is grouped by color sets or by reference, respectively. Figure 7 shows an example of these paradigms: intuitively, partitions of similar rows are created by horizontal partitioning, while partitions of similar columns are created by vertical partitioning.

Before presenting the details of the solutions, we establish the following fact. Given an integer $q > 0$, let $\mathcal{N} = \{\mathcal{N}_1, ..., \mathcal{N}_r\}$ be a partition of $[q] = \{1, ..., q\}$ of



C_1=[3, 4, 5, 9, 10, 11, 13, 15]
C_2=[2, 3, 15]
C_3=[1, 3, 5, 7, 9, 10, 11]
C_4=[1, 3, 5, 7, 9, 11, 13]
C_5=[1, 3, 6, 7, 9, 11, 12, 13, 14, 16]
C_6=[6, 8]
C_7=[1, 3, 8, 11, 12, 13, 14, 16]
C_8=[12, 16]

Figure 6: Color sets from Figure 3 where repeating patterns are highlighted in different shades



C_1=[3, 4, 5, 9, 10, 11, 13, 15]
C_2=[2, 3, 15]
C_3=[1, 3, 5, 7, 9, 10, 11]
C_4=[1, 3, 5, 7, 9, 11, 13]
C_5=[1, 3, 6, 7, 9, 11, 12, 13, 14, 16]
C_6=[6, 8]
C_7=[1, 3, 8, 11, 12, 13, 14, 16]
C_8=[12, 16]

(a) Three horizontal partitions

C_1=[3, 4, 5, 9, 10, 11, 13, 15]
C_2=[2, 3, 15]
C_3=[1, 3, 5, 7, 9, 10, 11]
C_4=[1, 3, 5, 7, 9, 11, 13]
C_5=[1, 3, 6, 7, 9, 11, 12, 13, 14, 16]
C_6=[6, 8]
C_7=[1, 3, 8, 11, 12, 13, 14, 16]
C_8=[12, 16]

(b) Four vertical partitions

Figure 7: Color sets from Figure 3 partitioned (a) horizontally and (b) vertically.

size r. Let an order between the elements $\{e_{ij}\}$ of each $\mathcal{N}_i$ be fixed, e.g. by sorting the elements in increasing order.

Fact 1. Any partition $\mathcal{N}$ induces a permutation $\pi : [q] \to [q]$ defined as $\pi(e_{ij}) : j + B_{i-1}$, where $B_i := \sum_{t=1}^{i} |\mathcal{N}_t|$ for $i > 0$ and $B_0 := 0$, for $i = 1, ..., r$ and $j = 1, ..., |\mathcal{N}_i|$

Example (Figure 7a). We have $q = z = 8$ and $r = 3$. The partitions are defined as $\mathcal{N}_1 = \{1, 3, 4\}$, $\mathcal{N}_2 = \{2, 6\}$, and $\mathcal{N}_3 = \{5, 7, 8\}$. The **boundaries** B_i are therefore $B_0 = 0$, $B_1 = 3$, $B_2 = 5$, $B_3 = 8$. The permutation π can be visually obtained by concatenating the sets $\mathcal{N}_i$ and assigning consecutive integers from 1 to q to the elements:

$$\begin{array}{rccccccccccccc}
 & & e_{11} & e_{12} & e_{13} & & e_{21} & e_{22} & & e_{31} & e_{32} & e_{33} & \\
\mathcal{N} = & \{ & 1 & 3 & 4 & \}\,\{ & 2 & 6 & \}\,\{ & 5 & 7 & 8 & \} \\
 & & \downarrow & \downarrow & \downarrow & & \downarrow & \downarrow & & \downarrow & \downarrow & \downarrow & \\
\pi(e_{ij}) = & & 1 & 2 & 3 & & 4 & 5 & & 6 & 7 & 8 &
\end{array}$$

This means that $\pi(1) = 1$, $\pi(3) = 2$, $\pi(4) = 3$, and so on, that is $\pi = [1, 4, 2, 3, 6, 5, 7, 8]$.

To better understand some implementation details described in the following sections, we will now illustrate a framework for the construction of the partitioned indexes.

5.1 The SCPO Framework

This framework is based on the intuition that similar objects (color sets in $\mathcal{C}$ for horizontal partitioning, references in $\mathcal{R}$ for vertical partitioning) should be grouped in the same partition to increase the likeliness of having shared patterns. It consists of four steps: Sketching, Clustering, Partitioning, and Ordering.

Sketching. Before the actual partitioning, the objects are pre-processed to make the process faster and less memory intensive. This is carried out by computing the *sketches* of the input objects, which, besides being much smaller than the original data, should also preserve similarity: if two sketches are similar, then the original objects should be also similar.

For the horizontal partitioning, it's sufficient to build one sketch per color set. For the vertical partitioning, recall from Property 1 (Section 3.2) that each reference $R_i \in \mathcal{R}$ can be spelled by a concatenation of unitigs having i in their color set. If these unitigs are assigned a unique identifier, as in SSHash, each R_i can be seen as a list of unitig identifiers, i.e. an integer list. Sketches are thus built for each such list.

Clustering. The sketches are passed as input to a clustering algorithm

Partitioning. Once the clustering is completed, each object is labeled with the cluster identifier of the corresponding sketch, so that $\mathcal{N} = \{\mathcal{N}_1, ..., \mathcal{N}_r\}$ is uniquely determined.

Ordering. This step is optional, depending on the type of partitioning. While it is completely superfluous for the horizontal partitioning, it is decisive for the compression of the vertical partitioned index, as it will be shown in Section 5.3.

For the experiments in this work, we set up the framework as follows:

- Sketches are built using the *hyper-log-log* algorithm [61], with a size of $W = 2^{10}$ bytes each.

- As for the clustering algorithm, we used a *divisive K-means* approach, that dynamically computes the number of clusters to generate based on the input sketches, without requiring it as a parameter. At the beginning, all input data forms a single cluster, that is recursively split until the mean square error (MSE) between the sketches in the cluster and its centroid is not below a certain threshold, in our case 10% of the MSE at the start of the algorithm. The complexity of the algorithm depends on the topology of the binary tree representing the cluster splits. Let Z be the number of sketches. In the worst case, the tree is completely unbalanced and the complexity is $O(WZr)$, while in the best case, the tree is perfectly balanced and the complexity is $O(WZ \log r)$. Since $z \gg N$, we expect the horizontal clustering step to require more resources than vertical clustering.

5.2 Horizontal Partitioning: Representative and Differential Color Sets

The general idea of horizontal partitioning is to organize the sets in $\mathcal{C}$ into groups of similar sets (Figure 7a). Then, for each group, a representative color set is computed and all sets in the group are encoded using a differential set, with respect to the representative.

Definition 4 (Horizontal partitioning)**.** Let $\mathcal{N} = \{\mathcal{N}_1, ..., \mathcal{N}_r\}$ be a partition of $[z]$, of size $r > 0$, and let π be its corresponding permutation. For each $\mathcal{N}_i$, we build a set A_i so that each color set C_j can be represented as $(A_i \Delta C_j)$, for all $j \in \mathcal{N}_i$. Notation $(X \Delta Y)$ stands for the *symmetric difference* between the sets X and Y, that is $(X \cup Y) \setminus (X \cap Y)$. The idea is that the set A_i should include the most repetitive colors that occur in the color sets of partition $\mathcal{N}_i$ so that each difference $(A_i \Delta C_j)$ is small. Since A_i expresses the repetitiveness of $\mathcal{N}_i$, it is named **representative** color set of $\mathcal{N}_i$. The symmetric difference $(A_i \Delta C_j)$ is instead called **differential** color set of C_j. We then define $\mathcal{A} = \{A_1, ..., A_r\}$ as the set of all representative color sets, one per partition. Similarly, we indicate with Δ the set of all differential color sets, where $|\Delta| = z$ as we have one differential color set for each original color set in $\mathcal{C}$.

Compared to a c-dBG $G(\mathcal{U}, \mathcal{C})$, its differential c-dBG (Dfc-dBG) variant is the graph $G(\mathcal{U}, \mathcal{N}, \pi, \mathcal{A}, \Delta)$, where the set of nodes $\mathcal{U}$ is the same, but the color sets $\mathcal{C}$ are split into $\mathcal{A}$ and Δ.

Example. Consider the $r = 3$ partitions from Figure 7a, and assume the following representative color sets are built:

$$A_1 = [1, 3, 5, 7, 9, 10, 11, 13]$$
$$A_2 = [2, 3, 6, 8, 15]$$
$$A_3 = [1, 3, 11, 12, 13, 14, 16]$$

Then, we can compute the differential color sets:

$$\begin{array}{lll}
(A_1 \,\Delta\, C_1) = [1, 4, 7, 15] & (A_2 \,\Delta\, C_2) = [6, 8] & (A_3 \,\Delta\, C_5) = [6, 7, 9] \\
(A_1 \,\Delta\, C_3) = [13] & (A_2 \,\Delta\, C_6) = [2, 3, 15] & (A_3 \,\Delta\, C_7) = [8] \\
(A_1 \,\Delta\, C_4) = [10] & & (A_3 \,\Delta\, C_8) = [1, 3, 11, 13, 14]
\end{array}$$

We can see that the pattern $[3, 9, 11]$, shared by the sets C_1, C_3, and C_4 is now encoded only once in the set A_1, and implicitly encoded in each differential set $(A_1 \Delta C_1)$, $(A_1 \Delta C_3)$, and $(A_1 \Delta C_4)$. The same is true for C_5 and C_7, where the whole representative set is present in both color sets. Unfortunately, it is not always the case that the differential set is shorter than the original color set. C_8, for example, is only 2 integers long, whereas its differential set $(A_3 \,\Delta\, C_8)$ has a length of 5 since we must "remove" many colors from the representative set. However, inside the third partition, we still had a net gain on the required integers to encode all the color sets.

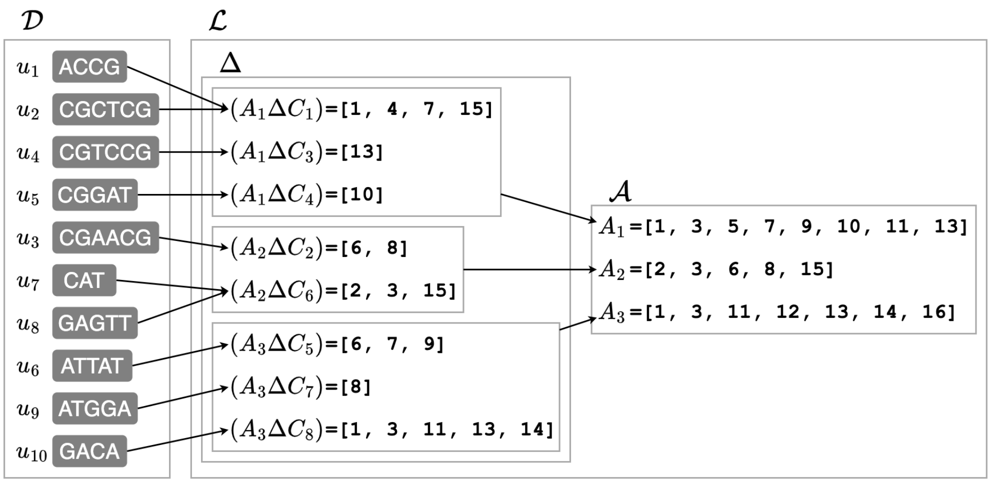


Figure 8: The Dfc-dBG built from the color sets in Figure 3.

Figure 8 is a schematic representation of all the components of the Dfc-dBG just described. Note that the color sets have been permuted according to $\pi = [1,4,2,3,6,5,7,8]$ and that the unitigs in $\mathcal{D}$ are sorted following the permuted order of the color sets.

5.2.1 The Optimization Problem

The effectiveness of the Dfc-dBG clearly depends on the choice of the partition $\mathcal{N}$ and how the representative for each partition $\mathcal{N}_i$ is built. The main idea is to group similar color sets in the same cluster and construct their representative based on their most repetitive patterns. Having fewer partitions, thus a small r, amortizes the cost of the representative color sets, but leads to bigger — and consequently more varied — partitions. Conversely, smaller partitions better highlight the repetitiveness of the patterns in the collection, albeit requiring more space to store the many representative colors that must be created.

Let $\textsc{Cost}(L)$ be the encoding cost of the sorted list L. The optimization problem faced by the Dfc-dBG, called **minimum-cost partition arrangement** (MPA), can be stated as follows:

> **Problem 2** (MPA for Dfc-dBG)**.** Let $G(\mathcal{U},\mathcal{C})$ be the compacted c-dBG built from the reference collection $\mathcal{R} = \{R_1, ..., R_N\}$, where $|\mathcal{C}| = z$. Determine the partition $\mathcal{N} = \{\mathcal{N}_1, ..., \mathcal{N}_r\}$ of $[z]$ for some $r > 0$ and the sets $A_1, ..., A_r$, such that
> $$\sum_{i=1}^{r} \textsc{Cost}(A_i) + \sum_{i=1}^{r} \sum_{j \in \mathcal{N}_i} \textsc{Cost}(A_i \,\Delta\, C_j)$$
> is minimum.

We suspect that this problem is hard depending on the chosen encoding method. Instead, we prove the following theorem for the construction of representative colors.

***Theorem* 2:** Given a partition $\mathcal{N} = \{\mathcal{N}_i, ..., \mathcal{N}_r\}$ of $[z]$ of size $r > 0$, let $\textsc{Cost}(L) = |L|$ and

$$A_i = \left\{ c \in C_j \mid j \in \mathcal{N}_i \wedge occ_i(c) \geq \left\lceil \frac{|\mathcal{N}_i|}{2} \right\rceil \right\}, \text{ for } i = 1, ..., r$$

where $occ_i(c) \in [1, |\mathcal{N}_i|]$ is the number of occurrences of the integer c in the color sets of $\mathcal{N}_i$. Then the cost $\sum_{i=1}^{r} \sum_{j \in \mathcal{N}_i} |(A_i \,\Delta\, C_j)|$ is minimum.

Posed in simpler terms, the representative set A_i that minimizes the size of the differential sets consists of all integers appearing in at least half the color sets of the partition $\mathcal{N}_i$. In this case, it was chosen to define $\textsc{Cost}(L) = |L|$ to minimize the number of integers being encoded in the differential color sets, as it is the most expensive component in the index.

Proof. (By contradiction) Assume that A_i is optimal and there exist an integer $c \in A_i$ such that $occ_i(c) < \lceil |\mathcal{N}_i|/2 \rceil$. This means that there are $|\mathcal{N}_i| - occ_i(c)$ differential color sets containing x since it must be "removed" from all color sets where it is not present. Let $\textsc{Cost}(A_i) = \sum_{j \in \mathcal{N}_i} |(A_i \,\Delta\, C_j)|$. We can therefore remove c from A_i to obtain a new solution $A'_i = A_i \backslash \{c\}$ such that

$$\begin{aligned}
\textsc{Cost}(A'_i) &= \textsc{Cost}(A_i) - (|\mathcal{N}_i| - occ_i(c)) + occ_i(c) \\
&= \textsc{Cost}(A_i) - |\mathcal{N}_i| + 2 \cdot occ_i(c) \\
&< \textsc{Cost}(A_i) - |\mathcal{N}_i| + 2 \cdot \lceil |\mathcal{N}_i|/2 \rceil \\
&< \textsc{Cost}(A_i)
\end{aligned}$$

Hence, solution A'_i has a lower cost than A_i, contradicting the initial assumption that A_i is optimal. □

All representative color sets in the previous examples are built with this strategy. Consider the color sets in the partition $\mathcal{N}_3 = \{C_5, C_7, C_8\}$ from Figure 7a. In this case $\mathcal{N}_3 = 3$ and $\lceil |\mathcal{N}_3|/2 \rceil = 2$, so any integer appearing at least twice among C_5, C_7, and C_8 is included in A_3. It is easy to see that the integers appearing more than once are $[1, 3, 11, 12, 13, 14, 16]$.

The number of occurrences $occ_i(c)$ of each color c can be computed by iterating all color sets $\mathcal{C}$ once and, since there are at most N distinct integers in each partition $\mathcal{N}_i$, the sets $A_1, ..., A_r$ are computed in a total time of $O(\sum_{i=1}^{z} |C_i| + r \cdot N)$.

5.2.2 Analysis and Implementation Details

Both sets $\mathcal{A}$ and Δ consist of sequences of increasing integers, that can be compressed efficiently using many different methods. In our case, representative and differential sets are stored as the differences between consecutive integers encoded with Elias δ codes, plus $O(|\delta(N)|)$ bits to store their sizes.

The order in which differential color sets are stored is crucial to efficiently determine the partition a color belongs to. Color sets are stored in the order given by the permutation π so that sets belonging to the same partition are placed consecutively. Then, a bitvector $b[1..z]$ can be used to define the last color set i of each partition, by setting the value $b[i] = 1$. Continuing from the previous example:

$$\begin{array}{r|ccc|cc|ccc}
 & C_1 & C_3 & C_4 & C_2 & C_6 & C_5 & C_7 & C_8 \\
b = & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 1
\end{array}$$

In this way, by using just $1+o(1)$ extra bits per color set, it is possible to compute the partition $p \in [1, r]$ of color C_j as $p = \text{RANK}_1(b, \pi(j)) + 1$ in constant time.

Decoding C_j from $(A_i \Delta C_j)$ is efficient. By definition of the symmetric set difference, $C_j = (A_i \Delta (A_i \Delta C_j))$, which can be implemented in linear time in the size of the two sets. Also note that $|C_j| \leq |A_i| + |(A_i \Delta C_j)|$, with equality holding only when $A_i = \varnothing$ or $(A_i \Delta C_j) = \varnothing$. Thus, decoding takes more time than just scanning the original color C_j, imposing some overhead compared to other representations that encode sets individually.

Recall that the complexity of the clustering algorithm is $O(WKr)$, where W is the size of the sketches and $K = z$ is the number of input objects. Also note that r is strictly related to K, as a greater number of data points will most likely increase the number of clusters. Since z is in the order of 10^6 (see Table 3), it takes a considerable amount of time to cluster the sketches altogether, in some cases even days. To overcome this issue and improve the cluster's locality as well, sketches are first grouped in **slices** based on their density, then each slice is clustered independently from the others, and the results are combined. This tweak led to a significant reduction in the build times, together with a slight improvement in the index size.

In our implementation of the index, the slicing method has not been thoroughly explored yet and has been implemented as a series of predefined density thresholds: $[0, 0.25)$, $[0.25, 0.50)$, $[0.50, 0.75)$, $[0.75, 1]$. We leave the investigation of better strategies as an opportunity for future work.

5.2.3 Experimental Results

Refer to Section 4.7 for a rundown on the modality of the experiments.

To determine the compression capabilities of horizontal partitioning, we applied the techniques just described to the state-of-the-art Fulgor index. We called this new index variant **d-Fulgor** (i.e. *differential Fulgor*).

We will now focus on comparing Fulgor and d-Fulgor, particularly regarding their space effectiveness, query efficiency, and build time.

Index size

Table 7 reports the total index size of Fulgor compared to d-Fulgor. We pose our attention on the space taken by the compressed color sets rather than the total size, given that the partitioning only regards the color sets and not the dBG part of the index (i.e. SSHash).

The results show that horizontal partitioning has a great improvement over the space required to store the color sets, up to 4.3× smaller space for the SE-150k dataset.

Dataset	dBG	Fulgor			d-Fulgor		
		Color sets		Total	Color sets		Total
EC	0.29	**1.36**	(83%)	1.65	**0.45**	(61%)	0.74
SE-5k	0.16	**0.59**	(79%)	0.75	**0.20**	(56%)	0.36
SE-10k	0.35	**1.66**	(83%)	2.01	**0.48**	(58%)	0.83
SE-50k	1.25	**17.03**	(93%)	18.29	**4.31**	(77%)	5.57
SE-100k	1.71	**40.71**	(96%)	42.43	**9.37**	(84%)	11.10
SE-150k	2.02	**68.61**	(97%)	70.65	**15.73**	(89%)	17.77
GB	21.29	**15.54**	(42%)	36.83	**7.51**	(26%)	28.81

Table 7: Index spaces in GB, broken down to space required for indexing the k-mers in the dBG (equal for both Fulgor and d-Fulgor), and the data structures needed to encode the color sets and map them to the k-mers. In gray the percentage of space taken by the color sets with respect to the total.

Dataset	Hit rate	Fulgor		d-Fulgor	
		mm:ss	GB	h:mm:ss	GB
EC	98.99%	**2:10**	1.67	**5:20**	0.78
SE-5k	89.49%	**1:10**	0.80	**2:00**	0.41
SE-10k	89.71%	**2:20**	2.06	**4:30**	0.90
SE-50k	91.25%	**12:00**	18.24	**29:00**	5.82
SE-100k	91.41%	**24:00**	42.20	**1:02:00**	11.58
SE-150k	91.52%	**37:00**	70.55	**1:38:00**	18.51
GB	92.91%	**1:10**	36.01	**1:00**	28.17

Table 8: Total query time and memory used by the process as reported by `/usr/bin/time -v`, using 16 threads. For this experiment, the output is written to `/dev/null` to avoid recording I/O overhead.

Query efficiency

As predicted earlier, Table 8 shows that d-Fulgor is slower at querying the results compared to the original Fulgor, taking around 2.5× more time. This slowdown is caused by the fact that to decode each color set, d-Fulgor has to iterate both the representative and the differential set once, whose combined length is (almost always) greater than the original color set.

Construction time and space

For these experiments, the d-Fulgor index was built from an existing Fulgor index. Indeed, to obtain the real time taken to construct the differential index, one must sum the times of the two indices reported in Table 9, as hinted by the + sign on the fourth column.

We can see how the partitioning step is relatively fast compared to the construction of the original index, except for the GB dataset, where the number of

Dataset	Fulgor		d-Fulgor	
	h:mm	GB	h:mm	GB
EC	0:06	17	+0:12	11
SE-5k	0:04	13	+0:06	8
SE-10k	0:09	24	+0:09	14
SE-50k	1:13	44	+0:43	105
SE-100k	2:56	74	+1:20	207
SE-150k	4:36	137	+1:55	305
GB*	2:27	115	+10:00	182

Table 9: Total index construction time and GB of memory used during construction, as reported by `/usr/bin/time -v`, using 48 threads. The times include the time to serialize the index on the disk.
*Given the peculiarity of the data, GB was built using different clustering parameters, significantly improving build time and process memory, without impacting the index quality.

distinct color sets (227.8×10^6) is one order of magnitude greater than SE-150k. Consequently, also considering that GB is a highly varied collection of data, the clustering process takes a considerable amount of time (around 8 hours) to complete.

Nonetheless, memory usage is extremely high, taking more than half of the available RAM for the biggest datasets. This arises from the fact that the number of sketches that are being clustered is very high, proportional to the number of distinct color sets.

5.3 Vertical Partitioning: Partial and Meta Color Sets

The other solution, based on vertical partitioning (Figure 7b), works by creating a color set hierarchy. Each color set C_i is spelled by a list of references — called meta colors — to smaller repetitive patterns, referred to as partial color sets.

Definition 5 (Vertical partitioning)**.** Let $\mathcal{N} = \{\mathcal{N}_1, ..., \mathcal{N}_r\}$ be a partition of $[N]$, of size $r > 0$, and let π be its induced permutation. Assume that the N reference identifiers and the integers in the sets of $\mathcal{C}$ have been permuted according to π. After the permutation, $\mathcal{N}$ determines a partition of $\mathcal{R}$ into r disjoint sets:

$$\mathcal{R}_1 = \{R_i \mid 0 = B_0 < i \leq B_1\}, ..., \mathcal{R}_r = \{R_i \mid B_{r-1} < i \leq B_r = N\}$$

Let $\mathcal{P}_i$ be the set

$$\mathcal{P}_i = \{\{c - B_{i-1} \mid c \in C_t \cap \{B_{i-1}+1, B_{i-1}+2, ..., B_i - 1, B_i\}\}, \forall C_t \in \mathcal{C}\}$$

for $i = 1, ..., r$. The elements P_{ij} of the set $\mathcal{P}_i$ are the **partial color sets** induced by the partition $\mathcal{N}_i$. We indicate with $\mathcal{P} = \{\mathcal{P}_1, ..., \mathcal{P}_r\}$ the set of all partial color sets. In simpler terms, $\mathcal{P}_i$ is the set obtained by considering the distinct color sets only for the references in the i-th partition $\mathcal{R}_i$, noting that they comprise integers c such that $B_{i-1} < c \leq B_i$.

Let $C_t \in \mathcal{C}$ be a color set. A **meta color** is an integer pair (i, j) indicating the sub-list $L := C_t = [b..\ b + |P_{ij}|]$ if there exists a value $b \in (0, |C_t| - |P_{ij}|]$ such that $L[l] = P_{ij}[l] + B_{i-1}$, for each $l = 1, ..., |P_{ij}|$. In other words, the tuple refers to a sub-list of C_t of size $|P_{ij}|$ such that each element in L is equal to each element in P_{ij}, accounting for the B_{i-1} shift applied in the definition of $\mathcal{P}_i$. It follows that C_t can be modeled as a list M_t of at most r meta colors. We indicate with $\mathcal{M} = \{M_1, ..., M_z\}$ the set of all meta color sets.

We have obtained a representation of $\mathcal{C}$ that consists of the sets $\mathcal{M}$ and $\mathcal{P}$, which are made up of integer sequences that can be further compressed. Thus, given $G(\mathcal{U}, \mathcal{C})$, the meta-colored c-dBG (or Mac-dBG) is the graph $G(\mathcal{U}, \mathcal{N}, \pi, \mathcal{P}, \mathcal{M})$, where the set of nodes $\mathcal{U}$ is the same, but the color sets are split into $\mathcal{P}$ and $\mathcal{M}$

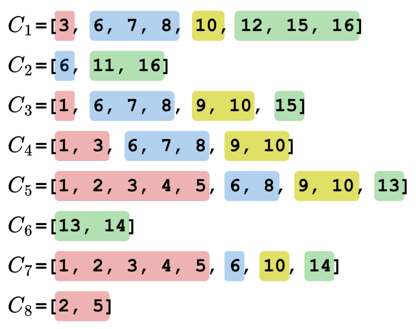
Example. Consider the $r = 4$ partitions from figure 7b, with $\mathcal{N}_1 = \{1, 12, 13, 14, 16\}$, $\mathcal{N}_2 = \{3, 5, 9\}$, $\mathcal{N}_3 = \{7, 11\}$, and $\mathcal{N}_4 = \{2, 4, 6, 8, 10, 15\}$. Thus we have $B_1 = 5$, $B_2 = 8$, $B_3 = 10$, $B_4 = 16$, and the corresponding permutation $\pi = [1, 11, 6, 12, 7, 13, 9, 14, 8, 15, 10, 2, 3, 4, 16, 5]$. After applying the permutation to each color set, we obtain the permuted color sets as in Figure 9a. Notice how the color sets are shaded in the same way as in Figure 7b, but integers belonging to the same partition are now consecutive. For example C_1, that before was $[3, 4, 5, 9, 10, 11, 13, 15]$, now is

$$[\pi(3), \pi(4), \pi(5), \pi(9), \pi(10), \pi(11), \pi(13), \pi(15)] = [6, 12, 7, 8, 15, 10, 3, 16]$$

or $[3, 6, 7, 8, 10, 12, 15, 16]$ once sorted.

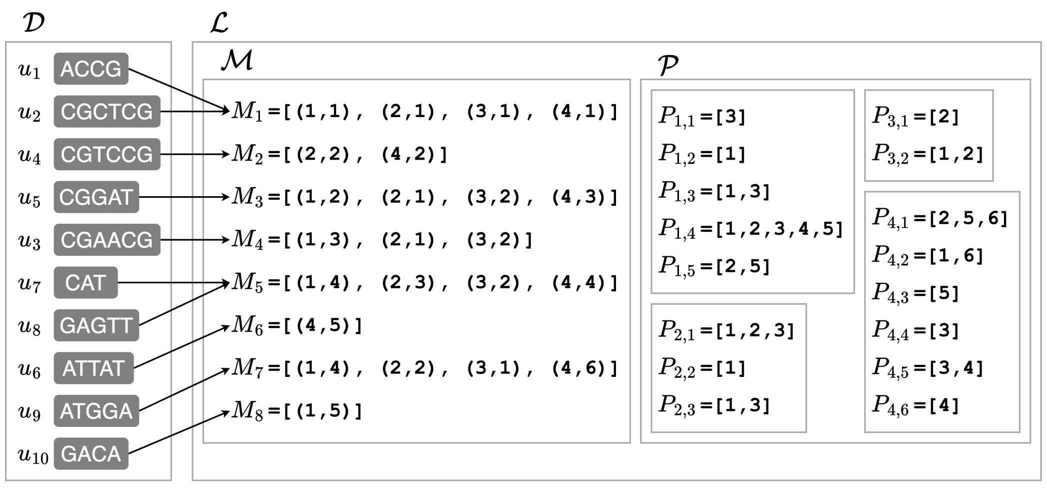
The partial color sets are the distinct subsequences in each partition of the permuted color sets. For example, $\mathcal{P}_2$ is the set of the distinct subsequences in partition 2, i.e. those comprising the integers c such that $6 \leq c \leq 8$. Looking again at Figure 9a, the subsequences shaded in blue are $[6, 7, 8]$, $[6]$, and $[6, 8]$. From each integer in these lists is then subtracted $B_{2-1} = 5$, obtaining $[1, 2, 3]$, $[1]$, and $[1, 3]$.

Figure 9b shows all partial sets obtained in this way, as well as the color sets $\mathcal{C} = \{C_1, ..., C_8\}$ expressed as meta color lists.



C_1=[3, 6, 7, 8, 10, 12, 15, 16]
C_2=[6, 11, 16]
C_3=[1, 6, 7, 8, 9, 10, 15]
C_4=[1, 3, 6, 7, 8, 9, 10]
C_5=[1, 2, 3, 4, 5, 6, 8, 9, 10, 13]
C_6=[13, 14]
C_7=[1, 2, 3, 4, 5, 6, 10, 14]
C_8=[2, 5]

(a) Vertical partitioning with permuted colors



(b) The Mac-dBG built from the color sets in Figure 3.

Figure 9: An example application of vertical partitioning. (a) The color sets after the colors are assigned their new identifiers, with the partitions highlighted in different colors. (b) The resulting Mac-dBG.

5.3.1 The Optimization Problem

As already noted in Section 5.2.1 for the Dfc-dBG, the effectiveness of the Mac-dBG also strongly depends on the choice of the partition $\mathcal{N}$, and the order of the references as given by π. Indeed, there is a trade-off between the encoding cost of the partial and meta color sets. Let $N_m = |\mathcal{M}|$ be the number of meta color sets and $N_p = \sum_{i=1}^{r} |\mathcal{P}_i|$ the number of partial color sets. Since each meta color can be indicated with $\log_2(N_p)$, the meta color sets cost in total $N_m \log_2(N_p)$ bits. Let $\textsc{Cost}(P_{ij}, \pi)$ be the cost in bits of the partial color set P_{ij}. On one hand, we would like to select a large value of r (a greater number of partitions) so that N_p becomes small, as each color is partitioned into small partial color sets, thus increasing the chances that each partition has many repeated patterns. This helps in reducing the encoding cost for the partial color sets $\sum_{i=1}^{r}\sum_{j=1}^{|\mathcal{P}_i|}\textsc{Cost}(P_{ij},\pi)$, but, at the same time, increases the number of meta color sets N_m, possibly nullifying the benefits encoding shared patterns.

The minimum-cost partition arrangement (MPA) problem for the Mac-dBG is therefore as follows:

> **Problem 3** (MPA for Mac-dBG)**.** Let $G(\mathcal{U},\mathcal{C})$ be the compacted c-dBG built from the reference collection $\mathcal{R}=\{R_1,...,R_N\}$. Determine the partition $\mathcal{N}=\{\mathcal{N}_1,...,\mathcal{N}_r\}$ of $[N]$ for some $r>0$ and permutation $\pi:[N]\to[N]$ such that
>
> $$N_m\log_2(N_p)+\sum_{i=1}^{r}\sum_{j=1}^{|\mathcal{P}_i|}\textsc{Cost}(P_{ij},\pi)$$
>
> is minimum.

Depending on the chosen encoding, smaller values of $\textsc{Cost}(P_{ij},\pi)$ may be obtained when the gaps between subsequent reference identifiers are minimized. Finding the permutation π that minimizes the gaps between the identifiers, over all partial color sets, is an instance of the **bipartite minimum logarithmic arrangement** (BIMLOGA) problem [62]. In short, the objective of the BIMLOGA problem is to find a permutation π of some vertices in a bipartite graph, such that the sum of the logarithm of the gaps between consecutive integers is minimum. This problem is NP-hard. It can be easily seen that BIMLOGA is a special case of the MPA, for $r=1$ and with $\textsc{Cost}(P_{ij},\pi)$ being the objective function of the problem. Since a polynomial solution to the MPA would provide a polynomial solution for the BIMLOGA problem, it follows that also the MPA is NP-hard under these constraints, meaning that it is unlikely that polynomial-time algorithms exist for solving the MPA.

5.3.2 Analysis and Implementation Details

If $N_p=\sum_{i=1}^{r}|\mathcal{P}_i|$ is the total number of partial color sets, then each meta color (i,j) can be indicated with just $\log_2(N_p)$ bits. In particular, in the current implementation of the meta-partial index, each meta color (i,j) is stored as $m_{ij}=\sum_{t=1}^{i-1}|\mathcal{P}_t|+j$, with $1\le m_{ij}\le N_p$. In this way, potentially long patterns are encoded once in $\mathcal{P}$ and referenced with only $\log_2(N_p)$ bits, instead of replicating the representation. Since partial color sets are encoded once, the total number of integers in $\mathcal{P}$ is at most that in the original $\mathcal{C}$, but in practice it is much smaller.

Each color set P_{ij} can be encoded more succinctly thanks to the permutation π. By placing the colors belonging to the same partition consecutively, the integers in $\mathcal{N}_i$ are lower-bounded by $B_{i-1} + 1$ and upper-bounded by B_i. Therefore, only $\log_2(B_i - B_{i-1})$ bits per color are sufficient. Since the partial sets are encoded using the original Fulgor encoding (Section 4.6.2), the actual space required on average for each integer is much lower.

It is efficient to recover the original color set C_t from the meta color set M_t: for each meta color $(i, j) \in M_t$, sum B_{i-1} back to each decoded integer of P_{ij}. Hence, we decode strictly increasing integers, again thanks to the permutation π. Note that the representation of partial and meta color sets could be described without the permutation π, but this would sacrifice space, as explained above, and query time, since decoding a set would eventually require sorting the decoded integers.

Vertical partitioning opens the possibility to achieve even faster query times than a traditional c-dBG, by using a two-level intersection algorithm for pseudoalignment. First, only meta color sets are intersected, without accessing the partial color sets, to determine the common partitions. Then, only the common partitions are considered, where there are two cases. If the meta color is the same for all color sets being intersected the result is implicit, as it is the partial color sets. Otherwise, if the meta color is not the same, the intersection between partial color sets must be computed. It follows that the optimization is beneficial if the color sets being intersected have very few partitions in common or share many meta colors.

5.3.3 Experimental Results

Similarly to Section 5.2.3, we applied these vertical partitioning techniques to Fulgor, to create **m-Fulgor** (i.e. *meta Fulgor*). In this section, we will compare m-Fulgor against Fulgor and d-Fulgor, to better understand the advantages and disadvantages of both representations.

Index size

Table 10 reports the total index size of Fulgor compared to m-Fulgor. The results show that vertical partitioning has an even greater improvement than horizontal partitioning over the space required to store the color sets. SE-150k only takes 5.27GB, for a remarkable space reduction of 13×. Observe that m-Fulgor, despite offering the best compression on all EC and SE datasets over d-Fulgor, actually performs worse on the GB pangenome (9.16GB vs. 7.51GB). This suggests that vertical partitioning works better when the sets being represented are big, while the opposite is true for horizontal partitioning.

Dataset	dBG	Fulgor			d-Fulgor			m-Fulgor		
		Color sets		Total	Color sets		Total	Color sets		Total
EC	0.29	**1.36**	(83%)	1.65	**0.45**	(61%)	0.74	**0.40**	(58%)	0.69
SE-5k	0.16	**0.59**	(79%)	0.75	**0.20**	(56%)	0.36	**0.16**	(50%)	0.32
SE-10k	0.35	**1.66**	(83%)	2.01	**0.48**	(58%)	0.83	**0.34**	(49%)	0.70
SE-50k	1.25	**17.03**	(93%)	18.29	**4.31**	(77%)	5.57	**2.08**	(62%)	3.34
SE-100k	1.71	**40.71**	(96%)	42.43	**9.37**	(84%)	11.10	**3.75**	(68%)	5.47
SE-150k	2.02	**68.61**	(97%)	70.65	**15.73**	(89%)	17.77	**5.27**	(72%)	7.31
GB	21.29	**15.54**	(42%)	36.83	**7.51**	(26%)	28.81	**9.16**	(30%)	30.46

Table 10: Index spaces in GB, broken down to space required for indexing the k-mers in the dBG (equal for both Fulgor and m-Fulgor), and the data structures needed to encode the color sets and map them to the k-mers. In gray the percentage of space taken by the color sets with respect to the total.

Dataset	Hit rate	Fulgor		d-Fulgor		m-Fulgor	
		mm:ss	GB	h:mm:ss	GB	mm:ss	GB
EC	98.99%	**2:10**	1.67	**5:20**	0.78	**2:30**	0.73
SE-5k	89.49%	**1:10**	0.80	**2:00**	0.41	**1:16**	0.37
SE-10k	89.71%	**2:20**	2.06	**4:30**	0.90	**2:28**	0.77
SE-50k	91.25%	**12:00**	18.24	**29:00**	5.82	**13:10**	3.64
SE-100k	91.41%	**24:00**	42.20	**1:02:00**	11.58	**27:00**	6.08
SE-150k	91.52%	**37:00**	70.55	**1:38:00**	18.51	**41:30**	8.29
GB	92.91%	**1:10**	36.01	**1:00**	28.17	**1:09**	29.79

Table 11: Total query time and memory used by the process as reported by `/usr/bin/time -v`, using 16 threads. For this experiment, the output is written to `/dev/null` to avoid recording I/O overhead.

Query efficiency

Table 11 shows that, using the intersection algorithm described in section 5.3.2, m-Fulgor does not sacrifice query efficiency compared to Fulgor, despite the significant reduction in space.

Therefore, it can be confidently stated that m-Fulgor completely dominates the original Fulgor index, being much more space-efficient but equally fast to query.

Construction time and space

As for the representative/differential variant, the m-Fulgor index was built from an existing Fulgor index. Hence, the times in the fourth column of Table 12 must be summed to the second column to obtain the total construction time.

Construction time may be the biggest flaw of vertical partitioning, taking almost 8 hours to build the SE-150k index, which is 4× slower than its horizontal counterpart. The issue arises from the clustering algorithm used to partition the

Dataset	Fulgor		d-Fulgor		m-Fulgor	
	h:mm	GB	h:mm	GB	h:mm	GB
EC	0:06	17	+0:12	11	+0:05	3
SE-5k	0:04	13	+0:06	8	+0:04	1
SE-10k	0:09	24	+0:09	14	+0:10	3
SE-50k	1:13	44	+0:43	105	+1:50	22
SE-100k	2:56	74	+1:20	207	+4:37	48
SE-150k	4:36	137	+1:55	305	+7:41	77
GB	2:27	115	+10:00	182	+0:31	69

Table 12: Total index construction time and GB of memory used during construction, as reported by `/usr/bin/time -v`, using 48 threads. The times include the time to serialize the index on the disk.

color sets, even if the amount of points (the number of colors N) is orders of magnitude smaller compared to the quantity needed to construct d-Fulgor (i.e. the number of color sets z). Indeed, following the reasoning of Section 5.3.1, having arbitrarily small partitions could result in worse effectiveness, both in terms of space and query time. Thus, the tested implementation imposes a lower bound to the size of the partitions (contrary to d-Fulgor), which makes the clustering algorithm attempt to greedily reassign elements in small clusters to big clusters.

5.4 Combined Partitioning: Meta-Differential Color Sets

In the previous sections, we discussed two solutions to the same problem: representing a collection of sorted integer sets, taking into account repeating patterns of integers. These two solutions are very different, as they are based on orthogonal partitioning paradigms, both with their advantages and disadvantages:

- The representative/differential approach captures patterns formed by colors not necessarily appearing in consecutive positions. Partial/meta color sets, on the other hand, need to permute the reference identifiers to be effective, reducing the number of bits to encode the integers.

- Both methods add an extra layer of indirection when decoding a color set C_i, compared to a representation that encodes each set individually. Representative/differential color sets must be read in parallel to decode the color set C_i, possibly reading more than $|C_i|$ integers in the process, while partial/meta color sets decode exactly $|C_i|$.

- As reported in Sections 5.2.3 and 5.3.3, partial/meta indices are generally best in space effectiveness and query efficiency, but their construction is much slower than representative/differential indices.

These observations suggest that vertical partitioning has a net advantage over horizontal partitioning. It might argued that, since partial/meta color sets are so effective, a recursive approach could yield even better results. However, such reasoning is fundamentally incorrect, as the hypothetical benefits should have been obtained during the "outer" partitioning step.

Proof. Let $\mathcal{N} = \{\mathcal{N}_1, ..., \mathcal{N}_r\}$ be a partition of $[q] = \{1, ..., q\}$ obtained using the clustering algorithm A. Let $\mathcal{N}' = \{\mathcal{N}'_1, ..., \mathcal{N}'_r\}$ be the result of a recursive partition step, where each $\mathcal{N}_i$ is further split into $\mathcal{N}'_i = \{\mathcal{N}_{i1}, ..., \mathcal{N}_{ir_i}\}$, and such that $\textsc{Cost}(\mathcal{N}') < \textsc{Cost}(\mathcal{N})$. The sub-partitions $\mathcal{N}'_i$ can be concatenated together, yielding a new partition $\mathcal{N}' = \{\mathcal{N}_{11}, \mathcal{N}_{12}, ..., \mathcal{N}_{ij}, ...\}$ of $[q]$, with size $\sum_{i=1}^{r} r_i$. This new representation, however, can be obtained directly from $[q]$ using another clustering algorithm A', meaning that the result of any recursive partitioning step can always be obtained with a different clustering algorithm. $\square$

Nonetheless, an even improved representation can still be achieved when the two models are combined. In fact, both the set Δ of differential color sets and the set $\mathcal{P}$ of partial color sets are collections of sorted integer sets, in the same way as $\mathcal{C}$. We consider the two following scenarios:

1. the set Δ is encoded with meta/partial color sets;

2. the set $\mathcal{P}$ is encoded with representative/differential color sets.

The former combination is not promising, as the differential color sets in each partition are expected to be very different from each other since shared patterns are captured in the representative color sets. This is apparent in the example from Figure 8, where the intersection of any two sets in the same partition is always empty. The latter, instead, has good potential as the partial color sets belonging to the same partition tend to be very similar. This is also true in the example in Figure 9b.

5.4.1 Analysis and Implementation Details

The meta-differential index is structured as an m-Fulgor index, where the partial color sets are compressed using the representative/differential approach. See Figure 10 for an example.

Also, since this representation focuses most on space efficiency, we devised a different representation of the meta-colors to improve compression even further. Recall that each meta color is a pair (i, j), where i determines the partition and j

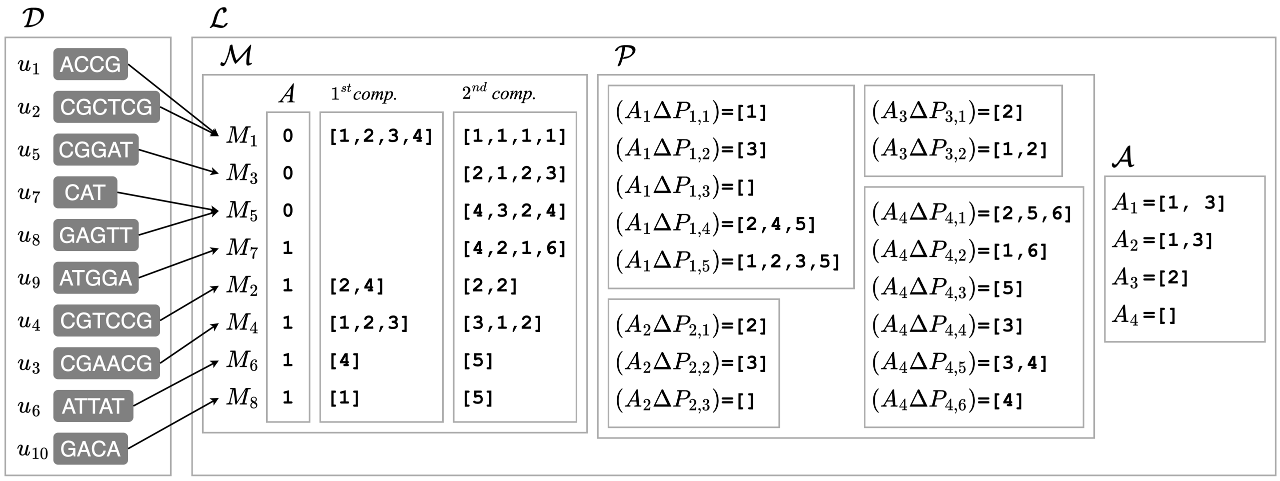


Figure 10: The meta-differential dBG built from the color sets in Figure 3

the offset of the partial color set P_{ij}. A list of meta colors $M_t = [(i_1, j_1), (i_2, j_2), ...]$ can therefore be decomposed into two integer lists, $[i_1, i_2, ...]$ and $[j_1, j_2, ...]$. We will refer to the lists as *first* and *second* components of the meta color, respectively. For example, the meta color list $M_3 = [(1, 2), (2, 1), (3, 2), (4, 3)]$ can be split into its first component $[1, 2, 3, 4]$ and second component $[2, 1, 2, 3]$. Note that, by construction, the first list is always sorted in strictly increasing order, while the second is not necessarily ordered and might contain duplicates.

The first observation is that — especially when the number of partitions is small — the number f of the distinct first component lists is small compared to the total number of color sets z. For example, in Figure 9b, meta color sets M_1, M_3, M_5, and M_7 all share the same first component list. Hence, it is possible to store the distinct first components in an array $A[1..f]$ and, for each meta color set, specify the index of the array entry corresponding to the first component. This can be done efficiently by sorting the lists in $\mathcal{M}$ according to their first component and implementing the map $\mathcal{M} \to [1..f]$ using a bitvector $b[1..z]$ and the RANK query as explained in Fact 1. The first component of the meta color M_t is thus $A[p]$, with $p = \text{RANK}_1(b, t) + 1$.

The second components are instead less repetitive, and their integers do not follow any particular pattern. For this reason, we use the following variable-length encoding: given the meta color (i, j) we encode the integer j using $\log_2(|\mathcal{P}_i|)$ bits[1]. The size of each partition is stored in an integer array for easy and fast access. Completing the example, the second component of M_3, $[2, 1, 2, 3]$ is encoded using

[1]For ease of notation we write $\log_2(x)$ instead of $\lfloor \log_2(x-1) \rfloor + 1$ for $x \geq 1$ and assume that $\log_2(0) = 0$.

Dataset	dBG	Fulgor			d-Fulgor			m-Fulgor			md-Fulgor		
		Color sets		Total	Color sets		Total	Color sets		Total	Color sets		Total
EC	0.29	**1.36**	(83%)	1.65	**0.45**	(61%)	0.74	**0.40**	(58%)	0.69	**0.24**	(45%)	0.52
SE-5k	0.16	**0.59**	(79%)	0.75	**0.20**	(56%)	0.36	**0.16**	(50%)	0.32	**0.11**	(40%)	0.27
SE-10k	0.35	**1.66**	(83%)	2.01	**0.48**	(58%)	0.83	**0.34**	(49%)	0.70	**0.22**	(39%)	0.57
SE-50k	1.25	**17.03**	(93%)	18.29	**4.31**	(77%)	5.57	**2.08**	(62%)	3.34	**1.38**	(52%)	2.64
SE-100k	1.71	**40.71**	(96%)	42.43	**9.37**	(84%)	11.10	**3.75**	(68%)	5.47	**2.26**	(57%)	3.98
SE-150k	2.02	**68.61**	(97%)	70.65	**5.73**	(89%)	17.77	**5.27**	(72%)	7.31	**3.22**	(61%)	5.26
GB	21.29	**15.54**	(42%)	36.83	**7.51**	(26%)	28.81	**9.16**	(30%)	30.46	**6.19**	(23%)	27.48

Table 13: Index spaces in GB, broken down to space required for indexing the k-mers in the dBG (equal for both Fulgor and md-Fulgor), and the data structures needed to encode the color sets and map them to the k-mers. In gray the percentage of space taken by the color sets with respect to the total.

$\log_2(|\mathcal{P}_1|) + \log_2(|\mathcal{P}_2|) + \log_2(|\mathcal{P}_2|) + \log_2(|\mathcal{P}_4|)$ bits, that is $\log_2(5) + \log_2(3) + \log_2(2) + \log_2(6) = 3 + 2 + 1 + 3 = 9$ bits. Thus, $[2, 1, 2, 3]$ is encoded as

2	1	2	3
1	0	1	10
001	00	1	010

$$\downarrow$$

001.00.1.010

Note that the binary representation of each integer x (second row) is actually $x-1$.

5.4.2 Experimental results

Index size

Table 13 reports the total index size of Fulgor compared to md-Fulgor. The results show that combining the two schemes leads to a massive improvement in the color sets space effectiveness. For SE-150k, the size of the color sets in the original Fulgor is almost 69GB, while it only takes a little more than 3GB in the meta-differential encoding, translating to a space reduction of more than 21.3×. Figure 11 also visually shows the relative sizes of the various index parts (i.e. dictionary and color sets) for each Fulgor variant on this dataset.

The GB dataset is also smaller than the d-Fulgor variant (6.19GB vs. 7.51GB), further proving that the combination of vertical and horizontal partitioning exploits the advantages of both paradigms.

It can be confidently said that the md-Fulgor variant is the most succinct representation of a de Bruijn Graph up to date.

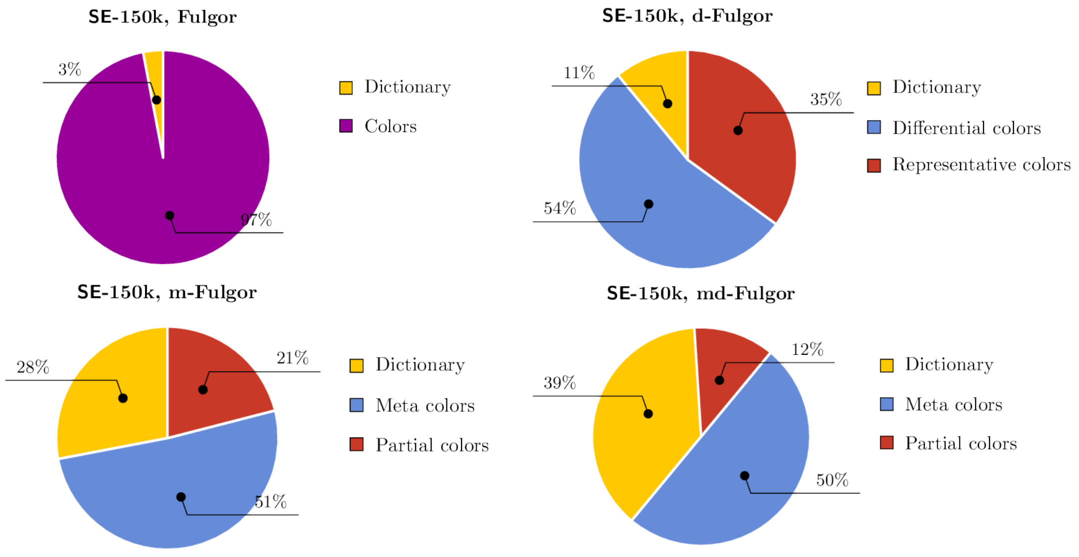


Figure 11: Index space breakdown of the SE-150k dataset for every Fulgor variant. Note that the space taken by the dictionary (in yellow) is constant, but its percentage increases as the space taken by the colors is progressively smaller.

Dataset	Hit rate	Fulgor		d-Fulgor		m-Fulgor		md-Fulgor	
		mm:ss	GB	h:mm:ss	GB	mm:ss	GB	h:mm:ss	GB
EC	98.99%	**2:10**	1.67	**5:20**	0.78	**2:30**	0.73	**5:00**	0.57
SE-5k	89.49%	**1:10**	0.80	**2:00**	0.41	**1:16**	0.37	**1:48**	0.32
SE-10k	89.71%	**2:20**	2.06	**4:30**	0.90	**2:28**	0.77	**3:34**	0.65
SE-50k	91.25%	**12:00**	18.24	**29:00**	5.82	**13:10**	3.64	**22:25**	2.95
SE-100k	91.41%	**24:00**	42.20	**1:02:00**	11.58	**27:00**	6.08	**50:00**	4.62
SE-150k	91.52%	**37:00**	70.55	**1:38:00**	18.51	**41:30**	8.29	**1:15:00**	6.28
GB	92.91%	**1:10**	36.01	**1:00**	28.17	**1:09**	29.79	**1:03**	26.88

Table 14: Total query time and memory used by the process as reported by `/usr/bin/time -v`, using 16 threads. For this experiment, the output is written to `/dev/null` to avoid recording I/O overhead.

Query efficiency

Table 14 shows that the combination of vertical and horizontal partitioning partially mitigates the slowdown given by d-Fulgor. Overall, the md-Fulgor index is more than an order of magnitude smaller than the original Fulgor, while only being about 2× slower. We consider this space/time trade-off to be more than acceptable for the sake of indexing larger c-dBGs in internal memory. Also note that md-Fulgor is still 1.7× faster than Themisto (e.g. for SE-150k, 1.25 hours vs. 2 hours), that is the next fastest index in the literature.

Dataset	Fulgor		d-Fulgor		m-Fulgor		md-Fulgor	
	h:mm	GB	h:mm	GB	h:mm	GB	h:mm	GB
EC	0:06	17	+0:12	11	+0:05	3	+0:19 (0:14)	4
SE-5k	0:04	13	+0:06	8	+0:04	1	+0:09 (0:05)	3
SE-10k	0:09	24	+0:09	14	+0:10	3	+0:21 (0:11)	4
SE-50k	1:13	44	+0:43	105	+1:50	22	+2:20 (0:30)	13
SE-100k	2:56	74	+1:20	207	+4:37	48	+5:20 (0:43)	20
SE-150k	4:36	137	+1:55	305	+7:41	77	+8:36 (0:55)	25
GB	2:27	115	+10:00	182	+0:31	69	+7:43 (7:12)	127

Table 15: Total index construction time and GB of memory used during construction, as reported by `/usr/bin/time -v`, using 48 threads. The times include the time to serialize the index on the disk. Inside the parentheses, the time taken by the horizontal partitioning step.

Construction time

For these experiments, md-Fulgor was built from an already existing m-Fulgor index. The times reported in Table 15 are the total times to build the indices — starting from Fulgor — while the ones between parentheses represent the time only of the horizontal partitioning step for md-Fulgor.

Compared to the other indices (Section 4.7), the construction times of the Fulgor variants are competitive to the ones of Themisto and much faster than MetaGraph.

Chapter 6

Conclusions and Future Work

In this work, we introduced new compressed representations for the colored de Bruijn graph, in which repetitive patterns within color sets are encoded once to improve the memory usage of pseudoalignment queries. The resulting compression algorithms have been applied to the recently proposed Fulgor index, as it possesses the best space vs. time trade-off. In particular, these representations focus on two distinct and essentially opposite approaches for factorizing and compressing redundant patterns in the color sets: **d-Fulgor** is a horizontal compression method that encodes color sets into representative and differential sets, conversely, **m-Fulgor** is a vertical compression method that creates a two-layered set representation through meta and partial color sets. These methods exploit different characteristics for compression, so they can be combined to achieve an even better compression of the color sets. This combination is represented by the **md-Fulgor** index.

After extensive experimental analyses across multiple and varied datasets to evaluate the different schemes, we compared the results against alternative c-dBG representations. From this, we conclude that:

- m-Fulgor does not introduce any tradeoffs compared to the original Fulgor index and can simply replace it, as it is equally fast but significantly smaller.
- md-Fulgor is even more compact, with a relatively minor query overhead over Fulgor, especially considering the space reduction it provides.

These new representations provide a new reference point for the problem of indexing c-dBGs, as shown in Figure 12. md-Fulgor is competitive with the smallest variant of MetaGraph while still being more than an order of magnitude faster. It is also up to 20× smaller than Themisto, but still faster on the great majority of the datasets.

We believe that this improved performance has the potential to enable large-scale color set queries across multiple applications.

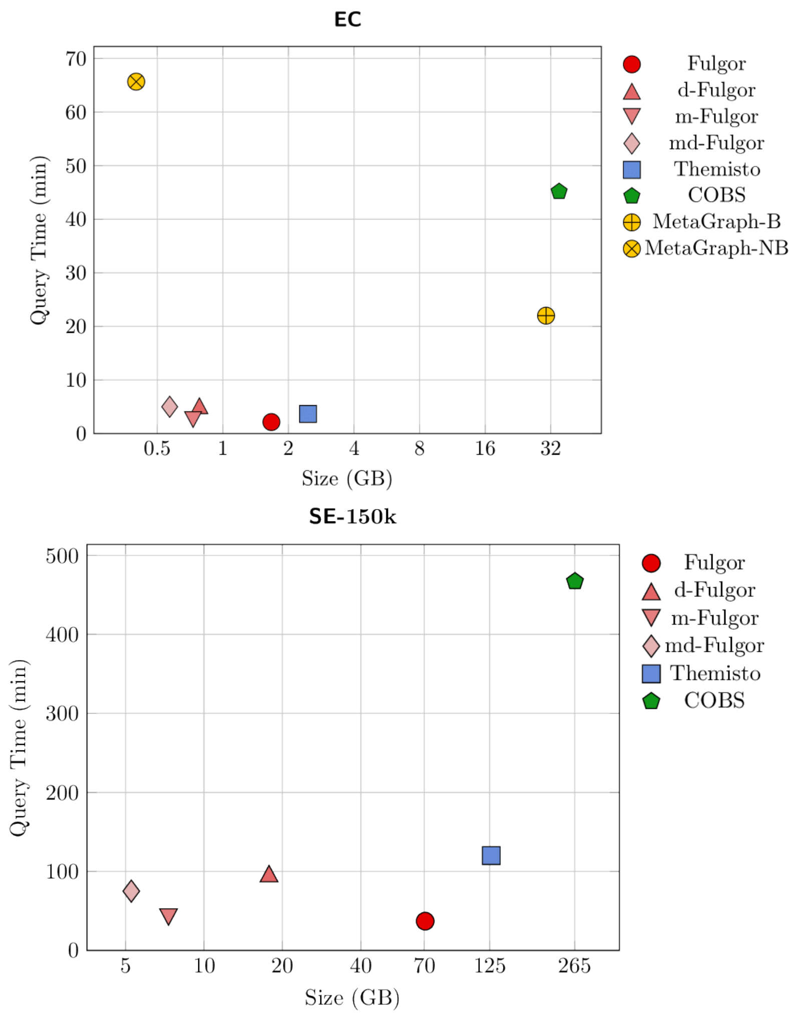


Figure 12: Data from Tables 5 and 14 combined and shown as space/time plots, for the EC and SE-150k datasets. Note that the horizontal axis (Size) is logarithmic.

Future work

Future work will focus on improving the index space effectiveness and the time efficiency of its relevant operations:

- Accelerate pseudoalignment queries, in particular for d-Fulgor and md-Fulgor.
- Provide a better build pipeline. At the moment the building process is not deeply engineered, but we believe that some speedups and usability refinements are still possible.

- Improve d-Fulgor construction. When performing horizontal partitioning the size of the sketches and the slicing thresholds are predefined values. Finding a way to dynamically determine both values based on the properties of the input dataset could lead to better compression and faster build times.

- Explore the effects of approximately optimal ordering within partial and differential color sets, that is, the last step of the SCPO framework. Indeed, reassigning the color identifiers so that repeating patterns are made up of integers whose difference is very small, can significantly reduce color sets size.

- Extend the indexing capabilities of Fulgor by annotating the graph with more information, like k-mer abundances and their positions in the references.

Bibliography

[1] Alessio Campanelli et al. “Where the patterns are: repetition-aware compression for colored de Bruijn Graphs.” In: *Journal of Computational Biology* (2024, to appear).

[2] Felix Gabler et al. “Protein sequence analysis using the MPI bioinformatics toolkit”. In: *Current Protocols in Bioinformatics* 72.1 (2020), e108.

[3] David Kovalic et al. “The use of next generation sequencing and junction sequence analysis bioinformatics to achieve molecular characterization of crops improved through modern biotechnology”. In: *The Plant Genome* 5.3 (2012).

[4] Martin Blueggel, Daniel Chamrad, and Helmut E Meyer. “Bioinformatics in proteomics”. In: *Current pharmaceutical biotechnology* 5.1 (2004), pp. 79–88.

[5] Mauno Vihinen. “Bioinformatics in proteomics”. In: *Biomolecular engineering* 18.5 (2001), pp. 241–248.

[6] Xuhua Xia. “Bioinformatics and drug discovery”. In: *Current topics in medicinal chemistry* 17.15 (2017), pp. 1709–1726.

[7] Sarah K Wooller et al. “Bioinformatics in translational drug discovery”. In: *Bioscience reports* 37.4 (2017), BSR20160180.

[8] Casey Lynnette Overby and Peter Tarczy-Hornoch. “Personalized medicine: challenges and opportunities for translational bioinformatics”. In: *Personalized medicine* 10.5 (2013), pp. 453–462.

[9] Guy Haskin Fernald et al. “Bioinformatics challenges for personalized medicine”. In: *Bioinformatics* 27.13 (2011), pp. 1741–1748.

[10] Jordan M Eizenga et al. “Pangenome graphs”. In: *Annual review of genomics and human genetics* 21.1 (2020), pp. 139–162.

[11] Peter Elias. “Universal codeword sets and representations of the integers”. In: *IEEE transactions on information theory* 21.2 (1975), pp. 194–203.

[12] Guy Joseph Jacobson. *Succinct static data structures.* Carnegie Mellon University, 1988.

[13] David Clark. "Compact pat trees". In: (1997).

[14] Gonzalo Navarro. *Compact data structures: A practical approach.* Cambridge University Press, 2016.

[15] Peter Elias. "Efficient storage and retrieval by content and address of static files". In: *Journal of the ACM (JACM)* 21.2 (1974), pp. 246–260.

[16] Robert Mario Fano. *On the number of bits required to implement an associative memory.* Massachusetts Institute of Technology, Project MAC, 1971.

[17] Frederick Sanger, Steven Nicklen, and Alan R Coulson. "DNA sequencing with chain-terminating inhibitors". In: *Proceedings of the national academy of sciences* 74.12 (1977), pp. 5463–5467.

[18] I Illumina. "An introduction to next-generation sequencing technology". In: *Illumina, Inc* (2015).

[19] Anthony Rhoads and Kin Fai Au. "PacBio sequencing and its applications". In: *Genomics, Proteomics and Bioinformatics* 13.5 (2015), pp. 278–289.

[20] Miten Jain et al. "The Oxford Nanopore MinION: delivery of nanopore sequencing to the genomics community". In: *Genome biology* 17 (2016), pp. 1–11.

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

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

[23] Lifu Song et al. "Robust data storage in DNA by de Bruijn graph-based de novo strand assembly". In: *Nature communications* 13.1 (2022), p. 5361.

[24] Leena Salmela et al. "Accurate self-correction of errors in long reads using de Bruijn graphs". In: *Bioinformatics* 33.6 (2017), pp. 799–806.

[25] Ilia Minkin, Son Pham, and Paul Medvedev. "TwoPaCo: an efficient algorithm to build the compacted de Bruijn graph from many complete genomes". In: *Bioinformatics* 33.24 (2017), pp. 4024–4032.

[26] Burton H Bloom. "Space/time trade-offs in hash coding with allowable errors". In: *Communications of the ACM* 13.7 (1970), pp. 422–426.

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

[28] Michael Roberts et al. "Reducing storage requirements for biological sequence comparison". In: *Bioinformatics* 20.18 (2004), pp. 3363–3369.

[29] Saul Schleimer, Daniel S Wilkerson, and Alex Aiken. “Winnowing: local algorithms for document fingerprinting”. In: *Proceedings of the 2003 ACM SIGMOD international conference on Management of data*. 2003, pp. 76–85.

[30] Jamshed Khan et al. “Scalable, ultra-fast, and low-memory construction of compacted de Bruijn graphs with Cuttlefish 2”. In: *Genome biology* 23.1 (2022), p. 190.

[31] Antoine Limasset et al. “Fast and scalable minimal perfect hashing for massive key sets”. In: *arXiv preprint arXiv:1702.03154* (2017).

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

[33] Giulio Ermanno Pibiri and Rossano Venturini. “Techniques for inverted index compression”. In: *ACM Computing Surveys (CSUR)* 53.6 (2020), pp. 1–36.

[34] Antoine Limasset, Jean-François Flot, and Pierre Peterlongo. “Toward perfect reads: self-correction of short reads via mapping on de Bruijn graphs”. In: *Bioinformatics* 36.5 (2020), pp. 1374–1381.

[35] Bo Liu et al. “deBGA: read alignment with de Bruijn graph-based seed and extension”. In: *Bioinformatics* 32.21 (2016), pp. 3224–3232.

[36] Mark Reppell and John Novembre. “Using pseudoalignment and base quality to accurately quantify microbial community composition”. In: *PLoS computational biology* 14.4 (2018), e1006096.

[37] Daniel R Zerbino and Ewan Birney. “Velvet: algorithms for de novo short read assembly using de Bruijn graphs”. In: *Genome research* 18.5 (2008), pp. 821–829.

[38] Zamin Iqbal et al. “De novo assembly and genotyping of variants using colored de Bruijn graphs”. In: *Nature genetics* 44.2 (2012), pp. 226–232.

[39] Guillaume J Filion, Ruggero Cortini, and Eduard Zorita. “Calibrating seed-based heuristics to map short reads with sesame”. In: *Frontiers in Genetics* 11 (2020), p. 572.

[40] Nicolas L Bray et al. “Near-optimal probabilistic RNA-seq quantification”. In: *Nature biotechnology* 34.5 (2016), pp. 525–527.

[41] Jarno N Alanko et al. “Themisto: a scalable colored k-mer index for sensitive pseudoalignment against hundreds of thousands of bacterial genomes”. In: *Bioinformatics* 39.Supplement_1 (2023), pp. i260–i269.

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

[43] Fatemeh Almodaresi et al. “An efficient, scalable, and exact representation of high-dimensional color information enabled using de Bruijn graph search”. In: *Journal of Computational Biology* 27.4 (2020), pp. 485–499.

[44] Prashant Pandey et al. “A general-purpose counting filter: Making every bit count”. In: *Proceedings of the 2017 ACM international conference on Management of Data.* 2017, pp. 775–787.

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

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

[47] Guillaume Holley and Páll Melsted. “Bifrost: highly parallel construction and indexing of colored and compacted de Bruijn graphs”. In: *Genome biology* 21 (2020), pp. 1–20.

[48] Daniel Lemire et al. “Roaring bitmaps: Implementation of an optimized software library”. In: *Software: Practice and Experience* 48.4 (2018), pp. 867–895.

[49] Mikhail Karasikov et al. “Sparse binary relation representations for genome graph annotation”. In: *Journal of Computational Biology* (2020).

[50] Alexander Bowe et al. “Succinct de Bruijn graphs”. In: *International workshop on algorithms in bioinformatics.* Springer. 2012, pp. 225–235.

[51] Paolo Ferragina et al. “Compressing and indexing labeled trees, with applications”. In: *Journal of the ACM (JACM)* 57.1 (2009), pp. 1–33.

[52] Michael Burrows et al. “A block-sorting lossless data compression algorithm. 1994”. In: *Systems Research Center, Palo Alto* (1994).

[53] Gonzalo Navarro. “Wavelet trees for all”. In: *Journal of Discrete Algorithms* 25 (2014), pp. 2–20.

[54] Jarno N Alanko, Simon J Puglisi, and Jaakko Vuohtoniemi. “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.

[55] Giulio Ermanno Pibiri. “Sparse and skew hashing of k-mers”. In: *Bioinformatics* 38.Supplement_1 (2022), pp. i185–i194.

[56] Jason Fan et al. “Fulgor: a fast and compact k-mer index for large-scale matching and color queries”. In: *Algorithms for Molecular Biology* 19.1 (2024), p. 3.

[57] Johan Ludwig William Valdemar Jensen. “Sur les fonctions convexes et les inégalités entre les valeurs moyennes”. In: *Acta mathematica* 30.1 (1906), pp. 175–193.

[58] Jarno N. Alanko. *3682 E. Coli Assemblies from NCBI*. 2022. URL: https://zenodo.org/records/6577997.

[59] Grace A. Blackwell et al. “Exploring bacterial diversity via a curated and searchable snapshot of archived DNA sequences”. In: *PLOS Biology* 19.11 (Nov. 2021), pp. 1–16. URL: http://ftp.ebi.ac.uk/pub/databases/ENA2018-bacteria-661k.

[60] Pranvera Hiseni et al. “HumGut: a comprehensive human gut prokaryotic genomes collection filtered by metagenome data”. In: *Microbiome* 9.1 (2021), pp. 1–12. URL: https://arken.nmbu.no/~larssn/humgut/index.htm.

[61] Philippe Flajolet et al. “Hyperloglog: the analysis of a near-optimal cardinality estimation algorithm”. In: *Discrete mathematics & theoretical computer science* Proceedings (2007).

[62] Laxman Dhulipala et al. “Compressing graphs and indexes with recursive graph bisection”. In: *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. 2016, pp. 1535–1544.