Meta-colored compacted de Bruijn graphs

Giulio Ermanno Pibiri[1,2][0000−0003−0724−7092], Jason Fan[3][0000−0001−7617−4814], and Rob Patro[3][0000−0001−8463−1675]

[1] DAIS, Ca' Foscari University of Venice, Venice, Italy
[2] ISTI-CNR, Pisa, Italy
[3] Department of Computer Science, University of Maryland, College Park, MD 20440, USA

Abstract. The colored compacted de Bruijn graph (c-dBG) has become a fundamental tool used across several areas of genomics and pangenomics. For example, it has been widely adopted by methods that perform read mapping or alignment, abundance estimation, and subsequent downstream analyses. These applications essentially regard the c-dBG as a map from k-mers to the set of references in which they appear. The c-dBG data structure should retrieve this set — the *color* of the k-mer — efficiently for any given k-mer, while using little memory. To aid retrieval, the colors are stored explicitly in the data structure and take considerable space for large reference collections, even when compressed. Reducing the space of the colors is therefore of utmost importance for large-scale sequence indexing.

We describe the *meta-colored* compacted de Bruijn graph (Mac-dBG) — a new colored de Bruijn graph data structure where colors are represented holistically, i.e., taking into account their redundancy across the whole collection being indexed, rather than individually as atomic integer lists. This allows the factorization and compression of common sub-patterns across colors. While optimizing the space of our data structure is NP-hard, we propose a simple heuristic algorithm that yields practically good solutions. Results show that the Mac-dBG data structure improves substantially over the best previous space/time trade-off, by providing remarkably better compression effectiveness for the same (or better) query efficiency. This improved space/time trade-off is robust across different datasets and query workloads.

Code availability. A `C++17` implementation of the Mac-dBG is publicly available on GitHub at: https://github.com/jermp/fulgor.

1 Introduction

The colored compacted de Bruijn graph (c-dBG) has become a fundamental tool used across several areas of genomics and pangenomics. For example, it has been widely adopted by methods that perform read mapping or alignment, specifically with respect to RNA-seq and metagenomic identification and abundance estimation [21,8,33,3,32,23,4,34]; among methods that perform homology assessment and mapping of genomes [26,27]; for a variety of different tasks in

pangenome analysis [9,24,10,20,22], and for storage and compression of genomic data [31]. In most of these applications, a key requirement of the underlying representation of the c-dBG is to be able to determine — with efficiency being critical — the set of references in which an individual k-mer appears. These motivations bring us to the following problem formulation.

Problem 1 (Colored k-mer indexing). Let $\mathcal{R} = \{R_1, \ldots, R_N\}$ be a collection of 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 (referred to as the *index* in the following) that allows us to retrieve the set $\textsc{Color}(x) = \{i | x \in R_i\}$ as efficiently as possible for any k-mer $x \in \Sigma^k$. If the k-mer x does not occur in any reference, we say that $\textsc{Color}(x) = \varnothing$. Hereafter, we simply refer to the set $\textsc{Color}(x)$ as the *color* of the k-mer x.

Of particular importance for biological analysis is the case where $\mathcal{R}$ is a *pangenome*. Roughly speaking, a pangenome is a (large) set of genomes in a particular population, species or closely-related phylogenetic group. Pangenomes have revolutionized DNA analysis by providing a more comprehensive understanding of genetic diversity within a species [25,5]. Unlike traditional reference genomes, which represent a single individual or a small set of individuals, pangenomes incorporate genetic information from multiple individuals within a species or group. This approach is particularly valuable because it captures a wide range of genetic variations, including rare and unique sequences that may be absent from any particular reference genome.

Contributions. The goal of this paper is to propose a solution to Problem 1 focusing on the specific, important, application scenario where $\mathcal{R}$ is a pangenome. (We note, however, that the approaches described herein are general, and we expect them to work well on any corpus of highly-related genomes, whether or not they formally constitute a pangenome.) To best exploit the properties of Problem 1, we capitalize on recent indexing development for c-dBGs [13]. The result is the *meta-colored* compacted de Bruijn graph (Mac-dBG) — a new data structure where colors are represented *holistically*, i.e., taking into account their redundancy across the whole collection being indexed, rather than individually as atomic integer lists.

After covering preliminary concepts in Section 2, we describe the Mac-dBG in Section 3.1 and 3.2. We present the underlying NP-hard optimization problem in Section 3.3 and discuss a simple framework for constructing the Mac-dBG in Section 3.4. Section 4 presents experimental results to demonstrate that the Mac-dBG remarkably improves the best previous space/time trade-off in the literature. In fact, it essentially combines the space effectiveness of the most compact solutions with the query efficiency of the fastest solutions, at the expense of a slower construction algorithm. We conclude in Section 5.

A `C++17` implementation of the Mac-dBG is available at: https://github.com/jermp/fulgor.

2 Preliminaries: modular indexing of colored compacted de Bruijn graphs

In this section we provide some background information to better understand the design principles of the solution we propose in Section 3.

In principle, Problem 1 could be solved using a classic data structure from Information Retrieval — the *inverted index* [30]. In the context of this problem, the indexed documents are the references $\{R_1, \ldots, R_N\}$ in the collection $\mathcal{R}$ and the terms of the inverted index are all distinct k-mers of $\mathcal{R}$. Using the notation from Problem 1, it follows that $\textsc{Color}(x)$ is the inverted list of the term x. Let $\mathcal{L}$ denote the inverted index for $\mathcal{R}$. The inverted index $\mathcal{L}$ explicitly stores the ordered set $\textsc{Color}(x)$ for each k-mer $x \in \mathcal{R}$. The goal is to implement the map $x \rightarrow \textsc{Color}(x)$ as efficiently as possible in terms of both memory usage and query time. To this end, all the distinct k-mers of $\mathcal{R}$ are stored in an *associative* dictionary data structure $\mathcal{D}$. Suppose we have n distinct k-mers in $\mathcal{R}$. These k-mers are stored losslessly in $\mathcal{D}$. To implement the map $x \rightarrow \textsc{Color}(x)$, $\mathcal{D}$ is required to support the operation $\textsc{Lookup}(x)$, which returns $\perp$ if k-mer x is not found in the dictionary or a unique integer identifier in $[n] = \{1, \ldots, n\}$ if x is found. Problem 1 can then be solved using these two data structures — $\mathcal{D}$ and $\mathcal{L}$ — thanks to the interplay between $\textsc{Lookup}(x)$ and $\textsc{Color}(x)$: logically, the index stores the sets $\{\textsc{Color}(x)\}_{x \in \mathcal{R}}$ in some compressed form, *sorted by* the value of $\textsc{Lookup}(x)$.

To exploit at best the potential of this modular decomposition into $\mathcal{D}$ and $\mathcal{L}$, it is essential to rely on the specific properties of Problem 1. For example, we know that consecutive k-mers share $(k-1)$-length overlaps; also, k-mers that co-occur in the same set of references have the same color. A useful, standard, formalism that captures these properties is the so-called *colored (compacted) de Bruijn graph* (c-dBG).

Let $\mathcal{K}$ be the set of all the distinct k-mers of $\mathcal{R}$. The node-centric de Bruijn graph (dBG) of $\mathcal{R}$ is a directed graph $G(\mathcal{K}, E)$ whose nodes are the k-mers in $\mathcal{K}$. There is an edge $(u, v) \in E$ if the $(k-1)$-length suffix of u equals the $(k-1)$-length prefix of v. Note that the edge set E is implicitly defined by the set of nodes, and can therefore be omitted from subsequent definitions. We refer to k-mers and nodes in a dBG interchangeably. Likewise, a path in a dBG spells the string obtained by concatenating together all the k-mers along the path, without repeating the shared $(k-1)$-length overlaps. In particular, unary paths (i.e., non-branching) can be collapsed into single nodes spelling strings that are referred to as *unitigs*. Let $\mathcal{U} = \{u_1, \ldots, u_m\}$ be the set of unitigs of the graph. The dBG arising from this compaction step is called the *compacted* dBG, and indicated with $G(\mathcal{U})$.

The *colored* compacted dBG (c-dBG) is obtained by logically annotating each k-mer x with its color, $\textsc{Color}(x)$. While different conventions have been adopted in the literature, here we assume that only non-branching paths with nodes having the *same* color are collapsed into unitigs. The unitigs of the c-dBG we consider in this work have the following key properties.

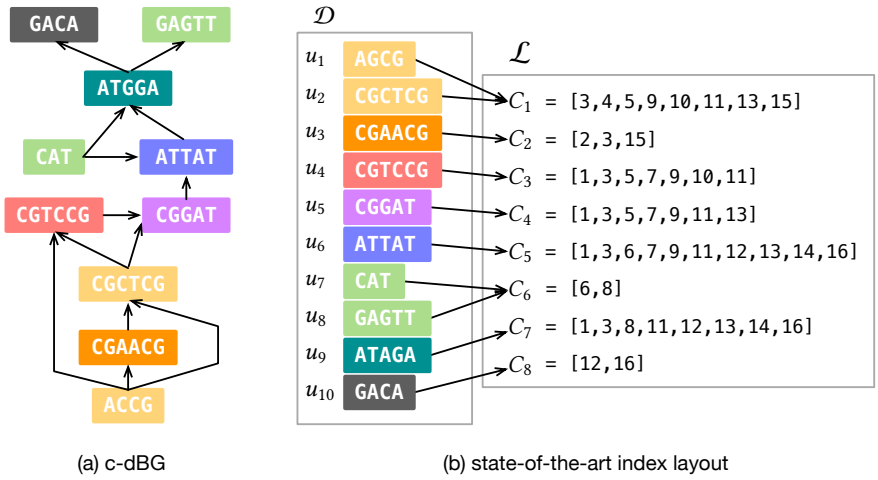


Fig. 1: In panel (a), an example colored compacted de Bruijn graph (c-dBG) for $k = 3$. (In the figure, a k-mer and its reverse complement are considered as different k-mer for ease of illustration. In practice, these are considered identical.) The unitigs of the graph are colored according to the set of references they appear in. In panel (b), we schematically illustrate the state-of-the-art index layout (the Fulgor index [13]) assuming the c-dBG was built for $N = 16$ references, highlighting the modular composition of a k-mer dictionary, $\mathcal{D}$, and an inverted index, $\mathcal{L}$. Note that unitigs are stored in $\mathcal{D}$ in color order, hence allowing a very efficient mapping of k-mers to their distinct colors.

1. *Unitigs spell references in* $\mathcal{R}$. Each distinct k-mer of $\mathcal{R}$ appears once, as sub-string of some unitig of the c-dBG. By construction, each reference $R_i \in \mathcal{R}$ can be spelled out by some *tiling* of the unitigs — an ordered sequence of unitig occurrences that, when glued together (accounting for (k-1)-symbol overlap and orientation), spell R_i [12]. Joining together k-mers into unitigs reduces their storage requirements and accelerates looking up k-mers in consecutive order [28].
2. *Unitigs are monochromatic.* The k-mers belonging to the same unitig u_i all have the same color. We write $x \in u_i$ to indicate that k-mer x is a sub-string of the unitig u_i. Thus, we shall use $\textsc{Color}(u_i)$ to denote the color of each k-mer $x \in u_i$.
3. *Unitigs co-occur.* Distinct unitigs often have the *same* color, i.e., they co-occur in the same set of references, because they derive from conserved sequences in indexed references that are longer than the unitigs themselves. We indicate with z the number of distinct colors $\mathcal{C} = \{C_1, \ldots, C_z\}$. Note that $z \leq m$ and that, in practice, there are almost always *many more* unitigs than there are distinct colors.

Fig. 1a illustrates an example c-dBG with these properties. In the following, we refer to a compacted c-dBG as $G(\mathcal{U}, \mathcal{C})$.

State of the art. To the best of our knowledge, the only solution that exploits *all* three properties is the recently-introduced Fulgor index [13], which we now review since it is the basis of our development in Section 3.

The solution implemented by Fulgor is to first map k-mers to unitigs using the dictionary $\mathcal{D}$, and then succinctly map unitigs to their colors. The colors $\mathcal{C} = \{C_1, \ldots, C_z\}$ themselves are stored in compressed form in an inverted index $\mathcal{L}$. By *composing* these mappings, Fulgor obtains an efficient map directly from k-mers to their associated colors (see also Fig. 1b). The composition is made possible by leveraging the *order-preserving* property of its dictionary data structure — SSHash [28,29] — which explicitly stores the set of unitigs in *any* desired order. This property has some important implications. First, looking up consecutive k-mers is cache-efficient since unitigs are stored contiguously in memory as sequences of 2-bit characters. Second, if k-mer x occurs in unitig u_i, the LOOKUP(x) operation of SSHash can efficiently determine the unitig identifier i, allowing to map k-mers to unitigs. Third, if unitigs are sorted in color order, so that unitigs having the same color are consecutive, then mapping a unitig to its color can be implemented in as little as $1 + o(1)$ bits *per unitig* and in constant time via a RANK query.

3 Meta-colored compacted de Bruijn graphs

When indexing large pangenomes, the space taken by the (compressed) colors dominates the whole index space [13,16,2]. Efforts toward improving the memory usage of c-dBGs should therefore be spent in devising better compression algorithms for the colors. In this work, we focus on exploiting the following crucial property that can enable substantially better compression effectiveness: *The genomes in a pangenome are very similar* which, in turn, implies that the *colors are also very similar* (albeit distinct).

By "similar" colors we mean that they share many (potentially, very long) identical integer sub-sequences. This property is not exploited if each color C_i is compressed *individually* from the other colors. For example, if C_i shares a long sub-sequence with C_j, this sub-sequence is actually represented *twice* in the index, which wastes space. This example is instrumentally simple; yet, it suggests that the identification of such common sub-sequences across a large collection, as well as the design of an effective compression mechanism for these patterns, is not easy. A further complicating matter is that the example clearly generalizes to more than two sub-sequences, hence increasing with pangenome redundancy and aggravating the memory usage of an index that encodes them redundantly in each color.

To address this issue, we describe here the meta-colored compacted de Bruijn graph, or Mac-dBG. In the Mac-dBG, a color is represented as a sequence of references to sub-sequences that are shared with potentially many other colors. We refer to these references as *meta colors*. These common sub-sequences, which we call *partial colors*, are encoded once, rather than a number of times equal to the number of colors in which they appear. This allows reducing the required

space for the index while incurring low query overhead when partial colors are sufficiently long. Indeed, we demonstrate experimentally in Section 4 that the Mac-dBG substantially improves over the space/time trade-off of a traditional c-dBG data structure.

Another key strength of this representation via meta/partial colors is its generality: it applies to any c-dBG data structure arising from the composition of $\mathcal{D}$ and $\mathcal{L}$ to readily improve its space and query time.

3.1 Definition

Let $G(\mathcal{U}, \mathcal{C})$ be the c-dBG built from the reference collection $\mathcal{R} = \{R_1, \ldots, R_N\}$. We recall from Section 2 that we indicate with $\mathcal{C} = \{C_1, \ldots, C_z\}$ the set of distinct colors of G. Let $\mathcal{N} = \{\mathcal{N}_1, \ldots, \mathcal{N}_r\}$ be a partition of $[N] = \{1, \ldots, N\}$ for some $r \geq 1$, i.e., $\mathcal{N}_i \neq \varnothing$ for all i, $\mathcal{N}_i \cap \mathcal{N}_j = \varnothing$ for all (i, j) such that $i \neq j$, and $\cup \mathcal{N}_i = [N]$. Let an order between the elements of each $\mathcal{N}_i = \{e_{i,j}\}$ be fixed (for example, by sorting the elements in increasing order). Any $\mathcal{N}$ induces a permutation $\pi : [N] \to [N]$, defined as $\pi(e_{i,j}) := j + B_{i-1}$ where $B_i = \sum_{t=1}^{i} |\mathcal{N}_t|$ for $i > 0$ and $B_0 = 0$, for $i = 1, \ldots, r$ and $j = 1, \ldots, |\mathcal{N}_i|$. We assume from now on that the N reference identifiers and the colors in $\mathcal{C}$ have been permuted according to π. After the permutation, $\mathcal{N}$ determines a partition of $\mathcal{R}$ into r disjoint sets:

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

Definition 1 (Partial colors). *Let $\mathcal{P}_i$ be the set*

$$\mathcal{P}_i = \Big\{ \{x - B_{i-1} | x \in C_t \cap \{B_{i-1} + 1, B_{i-1} + 2, \ldots, B_i - 1, B_i\}\} \,|\, \forall\, C_t \in \mathcal{C} \Big\},$$

for $i = 1, \ldots, r$. The elements $\{P_{ij}\}$ of the set $\mathcal{P}_i$ are the partial colors *induced by the partition $\mathcal{N}_i$. We indicate with $\mathcal{P} = \{\mathcal{P}_1, \ldots, \mathcal{P}_r\}$ the set of all partial color sets.*

In words, $\mathcal{P}_i$ is the set obtained by considering the distinct colors *only* for the references in the i-th partition $\mathcal{R}_i$ by noting that — by construction — they comprise integers x such that $B_{i-1} < x \leq B_i$.

The idea is that the set $\mathcal{P} = \{\mathcal{P}_1, \ldots, \mathcal{P}_r\}$ form a dictionary of sub-sequences (the partial colors) that spell the original colors $\mathcal{C} = \{C_1, \ldots, C_z\}$. Let us now formally define this spelling.

Definition 2 (Meta colors). *Let $C_t \in \mathcal{C}$ be a color. A* meta color *is an integer pair (i, j) indicating the sub-list $L := C_t[b \ldots b + |P_{ij}|]$ if there exists $0 < b \leq |C_t| - |P_{ij}|$ such that $L[l] = P_{ij}[l] + B_{i-1}$, for $l = 1, \ldots, |P_{ij}|$. It follows that C_t can be modeled as a list M_t of at most r meta colors. We indicate with $\mathcal{M} = \{M_1, \ldots, M_z\}$ the set of all meta color lists.*

Given $G(\mathcal{U}, \mathcal{C})$, the 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 as that of G but the colors $\mathcal{C}$ are represented with the partial colors $\mathcal{P}$ and the meta colors $\mathcal{M}$.

The Mac-dBG permits to encode the colors in $\mathcal{C}$ into smaller space compared to the original c-dBG and without compromising the efficiency of the $\textsc{Color}(x)$ query, for the following reasons.

1. 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. Potentially long sublists, shared between several color lists, are therefore encoded once in $\mathcal{P}$ and only referenced with $\log_2(N_p)$ bits instead of redundantly replicating their representation.
2. Each partial color P_{ij} can be encoded more succinctly because the permutation π guarantees that it only comprises integers lower-bounded by $B_{i-1}+1$ and upper-bounded by B_i. Hence only $\log_2(B_i - B_{i-1})$ bits per integer are sufficient.
3. The total number of integers in $\mathcal{P}$ is *at most* that in the original $\mathcal{C}$, i.e., $\sum_{i=1}^{r} \sum_{j=1}^{|\mathcal{P}_i|} |P_{ij}| \leq \sum_{t=1}^{z} |C_t|$ because partial colors are encoded once. In practice, $\mathcal{P}$ is expected to contain a much smaller number of integers than $\mathcal{C}$.
4. It is efficient to recover the original color C_t from the meta color list 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. This is, again, a direct consequence of having permuted the reference identifiers with π. Observe that, in principle, the representation of the colors with meta/partial colors could be described *without* any permutation π — however, one would sacrifice space (for the reason 2. above) *and* query time since decoding a color list from meta colors would eventually need to sort the decoded integers. In conclusion, permuting the reference identifiers with π is an extra degree of freedom that we can exploit to improve index space and preserve query efficiency, noting that the correctness of the index is not compromised when reference identifiers are re-assigned globally.

Example 1. Let us consider the $z = 8$ colors from Fig. 1b, for $N = 16$. Let $r = 4$ and $\mathcal{N}_1 = \{1, 12, 13, 14, 16\}, \mathcal{N}_2 = \{3, 5, 9\}, \mathcal{N}_3 = \{7, 11\}, \mathcal{N}_4 = \{2, 4, 6, 8, 10, 15\}$, assuming we use the natural order between the integers to determine an order between the elements of each $\mathcal{N}_i$. Thus, we have $B_1 = 5$, $B_2 = 8$, $B_3 = 10$, and $B_4 = 16$. The induced permutation π can be visualized by concatenating the sets $\mathcal{N}_i$ from $i = 1$ to 4 and assigning "new" identifiers, from 1 to N, in this concatenated order:

$$\begin{array}{rl} & \{\ 1\ 12\ 13\ 14\ 16\ \}\{\ 3\ 5\ 9\ \}\{\ 7\ 11\ \}\{\ 2\ \ 4\ \ 6\ \ 8\ \ 10\ 15\ \} \\ \textit{new identifiers} \rightarrow & 1\ \ 2\ \ 3\ \ 4\ \ 5\ \ \ \ 6\ 7\ 8\ \ \ \ 9\ 10\ \ \ \ 11\ 12\ 13\ 14\ 15\ 16 \end{array}$$

which results in $\pi(1) = 1$, $\pi(12) = 2$, $\pi(13) = 3$, etc., that is

$$\pi = [1, 11, 6, 12, 7, 13, 9, 14, 8, 15, 10, 2, 3, 4, 16, 5].$$

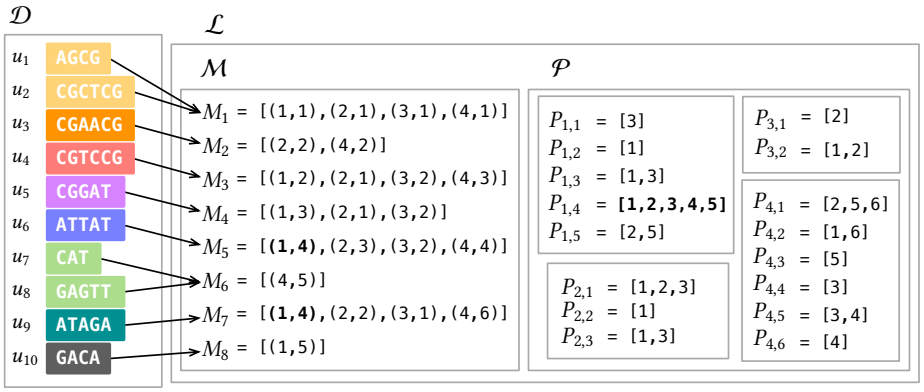


Fig. 2: Mac-dBG layout discussed in Example 1 for the colors of the c-dBG from Fig. 1. Note that the partial color $P_{1,4} = [1, 2, 3, 4, 5]$ shared between C_5 and C_7 is now represented *once* as a direct consequence of partitioning, and indicated with the pair $(1, 4)$ instead of replicating the five integers it contains in both C_5 and C_7. The same consideration applies to other shared sub-sequences.

Now we apply the permutation π to each color, obtaining the following permuted colors (vertical bars represent the partial color boundaries $B_1, \ldots, B_4$).

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

For example, color C_1, that before was $[3, 4, 5, 9, 10, 11, 13, 15]$ (see Fig. 1b), 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 colors are the distinct sub-sequences in each partition of the permuted colors. For example, $\mathcal{P}_1$ is the set of the distinct sub-sequences in partition 1, i.e., those comprising the integers x such that $0 < x \leq B_1 = 5$. Hence, we have five distinct partial colors in partition 1, and these are $[3]$, $[1]$, $[1, 3]$, $[1, 2, 3, 4, 5]$, and $[2, 5]$. Importantly, note that from the integers in partial colors from partition $i > 1$ we can subtract the lower bound B_{i-1}. For example, from the integers in the partial color $[6, 7, 8]$ from C_1 in partition 2 we can subtract $B_1 = 5$, hence obtaining $[1, 2, 3]$. Overall, we thus obtain that $\mathcal{P}$ comprises four partial color sets, as shown in Fig. 2. The figure also shows the rendering of the colors $\mathcal{C} = \{C_1, \ldots, C_8\}$ via meta color lists, i.e., how each color can be spelled by a proper concatenation of partial colors.

3.2 Data structures used and two-level intersection algorithm

Given a Mac-dBG $G(\mathcal{U}, \mathcal{N}, \pi, \mathcal{P}, \mathcal{M})$, a concrete implementation includes a representation for $\mathcal{U}$, $\mathcal{P}$, and $\mathcal{M}$ (plus also the sorted array $B[1..r] = [0, B_1, \ldots, B_{r-1}]$).

The Mac-dBG is not bound to any specific compression scheme nor any specific dictionary data structure, allowing one to obtain a spectrum of different space/time trade-offs depending on choices made. In this paper, we made the following choices: (1) we use the SSHash data structure [28,29] to represent the set of unitigs $\mathcal{U}$; (2) we adopt the same compression methods as used in Fulgor [13] to compress the partial colors and the same mechanism to map unitigs to their colors (using a binary vector of length m, equipped with ranking capabilities); (3) we represent each meta color list as a list of $\log_2(N_p)$-bit integers.

Very importantly, note that choices (1) and (2) directly imply that our Mac-dBG implementation fully exploits the key unitig properties described in Section 2 as Fulgor does.

The Mac-dBG opens the possibility to achieve even faster query times than a traditional c-dBG, due to the manner in which the partitions factorize the space of references, if a two-level intersection algorithm is employed for pseudoalignment. There are several pseudoalignment algorithms (see [13, Section 4] for an overview) that standard c-dBG data structures directly support; here we focus on the *full intersection* algorithm. Given a query string Q, we consider it as a set of k-mers. Let $\mathcal{K}(Q) = \{x \in Q | \textsc{Color}(x) \neq \varnothing\}$. The full intersection method computes the intersection between the colors of all the k-mers in $\mathcal{K}(Q)$. Our two-level intersection algorithm is as follows. First, only meta colors are intersected (thus, without any need to access the partial colors) to determine the partitions in common to all colors being intersected. Then only the common partitions are considered. Two cases can happen for each partition. (1) The meta color is the same for all colors: in this case, the result of the intersection is implicit and it suffices to decode the partial color indicated by the meta color. (2) The meta color is not the same, hence we have to compute the intersection between different partial colors. This optimization is beneficial when the colors being intersected have very few partitions in common, or when they have identical meta colors.

3.3 The optimization problem

As evident from its definition, the effectiveness of a Mac-dBG crucially depends on the choice of the partition $\mathcal{N}$ and upon the order of the references within each partition as given by the permutation π. There is, in fact, an evident friction between the encoding costs of the partial and meta colors. Let N_m and $N_p = \sum_{i=1}^{r} |\mathcal{P}_i|$ be the number of meta and partial colors, respectively. Since each meta color can be indicated with $\log_2(N_p)$ bits, meta colors cost $N_m \log_2(N_p)$ bits overall. Instead, let $\textsc{Cost}(P_{ij}, \pi)$ be the encoding cost (in bits) of the partial color P_{ij} according to some function $\textsc{Cost}$. On one hand, we would like to select a large value of r so that N_p diminishes since each color is partitioned into several, small, partial colors, thereby increasing the chances that each partition has many repeated sub-sequences. This will help in reducing the encoding cost for the partial colors, i.e., the quantity $\sum_{i=1}^{r} \sum_{j=1}^{|\mathcal{P}_i|} \textsc{Cost}(P_{ij}, \pi)$. On the other hand, a large value of r will yield longer meta color lists, i.e., increase N_m. This, in turn, could erode the benefit of encoding shared patterns and would require more time to decode each meta color list.

We can therefore formalize the following optimization problem that we call *minimum-cost partition arrangement* (MPA).

Problem 2 (Minimum-cost partition arrangement). Let $G(\mathcal{U}, \mathcal{C})$ be the compacted c-dBG built from the reference collection $\mathcal{R} = \{R_1, \ldots, R_N\}$. Determine the partition $\mathcal{N} = \{\mathcal{N}_1, \ldots, \mathcal{N}_r\}$ of $[N] = \{1, \ldots, N\}$ for some $r \geq 1$ 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 upon 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 colors is an instance of the bipartite minimum logarithmic arrangement problem (BIMLOGA) as introduced by Dhulipala et al. [11] for the purpose of minimizing the cost of delta-encoded lists in inverted indexes. The BIMLOGA problem is NP-hard [11]. We note that BIMLOGA is a special case of MPA: that for $r = 1$ (one partition only) and $\textsc{Cost}(P_{ij}, \pi)$ being the $\log_2$ of the gaps between consecutive integers. It follows that also MPA is NP-hard under these constraints. This result immediately suggests that it is unlikely that polynomial-time algorithms exist for solving the MPA problem.

3.4 The SCPO framework

In this section we propose a construction algorithm for the Mac-dBG. The algorithm is an heuristic for the MPA optimization problem defined in the previous section (Problem 2), and it is based on the intuition that *similar* references should be grouped together in the same partition so as to *increase the likeliness of having a smaller number of longer shared sub-sequences.* The algorithm therefore consists in the following four steps: (1) *Sketching*, (2) *Clustering*, (3) *Partitioning*, and (4) *Ordering* (SCPO).

1. Sketching. We argue that a reasonable way of assessing the similarity between two references is determining the number of unitigs that they have in common. Recall from Property 1 (Section 2) that each reference $R_i \in \mathcal{R}$ can be spelled by a proper concatenation (a "tiling") of the unitigs of the underlying compacted dBG. If these unitigs are assigned unique identifiers by SSHash, it follows that each R_i can be seen as a list of unitig identifiers. The idea is that these integer lists are much shorter and take less space than the actual DNA references. To reduce the space of a list even further, we compute a *sketch* of the list based on the fact that if two sketches are similar, then the original lists are similar as well.

2. Clustering. The sketches are fed as input of a clustering algorithm.

3. Partitioning. Once the clustering is done, each input reference R_i is labeled with the cluster label of the corresponding sketch so that the partition of $\mathcal{R}$ into $\mathcal{R}_1, \ldots, \mathcal{R}_r$ is uniquely determined.

4. Ordering. Finally, one may *order* the references in each $\mathcal{R}_i$ to determine a permutation π that yields a better compression for the partial colors $\mathcal{P}_i$. In fact, while the goal of clustering and partitioning is to factor out repeated sub-patterns within the colors, the goal of the ordering step is to assign nearby identifiers to references that tend to co-occur within the partial colors (as already mentioned in Section 3.3).

Specific framework instance. In this work, we use the following specific instance of this framework. We build *hyper-log-log* [14] sketches of $W = 2^{10}$ bytes each. As clustering algorithm, we use a *divisive* K-means approach that does not need an a-priori number of clusters to be supplied as input. At the beginning of the algorithm, the whole input forms a single cluster that is recursively split into two clusters until the mean squared error (MSE) between the sketches in the cluster and the cluster's centroid is not below a prescribed threshold (which we fix to 10% of the MSE at the start of the algorithm). Let r be the number of found clusters. The complexity of the algorithm depends on the topology of the binary tree representing the hierarchy of splits performed. In the worst case, the topology is completely unbalanced and the complexity is $O(WNr)$; in the best case, the topology is perfectly balanced instead, for a cost of $O(WN \log r)$. Note that the worst-case bound is very pessimistic because, in practice, the formed clusters tend to be reasonably well-balanced in size.

In the current version of the work, we did not perform any ordering of the references within each cluster. We leave the investigation of this opportunity as future work.

4 Experiments

This section presents the results of experiments conducted to assess the performance of the Mac-dBG. We fixed the k-mer length to $k = 31$. All experiments were run on a machine equipped with Intel Xeon Platinum 8276L CPUs (clocked at 2.20GHz), 500 GB of RAM, and running Linux 4.15.0.

Datasets. We build Mac-dBGs with the proposed SCPO framework on the following pangenomes: 3,682 *E. Coli* (EC) genomes from NCBI [1]; different collections of *S. Enterica* (SE) genomes (from 5,000 up to 150,000 genomes) from the collection by Blackwell et al. [7]. Additionally, we also include a much more diverse collection of 30,691 genomes assembled from human gut samples (GB), originally published by Hiseni et al. [15].

Other evaluated tools. We compare the Mac-dBG against the following indexes: Fulgor [13], Themisto [2], MetaGraph [17,18,19], and COBS [6]. Links to the corresponding software libraries can be found in the References. We use the `C++` implementations from the respective authors. All software was compiled with gcc 11.1.0.

We provide some details on the tested tools. Both Themisto and COBS were built under default parameters as suggested by the authors, that is: option `-d 20` for Themisto which enables the sampling of k-mer colors in the

Table 1: Index space in GB, broken down by space required for indexing the k-mers in the dBG (SSHash for both Fulgor and Mac-dBG, SBWT for Themisto, and BOSS for MetaGraph) and data structures required to encode colors and map k-mers to colors.

	Genomes	Mac-dBG			Fulgor			Themisto			MetaGraph			COBS
		dBG	Colors	Total	dBG	Colors	Total	dBG	Colors	Total	dBG	Colors	Total	Total
EC	3,682	0.29	0.52	0.81	0.29	1.36	1.65	0.22	1.85	2.08	0.10	0.23	0.33	7.53
SE	5,000	0.16	0.16	0.32	0.16	0.59	0.75	0.14	1.29	1.43	0.07	0.19	0.26	9.11
	10,000	0.35	0.33	0.68	0.35	1.66	2.01	0.32	3.50	3.81	0.13	0.38	0.51	18.68
	50,000	1.26	2.14	3.40	1.26	17.03	18.30	1.07	32.42	33.48	0.36	1.95	2.31	88.61
	100,000	1.72	3.83	5.55	1.72	40.70	42.44	1.35	75.94	77.28	0.45	3.50	3.95	173.58
	150,000	2.03	5.37	7.40	2.03	68.60	70.66	1.58	125.16	126.74	—	—	—	265.49
GB	30,691	21.31	7.85	29.16	21.31	15.45	36.85	18.33	30.88	49.21	5.23	4.77	10.00	21.23

SBWT for better space effectiveness; in COBS, we have shards of at most 1024 references where each Bloom filter has a false positive rate of 0.3 and one hash function. MetaGraph indexes were built with the *relaxed row-diff* BRWT data structure [18] using a workflow available at https://github.com/theJasonFan/metagraph-workflows that we wrote with the input of the MetaGraph authors.

Index size. Table 1 reports the total *on disk* index size for all of the methods evaluated. Compared to the most recent indexes, Fulgor and Themisto, that where previously shown to achieve the most desirable space/time trade-offs, Mac-dBG substantially improves on the space (and, as we shall see next, without any negative impact on query time). In fact, the only index smaller on disk than Mac-dBG is MetaGraph in the relaxed row-diff BRWT configuration — at least in the cases where we were able to construct the latter within the construction resource constraints. However, unlike the other indexes evaluated, the *on disk* index size MetaGraph is not representative of the working memory required for query when using the (recommended and default) batch mode query.

The COBS index, despite being approximate, is consistently and considerably larger than all of the other (exact) indexes, except for the the Gut bacteria collection (GB). The differing behavior on GB likely derives from the fact that the diversity of that data cause the exact indexes to spend a considerable fraction of their total size on the representation of the k-mer dictionary itself (e.g., $18-21.3$ GB). However COBS, by design, eliminates this component of the index entirely.

Finally we observe that, as the number of references grow in the SE datasets, the already-large savings of Mac-dBG become even more prominent. For example Mac-dBG is 43% of the size of Fulgor (2.34× smaller) for SE 5,000, but is only 10% of the size of Fulgor (9.55× smaller) for SE 150,000. As the size of the collection grows, and more repetitive sub-patterns in the collection of colors appears, the Mac-dBG index is able to better capture and eliminate this redundancy.

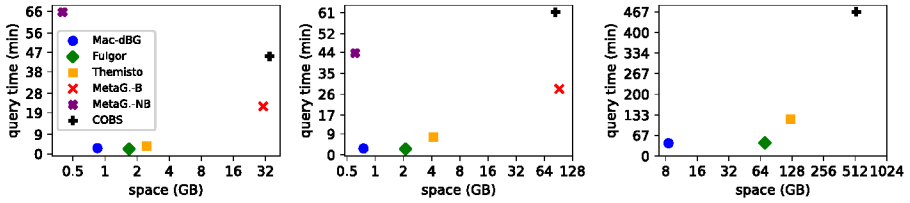
Table 2: Total query time (elapsed time) and memory used during query (max. RSS) as reported by `/usr/bin/time -v`, using 16 processing threads. The read-mapping output is written to `/dev/null` for this experiment. We also report the mapping rate in percentage (fraction of mapped read over the total number of queried reads). The query algorithm used here is full-intersection. The "B" query mode of MetaGraph corresponds to the batch mode (with default batch size); the "NB" corresponds to the non-batch query mode instead. In red font we highlight the workloads exceeding the available memory (> 500 GB).

	Genomes	Rate	Mac-dBG		Fulgor		Themisto		MetaG.-B		MetaG.-NB		COBS	
			mm:ss	GB	mm:ss	GB	h:mm:ss	GB	mm:ss	GB	h:mm:ss	GB	h:mm:ss	GB
EC	3,682	98.99	2:40	0.85	2:10	1.68	0:03:40	2.46	22:00	30.44	1:05:41	0.40	0:45:11	34.93
SE	5,000	89.49	1:16	0.37	1:16	0.82	0:03:50	1.82	14:14	36.54	0:20:32	0.33	0:38:34	41.93
	10,000	89.71	2:45	0.75	2:26	2.11	0:07:35	4.16	28:15	92.18	0:43:40	0.61	1:01:14	84.20
	50,000	91.25	14:00	3.65	19:15	18.53	0:42:02	33.14	—	—	4:30:03	2.72	3:54:18	408.82
	100,000	91.41	26:48	6.29	27:30	42.78	1:22:00	75.93	—	—	9:40:06	4.82	8:07:29	522.56
	150,000	91.52	41:30	8.51	42:30	70.55	2:00:13	124.27	—	—	—	—	7:47:14	522.63
GB	30,691	92.91	01:03	28.51	01:10	30.02	0:01:20	48.47	28:55	15.86	0:22:05	9.91	0:34:45	225.57

Query efficiency. Table 2 reports the query times of the indexes, performing full-intersection pseudoalignment (see Alg. 1 from [13]), on a high-hit workload. The queried reads consist of all FASTQ records in the first read file of the following accessions: SRR1928200 for EC, SRR801268 for SE, and ERR321482 for GB. These files contain several million reads each. Timings are relative to a second run of each experiment, where the indexes are loaded from the disk cache (which benefits the larger indexes more than the smaller ones).

Consistent with previously reported results [13], we find that among existing indexes, Fulgor provides the fastest queries. As expected, Mac-dBG does not not sacrifice query efficiency compared to Fulgor. After Mac-dBG and Fulgor, we note that Themisto is the next fastest index, followed by MetaGraph in batch query mode. The query speeds of COBS and of MetaGraph when not executed in batch mode are much lower than that of the other indexes, in some cases being (more than) an order of magnitude slower.

Critically, it is not the case with all indexes evaluated here that the size of the index *on disk* is a good proxy for the memory required to actually query the index. Specifically, for MetaGraph, when used in batch query mode ("B"), the required memory can exceed the on-disk index size by up to 2 orders of magnitude, and in several tests this resulted in the exhaustion of available memory and an inability to complete the queries under the tested configuration. On the other hand, Fulgor, Themisto, Mac-dBG 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 disk.



(a) EC (b) SE 10,000 (c) SE 150,000

Fig. 3: The same data from Table 2 but shown as space vs. time trade-off curves, for some example datasets.

5 Conclusions

We have introduced the Mac-dBG data structure. The Mac-dBG represents a new state-of-the-art representation for answering color queries over large collections of reference sequences, and achieves a considerable improvement over existing work in terms of the space/time trade-off it offers. Specifically, Mac-dBG is almost as small as the smallest variant of MetaGraph — which is the smallest compressed c-dBG representation on disk. Yet, when queried, Mac-dBG requires essentially the same space as is required for the index on disk, while the MetaGraph representation expands manyfold to improve query throughput via batch queries. At the same time, Mac-dBG provides query speed as fast as the fastest existing c-dBG index, Fulgor. This enhanced trade-off can be visualized in Fig. 3. We believe these characteristics make Mac-dBG a very promising data structure for enabling large-scale color queries across a range of applications.

To achieve these substantial improvements over the prior state of the art, the Mac-dBG focuses on providing an improved representation of the color table, the element of the index that tends to grow most quickly as the number of indexed references increases. Specifically, Mac-dBG compresses the colors by factoring out shared sub-patterns that occur across different colors. The color table is represented as a set of *meta* colors and *partial* colors which are combined to recover the original colors exactly. While most interesting formulations of determining the optimal factorization into meta colors and partial colors appear NP-hard, we nonetheless describe a heuristic approach that works well in practice.

Acknowledgements. We are grateful to Laxman Dhulipala for useful comments on an early draft of this paper.

Fundings. This work is supported by the NIH under grant award numbers R01HG009937 to R.P.; the NSF awards CCF-1750472 and CNS-1763680 to R.P, and DGE-1840340 to J.F. Funding for this research has also been provided by the European Union's Horizon Europe research and innovation programme (EFRA project, Grant Agreement Number 101093026). This work was also partially supported by DAIS – Ca' Foscari University of Venice within the IRIDE program.

Declarations. R.P. is a co-founder of Ocean Genomics inc.

References

1. Alanko, J.N.: 3682 E. Coli assemblies from NCBI (2022), https://zenodo.org/records/6577997
2. Alanko, J.N., Vuohtoniemi, J., Mäklin, T., Puglisi, S.J.: Themisto: a scalable colored k-mer index for sensitive pseudoalignment against hundreds of thousands of bacterial genomes. Bioinformatics **39**(Supplement_1), i260–i269 (Jun 2023), https://github.com/algbio/themisto
3. Almodaresi, F., Sarkar, H., Srivastava, A., Patro, R.: A space and time-efficient index for the compacted colored de Bruijn graph. Bioinformatics **34**(13), i169–i177 (2018)
4. Almodaresi, F., Zakeri, M., Patro, R.: PuffAligner: a fast, efficient and accurate aligner based on the pufferfish index. Bioinformatics **37**(22), 4048–4055 (Jun 2021)
5. Baier, U., Beller, T., Ohlebusch, E.: Graphical pan-genome analysis with compressed suffix trees and the burrows–wheeler transform. Bioinformatics **32**(4), 497–504 (2016)
6. Bingmann, T., Bradley, P., Gauger, F., Iqbal, Z.: Cobs: a compact bit-sliced signature index. In: International Symposium on String Processing and Information Retrieval. pp. 285–303. Springer (2019), https://github.com/bingmann/cobs
7. Blackwell, G.A., Hunt, M., Malone, K.M., Lima, L., Horesh, G., Alako, B.T.F., Thomson, N.R., Iqbal, Z.: Exploring bacterial diversity via a curated and searchable snapshot of archived DNA sequences. PLOS Biology **19**(11), 1–16 (11 2021), http://ftp.ebi.ac.uk/pub/databases/ENA2018-bacteria-661k
8. Bray, N.L., Pimentel, H., Melsted, P., Pachter, L.: Near-optimal probabilistic rna-seq quantification. Nature biotechnology **34**(5), 525–527 (2016)
9. Cleary, A., Ramaraj, T., Kahanda, I., Mudge, J., Mumey, B.: Exploring Frequented Regions in Pan-Genomic Graphs. IEEE/ACM Transactions on Computational Biology and Bioinformatics **16**(5), 1424–1435 (Sep 2019)
10. Dede, K., Ohlebusch, E.: Dynamic construction of pan-genome subgraphs. Open Computer Science **10**(1), 82–96 (Apr 2020)
11. Dhulipala, L., Kabiljo, I., Karrer, B., Ottaviano, G., Pupyrev, S., Shalita, A.: Compressing graphs and indexes with recursive graph bisection. In: Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining. pp. 1535–1544 (2016)
12. Fan, J., Khan, J., Pibiri, G.E., Patro, R.: Spectrum preserving tilings enable sparse and modular reference indexing. In: Research in Computational Molecular Biology. pp. 21–40 (2023)
13. Fan, J., Singh, N.P., Khan, J., Pibiri, G.E., Patro, R.: Fulgor: A Fast and Compact k-mer Index for Large-Scale Matching and Color Queries. In: 23rd International Workshop on Algorithms in Bioinformatics (WABI 2023). pp. 18:1–18:21 (2023), https://github.com/jermp/fulgor
14. Flajolet, P., Fusy, É., Gandouet, O., Meunier, F.: Hyperloglog: the analysis of a near-optimal cardinality estimation algorithm. In: Discrete Mathematics and Theoretical Computer Science. pp. 137–156. Discrete Mathematics and Theoretical Computer Science (2007)
15. Hiseni, P., Rudi, K., Wilson, R.C., Hegge, F.T., Snipen, L.: HumGut: a comprehensive human gut prokaryotic genomes collection filtered by metagenome data. Microbiome **9**(1), 1–12 (2021), https://arken.nmbu.no/~larssn/humgut/index.htm
16. Holley, G., Melsted, P.: Bifrost: highly parallel construction and indexing of colored and compacted de Bruijn graphs. Genome biology **21**(1), 1–20 (2020)

17. Karasikov, M., Mustafa, H., Danciu, D., Barber, C., Zimmermann, M., Rätsch, G., Kahles, A.: Metagraph: Indexing and analysing nucleotide archives at petabase-scale. BioRxiv pp. 2020–10 (2020)
18. Karasikov, M., Mustafa, H., Joudaki, A., Javadzadeh-no, S., Rätsch, G., Kahles, A.: Sparse Binary Relation Representations for Genome Graph Annotation. Journal of Computational Biology **27**(4), 626–639 (Apr 2020), https://github.com/ratschlab/metagraph
19. Karasikov, M., Mustafa, H., Rätsch, G., Kahles, A.: Lossless indexing with counting de bruijn graphs. Genome Research **32**(9), 1754–1764 (2022)
20. Lees, J.A., Mai, T.T., Galardini, M., Wheeler, N.E., Horsfield, S.T., Parkhill, J., Corander, J.: Improved Prediction of Bacterial Genotype-Phenotype Associations Using Interpretable Pangenome-Spanning Regressions. mBio **11**(4) (Aug 2020)
21. Liu, B., Guo, H., Brudno, M., Wang, Y.: deBGA: read alignment with de bruijn graph-based seed and extension. Bioinformatics **32**(21), 3224–3232 (Jul 2016)
22. Luhmann, N., Holley, G., Achtman, M.: BlastFrost: fast querying of 100, 000s of bacterial genomes in bifrost graphs. Genome Biology **22**(1) (Jan 2021)
23. Mäklin, T., Kallonen, T., David, S., Boinett, C.J., Pascoe, B., Méric, G., Aanensen, D.M., Feil, E.J., Baker, S., Parkhill, J., et al.: High-resolution sweep metagenomics using fast probabilistic inference [version 1; peer review: 1 approved, 1 approved with reservations]. Wellcome open research **5**(14) (2021)
24. Manuweera, B., Mudge, J., Kahanda, I., Mumey, B., Ramaraj, T., Cleary, A.: Pangenome-Wide Association Studies with Frequented Regions. In: Proceedings of the 10th ACM International Conference on Bioinformatics, Computational Biology and Health Informatics. ACM (Sep 2019)
25. Marcus, S., Lee, H., Schatz, M.C.: Splitmem: a graphical algorithm for pan-genome analysis with suffix skips. Bioinformatics **30**(24), 3476–3483 (2014)
26. Minkin, I., Medvedev, P.: Scalable multiple whole-genome alignment and locally collinear block construction with SibeliaZ. Nature Communications **11**(1) (Dec 2020)
27. Minkin, I., Medvedev, P.: Scalable pairwise whole-genome homology mapping of long genomes with BubbZ. iScience **23**(6), 101224 (Jun 2020)
28. Pibiri, G.E.: Sparse and skew hashing of k-mers. Bioinformatics **38**(Supplement_1), i185–i194 (06 2022)
29. Pibiri, G.E.: On weighted k-mer dictionaries. Algorithms for Molecular Biology **18**(3) (2023)
30. Pibiri, G.E., Venturini, R.: Techniques for inverted index compression. ACM Computing Surveys (CSUR) **53**(6), 125:1–125:36 (2021)
31. Rahman, A., Dufresne, Y., Medvedev, P.: Compression Algorithm for Colored de Bruijn Graphs. In: 23rd International Workshop on Algorithms in Bioinformatics (WABI 2023). pp. 17:1–17:14 (2023)
32. Reppell, M., Novembre, J.: Using pseudoalignment and base quality to accurately quantify microbial community composition. PLOS Computational Biology **14**(4), 1–23 (04 2018)
33. Schaeffer, L., Pimentel, H., Bray, N., Melsted, P., Pachter, L.: Pseudoalignment for metagenomic read assignment. Bioinformatics **33**(14), 2082–2088 (02 2017)
34. Skoufos, G., Almodaresi, F., Zakeri, M., Paulson, J.N., Patro, R., Hatzigeorgiou, A.G., Vlachos, I.S.: AGAMEMNON: an accurate metaGenomics and MEtatranscriptoMics quaNtificatiON analysis suite. Genome Biology **23**(1) (Jan 2022)