Meta-colored compacted de Bruijn graphs: overview and challenges

Giulio Ermanno Pibiri[1,2], Jason Fan[3], and Rob Patro[3]

[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. This note introduces the *meta-colored* compacted de Bruijn graph (Mac-dBG) – a colored de Bruijn graph data structure where colors are represented holistically, i.e., taking into account their redundancy across the whole set of colors, rather than individually as atomic integer lists. Preliminary results show that Mac-dBGs achieve remarkably better compression effectiveness of the color information compared to the state of the art, without sacrificing query time. We discuss a simple framework to build Mac-dBGs and conclude with a list of potentially fruitful research directions that we are currently pursuing.

1 Introduction

The colored de Bruijn graph (c-dBG) and its compacted variant have become core tools used across several areas of genomics and pangenomics. For example, they have been widely adopted by methods that perform read mapping or alignment and subsequent downstream analyses, specifically with respect to RNA-seq and metagenomic identification and abundance estimation [1,2,3,4,5,6,7,8]; among methods that perform homology assessment and mapping of genomes [9,10]; for a variety of different tasks in pangenome analysis [11,12,13,14,15,16,17]; for storage and compression of genomic data [18]. In most of these applications, a key requirement of the underlying representation used 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 $\Sigma = \{A, C, G, T\}$. We want to build a data structure (referred to as the *index*) that allows us to retrieve the set $\textsc{Color}(x) = \{i | x \in R_i\}$ as efficiently as possible for any k-mer $x \in \Sigma^k$. Note that $\textsc{Color}(x) = \varnothing$ if x does not occur in any reference. 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 species or population. Pangenomes have revolutionized DNA analysis by providing a more comprehensive understanding of genetic diversity within a species [19,20]. Unlike traditional reference genomes that represent a single individual or a small set of individuals, pangenomes incorporate genetic information from multiple individuals within a species or population. This approach is particularly valuable because it captures a wide range of genetic variations present in a species, including rare and unique genetic sequences that may be absent from the reference genome.

The goal of this note is to sketch a solution to Problem 1 for 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.) The result is the *meta-colored* compacted de Bruijn graph (Mac-dBG) – a new data structure for the c-dBG where colors are represented *holistically*, i.e., taking into account their redundancy across the whole set of colors, rather than individually

as atomic integer lists. We define Mac-dBGs in Section 4.1, present the underlying optimization problem in Section 4.2, and discuss a simple framework for their construction in Section 4.3. We present some preliminary experimental results in Section 5 to demonstrate that Mac-dBGs achieve remarkably better compression effectiveness compared to the state of the art (covered in Section 3), *without* sacrificing query time. We conclude in Section 6 with a list of potentially fruitful research directions that we are currently pursuing.

A prototype implementation of the Mac-dBG, implemented in `C++17`, can be found at the following GitHub repository: https://github.com/jermp/fulgor/tree/dev.

2 Preliminaries

2.1 The modular indexing framework

In principle, Problem 1 could be solved using a classic but elegant data structure: the *inverted index* [21,22]. Let $\mathcal{L}$ be an inverted index for $\mathcal{R}$. $\mathcal{L}$ explicitly stores the ordered set $\textsc{Color}(x)$ for each k-mer $x \in \mathcal{R}$. We wish to implement the map $x \to \textsc{Color}(x)$ 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 $x \to \textsc{Color}(x)$, $\mathcal{D}$ 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.

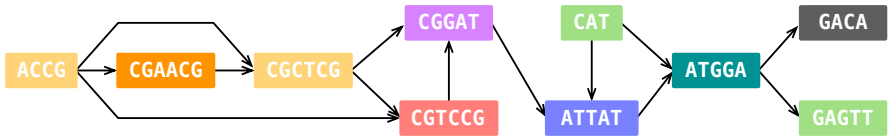
Problem 1 can then be solved using these two data structures – $\mathcal{D}$ and $\mathcal{L}$ – thanks to the interplay between $\textsc{Lookup}(x)$ and $\textsc{Color}(x)$. Logically, the index stores the sets $\{\textsc{Color}(x)\}_{x \in \mathcal{R}}$ in compressed form, *sorted by* the value of $\textsc{Lookup}(x)$.

2.2 The colored compacted de Bruijn graph and its properties

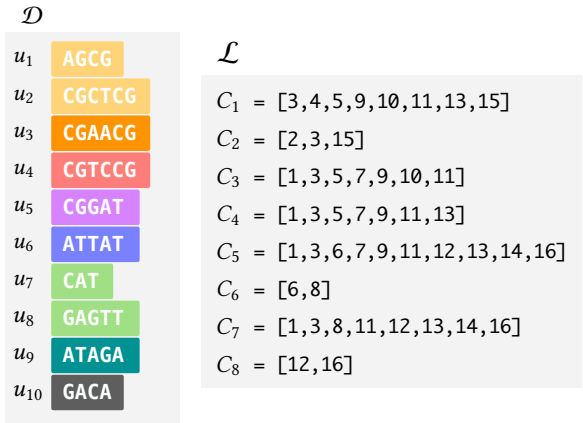
Problem 1 has some specific properties that we can exploit to implement as efficiently as possible the modular indexing framework described in Section 2.1. First, consecutive k-mers share $(k-1)$-length overlaps; second, k-mers that co-occur in the same set of references have the same color. A useful, standard, formalism that describes these properties is the so-called *colored (compacted) de Bruijn graph* (c-dBG).

Given the collection of references $\mathcal{R}$, the (node-centric) de Bruijn graph (dBG) of $\mathcal{R}$ is a directed graph whose nodes are all the distinct k-mers of $\mathcal{R}$ and there is an edge connecting node u to node v if the $(k-1)$-length suffix of u is equal to the $(k-1)$-length prefix of v. We refer to k-mers and nodes in a dBG interchangeably. Likewise, a path in a dBG spells the string obtained by "gluing" together all the k-mers along the path. In particular, unary (i.e., non-branching) paths can be collapsed into single nodes spelling strings that are referred to as *unitigs*. The dBG arising from this compaction step is called the compacted dBG. Often one wishes to construct a bi-directed de Bruijn graph and associate both a k-mer and its reverse-complement with the same node in the graph. In this case, special care must be taken to ensure the compacted de Bruijn graph is correctly constructed under these restrictions [24,25,26]. Here, we are concerned only with the indexing problem, and will assume that the appropriate compacted graph has already been constructed.

Lastly, the *colored* compacted dBG (c-dBG) is obtained by logically annotating each k-mer x with its color, $\textsc{Color}(x)$, and only collapsing non-branching paths with nodes having the same color. We note that different conventions have been adopted in the literature, with some variants of the c-dBG enforcing *monochromatic* unitigs as just described, while others allow the colors of k-mers to change along a unitig, as long as the underling compacted dBG does not branch [27]. Here, our discussion shall be in terms of c-dBGs with monochromatic unitigs. Figure 1a illustrates an example of a c-dBG with these properties.



(a) c-dBG



(b) state-of-the-art index layout

Fig. 1. In panel (a), an example compacted de Bruijn graph (dBG) for $k = 3$. (We assume that we treat a k-mer and its reverse complement as two *different* k-mers for ease of illustration. Our software implementations consider them as 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 (i.e., the Fulgor index [23]) assuming the dBG was built for $N = 16$ references, highlighting the modular composition of a k-mer dictionary, $\mathcal{D}$, and an inverted index, $\mathcal{L}$ (as explained in Section 2.1).

We indicate with $\{u_1, \ldots, u_m\}$ the set of unitigs of the c-dBG induced by the k-mers of $\mathcal{R}$. We write $x \in u_i$ to indicate that k-mer x is a substring of the unitig u_i. The unitigs have the following key properties.

Property 1. *Unitigs are contiguous sub-sequences that spell references in* $\mathcal{R}$. Each distinct k-mer of $\mathcal{R}$ appears once, as sub-string of some unitig of the cdBG. 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 overlap and oritentation), spell R_i [28]. Joining together k-mers into unitigs reduces their storage requirements and accelerates looking up k-mers in consecutive order.

Property 2. *Unitigs are monochromatic.* The k-mers belonging to the same unitig u_i all have the same color[4]. Thus, we shall use $\textsc{Color}(u_i)$ to denote the color of each k-mer $x \in u_i$.

Property 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 M the number of distinct color sets $\mathcal{C} = \{C_1, \ldots, C_M\}$. Note that $M \leq m$ and that, in practice, there are almost always *dramatically more* unitigs than there are distinct colors.

[4] We note that this property holds only if one considers k-mers appearing at the start or end of reference sequences to be *sentinel* k-mers that must terminate their containing unitig [29,24], and that such conventions are not always adopted [27,30].

3 State of the art

The solutions proposed in the literature to represent c-dBGs, and that fall under the "color-aggregative" classification [31], all provide different implementations of the modular indexing framework as described in Section 2.1. As such, they require an efficient k-mer dictionary along with a compressed inverted index. For example, Themisto [32] makes use of the *spectral* BWT (or SBWT) data structure [33] for its k-mer dictionary and an inverted index compressed with different strategies based on the sparseness of the color lists (ratio $|C_i|/N$). Metagraph [34] uses the BOSS [35] data structure for the dictionary and exposes several general schemes to compress metadata associated with k-mers [34,36], which essentially constitute an inverted index. Bifrost [27], instead, uses a (dynamic) hash table to store the set of unitigs and an inverted index compressed with Roaring bitmaps [37]. Themisto was shown to outperform both Bifrost and Metagraph under the most common performance metrics [32] and configurations, namely the efficiency of the $\textsc{Color}(x)$ query, the space of the index, and the construction time of the index. In particular, Themisto uses practically the same space as Bifrost, but is faster to build and query. Compared to the fastest variant of Metagraph, Themisto offers similar query performance, but is much more space-efficient; on the other hand, Themisto is much faster to query than the most-space efficient variant of Metagraph. It should be noted, however, that Metagraph exposes many different potential configurations and options regarding how the index is composed, and these have not been comprehensively evaluated in the experiments of prior work [32].

However, none of the solutions mentioned so far exploit all the three properties of the unitigs described above to achieve faster query time and better space effectiveness. For example, Themisto disregards Property 1 as a direct consequence of using the SBWT data structure that internally arranges the k-mers in *colexicographic* order, and not unitig order. It exploits Property 3 instead, by compressing only the set of the *distinct* colors, but still needs to annotate each k-mer with $O(\log(M))$ bits to explicitly indicate its color, thus failing to exploit Property 2 (again, as a consequence of using the SBWT). Alanko et al. describe how it is possible to reduce this storage to $O(\log(M))$ for only some k-mers (the so-called "key" k-mers), while instead using $1 + o(1)$ storage for the other k-mers. However, this still requires dedicated storage related to the color *per*-k-mer. Further, in practice, one must store at least one "key" k-mer *per*-unitig, and this sampling approach can negatively impact the query speed.

To the best of our knowledge, the only solution that exploits *all* the three properties is the recently-introduced Fulgor [23] index, which has been shown to largely improve over Themisto with regard to index size, query performance, and construction resources. The strategy implemented by Fulgor is to first map k-mers to unitigs using the dictionary $\mathcal{D}$, and then map unitigs to their colors $\mathcal{C} = \{C_1, \ldots, C_M\}$. The colors themselves in $\mathcal{C}$ are stored in compressed form in a inverted index $\mathcal{L}$. By *composing* these mappings, Fulgor obtains an efficient map directly from k-mers to their associated colors (see also Figure 1b). The result is achieved by leveraging the *order-preserving* property of its dictionary data structure – SSHash [38,39] – which explicitly stores the set of unitigs in *any* desired order. This property has some remarkable 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 $\textsc{Lookup}(x)$ operation of SSHash can efficiently determine the unitig identifier i, allowing to map k-mers to unitigs. Third, if unitigs are sorted, so that unitigs having the same color are consecutive, then mapping unitigs to the set of distinct colors can be implemented in $1 + o(1)$ bits *per unitig*, a dramatic space improvement over Themisto that has to store $O(\log(M))$ bits *per k-mer* to implement the mapping – or, as mentioned above, $O(\log(M))$ bits per "key" k-mer plus $1+o(1)$ bits per non-"key" k-mer at the potential cost of slower queries. Practically, this reduces the space requirement of the unitig-to-color mapping from several Giga bytes to a few dozen Mega bytes [23].

3.1 Weakness of the state of the art

When indexing large pangenomes, the space taken by the (compressed) inverted index storing the distinct colors $\mathcal{C} = \{C_1, \ldots, C_M\}$ dominates the whole index space [23,27,32] (see also the breakdowns in Fig. 3a, at page 13). Efforts toward improving the space taken by c-dBGs should therefore be spent in devising better compression algorithms for the colors. To this end, there is a *crucial* property that is *not* exploited by current state-of-the-art solutions, including Fulgor [23], that can enable substantially improved effectiveness in color compression. This property is:

The genomes in a pangenome are very similar which, in turn, implies that the colors in $\mathcal{C}$ are also very similar (albeit all distinct).

By "similar" colors we mean that they share many (potentially, very long) identical integer sub-sequences. This property is not currently exploited because each color C_i is compressed *individually* from the other colors. For example, if color C_i shares a long sub-sequence with color C_j, this sub-sequence is actually represented *twice*, once in C_i *and again* C_j, therefore wasting space. This illustrative scenario clearly generalizes to more colors, increasing with pangenome redundancy and aggravating the memory usage of the index.

Example 1. Consider the following $M = 8$ distinct colors from Fig. 1, comprising a total of $\sum_{i=1}^{8} |C_i| = 47$ integers.

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

We underline some shared sub-sequences. For example, note that the sub-sequence $[3, 5, 9]$ appears in colors C_1, C_3, and C_4, while $[1, 12, 13, 16]$ appears in C_5 and C_7. Hence, it is wasteful to encode them, respectively, three times and twice.

The main objective of this note is to address this limitation by introducing a new family of colored compacted de Bruijn graphs (Section 4) that exploits the similarity property of the references in a pangenome to boost compression effectiveness.

We note that, in the context of *non*-compacted colored de Bruijn graphs, some representations have been introduced that attempt to exploit the similarity between colors to reduce the size of the index by means of various types of *referential* encoding [40,36]. Also, approaches that look for shared patterns across the "whole" inverted index have been explored with success in the Information Retrieval literature [41,42]. The approach we describe in the following is distinct from these existing schemes and, as we discuss later, it is possible that such approaches may even be "stacked" so as to further reduce the space required to represent the color information.

4 Meta-Colored compacted de Bruijn Graphs

In this section we propose a new family of colored compacted de Bruijn graphs – termed *meta-colored* compacted de Bruijn graphs, or Mac-dBGs – targeted to reduce the redundancy in the representation of the color information of large pangenome collections. The main novelty of the approach is that colors are compressed *holistically*, i.e., taking into account their redundancy across the *whole* set of colors, rather than compressing them individually as atomic lists of integers. We remark that all the state-of-the-art solutions that we reviewed in Section 3 compress colors only individually. In Section 5 we will show some preliminary results indicating that Mac-dBGs offer much better compression effectiveness compared to traditional c-dBG data structures.

4.1 Definition and data structure

Definition. Let $\mathcal{P} = \{\mathcal{P}_1, \ldots, \mathcal{P}_r\}$ be a partition of $[N] = \{1, \ldots, N\}$ for some $r \geq 1$. Let an order between the elements of each $\mathcal{P}_i = \{\mathrm{id}_{i,1}, \ldots, \mathrm{id}_{i,|\mathcal{P}_i|}\}$ be fixed. Any $\mathcal{P}$ induces a permutation $\pi : [N] \to [N]$, defined as

$$\pi(\mathrm{id}_{i,j}) := j + B_{i-1}$$

where

$$B_i = \begin{cases} \sum_{t=1}^{i} |\mathcal{P}_t| & i > 0 \\ 0 & i = 0 \end{cases},$$

for $i = 1, \ldots, r$ and $j = 1, \ldots, |\mathcal{P}_i|$. Let us now assume that the N reference identifiers have been permuted according to π, as well as the color sets $\mathcal{C} = \{C_1, \ldots, C_M\}$. Any $\mathcal{P}$ determines a partition of the N references in the collection $\mathcal{R}$ into r disjoint sets of references:

$$\mathcal{R}_1 = \{R_i | 0 = B_0 < i \leq B_1\}, \mathcal{R}_2 = \{R_i | B_1 < i \leq B_2\}, \ldots, \mathcal{R}_r = \{R_i | B_{r-1} < i \leq B_r = N\}.$$

Let

$$\mathcal{C}_i = \Big\{ \{z - B_{i-1} | z \in C_j \cap \{B_{i-1} + 1, \ldots, B_i\}\} \,|\, \forall C_j \in \mathcal{C} \Big\},$$

for $i = 1, \ldots, r$, with $M_i = |\mathcal{C}_i|$. In words, $\mathcal{C}_i$ is the set obtained by considering the distinct colors from the i-th partition $\mathcal{R}_i$ and noting that – by construction – they only comprise integers z such that $B_{i-1} < z \leq B_i$. Let us indicate with $C_{i,t}$ the t-th color from $\mathcal{C}_i$. These colors are henceforth named *partial* colors.

Given the partial color sets $\mathcal{C}_1, \ldots, \mathcal{C}_r$, it follows that each original color $C_j \in \mathcal{C}$ can be represented as a list C'_j of at most r *meta colors* – each meta color being a pointer (i, t) replacing the sub-sequence $[z + B_{i-1} | z \in C_{i,t}] \subseteq C_j$. From C'_j, it is straightforward to recover C_j: for each meta color $(i, t) \in C'_j$, sum B_{i-1} back to each decoded integer of $C_{i,t}$. If $N_p = \sum_{i=1}^{r} M_i$ is the total number of partial colors, then each meta color (i, t) can be indicated with just $\log_2(N_p)$ bits.

Intuitively, the representation via meta colors permits to encode the colors in $\mathcal{C}$ into smaller space for three reasons: (1) Potentially long sub-sequences, shared between several colors, are represented with $\log_2(N_p)$ bits; (2) Each partial color $C_{i,t}$ can be encoded more succinctly as it comprises smaller integers, compared to the original cases where it is part of a color C_j; (3) The total number of integers in $\mathcal{C}_1, \ldots, \mathcal{C}_r$ is *less* than that in the original $\mathcal{C}$, i.e., $\sum_{i=1}^{r} \sum_{t=1}^{M_i} |C_{i,t}| \leq \sum_{j=1}^{M} |C_j|$.

Data structure. Overall, the meta-colored compacted dBG (or Mac-dBG) representation comprises three data structures (plus also the sorted array $B[1..r] = [0, B_1, \ldots, B_{r-1}]$): (1) the meta colors $C'_1, \ldots, C'_M$, (2) the partial colors $\mathcal{C}_1, \ldots, \mathcal{C}_r$, and (3) the dictionary data structure $\mathcal{D}$.

Note that this representation 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. For example, we can use off-the-shelf compression methods for integer sequences [22] or those used in Fulgor [23] to encode the partial colors as well as the meta colors. Furthermore, we can use the SSHash dictionary data structure [39,38] to fully exploit the key unitig properties described in Section 2.2.

An important remark is that the described Mac-dBG layout is different from having a collection of r individual, smaller, c-dBGs – one for each partition $\mathcal{R}_i$. In fact, while this latter approach is easy to implement and generally applies to any index, it is also wasteful in both space and time regards. First, since each c-dBG has its own dictionary data structure, a k-mer is redundantly represented for as many times as the number of partitions where it appears, hence wasting space. Second, and perhaps even more importantly, a $\textsc{Color}(x)$ query will now have to aggregate all the partial results collected from all the individual c-dBGs. More specifically, the same $\textsc{Color}(x)$

query will be first answered by each individual c-dBG; then, partial results will be merged together. The Mac-dBG layout avoids these problems altogether: it has a *single* dictionary data structure and meta colors act as a *coordination* layer between the different partitions, embodied by the partial colors.

Example 2. Let us consider the $M = 8$ colors from Example 1, for $N = 16$. Let $r = 4$ and $\mathcal{P}_1 = \{1, 12, 13, 14, 16\}$, $\mathcal{P}_2 = \{3, 5, 9\}$, $\mathcal{P}_3 = \{7, 11\}$, $\mathcal{P}_4 = \{2, 4, 6, 8, 10, 15\}$, assuming we use the natural order between the integers to determine an order between the elements of each $\mathcal{P}_i$. Thus, we have $B_1 = 5$, $B_2 = 8$, $B_3 = 10$, and $B_4 = 16$. The induced permutation π is therefore $\pi(1) = 1$, $\pi(12) = 2$, $\pi(13) = 3$, $\pi(14) = 4$, *et cetera*:

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

The permuted colors are as follows (vertical bars represent the partial color boundaries $B_1, \ldots, B_4$).

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

We have the following four partial color sets:

$$\begin{aligned}
\mathcal{C}_1 &= \{C_{1,1} = [3], C_{1,2} = [1], C_{1,3} = [1, 3], C_{1,4} = [1, 2, 3, 4, 5], C_{1,5} = [2, 5]\} \\
\mathcal{C}_2 &= \{C_{2,1} = [1, 2, 3], C_{2,2} = [1], C_{2,3} = [1, 3]\} \\
\mathcal{C}_3 &= \{C_{3,1} = [2], C_{3,2} = [1, 2]\} \\
\mathcal{C}_4 &= \{C_{4,1} = [2, 5, 6], C_{4,2} = [1, 6], C_{4,3} = [5], C_{4,4} = [3], C_{4,5} = [3, 4], C_{4,6} = [4]\}
\end{aligned}$$

with $M_1 = 5$, $M_2 = 3$, $M_3 = 2$, $M_4 = 6$. Next, we show how the colors $\mathcal{C} = \{C_1, \ldots, C_8\}$ can be spelled by a proper concatenation of these partial color sets (assuming to sum B_{i-1} to the integers in $C_{i,t}$.) To the right of the arrow we show the rendering into meta colors C'_i.

$$\begin{aligned}
C_1 &= C_{1,1} \cdot C_{2,1} \cdot C_{3,1} \cdot C_{4,1} & \rightarrow \quad C'_1 &= [(1, 1), (2, 1), (3, 1), (4, 1)] \\
C_2 &= C_{2,2} \cdot C_{4,2} & \rightarrow \quad C'_2 &= [(2, 2), (4, 2)] \\
C_3 &= C_{1,2} \cdot C_{2,1} \cdot C_{3,2} \cdot C_{4,3} & \rightarrow \quad C'_3 &= [(1, 2), (2, 1), (3, 2), (4, 3)] \\
C_4 &= C_{1,3} \cdot C_{2,1} \cdot C_{3,2} & \rightarrow \quad C'_4 &= [(1, 3), (2, 1), (3, 2)] \\
C_5 &= C_{1,4} \cdot C_{2,3} \cdot C_{3,2} \cdot C_{4,4} & \rightarrow \quad C'_5 &= [(1, 4), (2, 3), (3, 2), (4, 4)] \\
C_6 &= C_{4,5} & \rightarrow \quad C'_6 &= [(4, 5)] \\
C_7 &= C_{1,4} \cdot C_{2,2} \cdot C_{3,1} \cdot C_{4,6} & \rightarrow \quad C'_7 &= [(1, 4), (2, 2), (3, 1), (4, 6)] \\
C_8 &= C_{1,5} & \rightarrow \quad C'_8 &= [(1, 5)]
\end{aligned}$$

Note that the sub-sequence $C_{1,4} = [1, 2, 3, 4, 5]$ shared between C_5 and C_7 is now represented *once* (in $\mathcal{C}_1$) as a direct consequence of partitioning and just 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. In this small example, we reduced the number of integers represented from 47 (Example 1) to $\sum_{i=1}^{4} |\mathcal{C}_i| = \sum_{i=1}^{4} \sum_{t=1}^{M_i} |C_{i,t}| = 30$. While the Mac-dBG data structure also includes the space for the meta colors, we will show in Section 5 that the representation is very effective. Lastly, Fig. 2 shows the Mac-dBG index layout for this example.

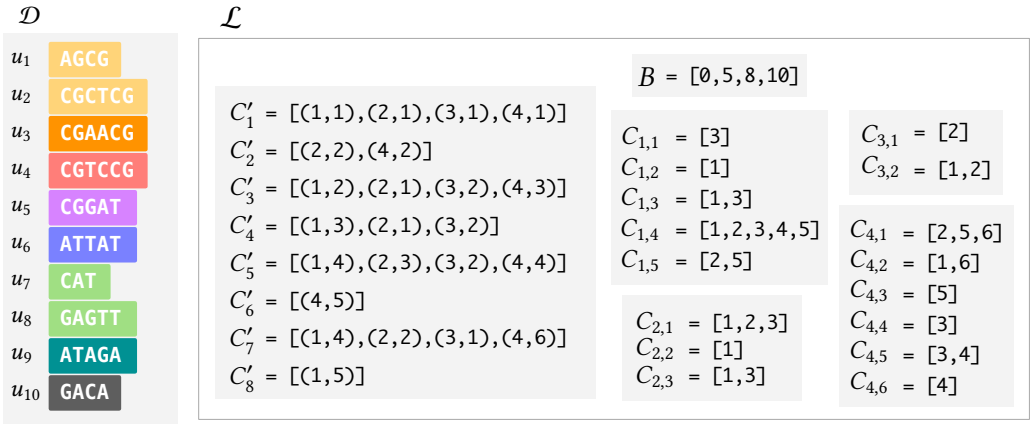


Fig. 2. Mac-dBG layout discussed in Example 2 for the colors of the c-dBG from Fig. 1.

4.2 The optimization problem

As evident from its definition, the effectiveness of a Mac-dBG crucially depends on the choice of the partition $\mathcal{P}$ 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 $N_p = \sum_{i=1}^{r} M_i$ 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 $\textsc{Cost}(C_{i,t}, \pi)$ be the encoding cost (in bits) of the partial color $C_{i,t}$ 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 $\mathcal{C}_j$ has many repeated sub-sequences. This will help in reducing the encoding cost for the partial colors, i.e., the quantity $\sum_{i=1}^{r} \sum_{t=1}^{M_i} \textsc{Cost}(C_{i,t}, \pi)$. 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 $\mathcal{C} = \{C_1, \ldots, C_M\}$ be the set of colors for the collection $\mathcal{R} = \{R_1, \ldots, R_N\}$. Determine the partition $\mathcal{P} = \{\mathcal{P}_1, \ldots, \mathcal{P}_r\}$ of $[N] = \{1, \ldots, N\}$ for some $r \geq 1$ and permutation $\pi : [N] \to [N]$ such that

$$\textsc{Cost}(\mathcal{C}, \mathcal{P}, \pi) := N_m \log_2(N_p) + \sum_{i=1}^{r} \sum_{t=1}^{M_i} \textsc{Cost}(C_{i,t}, \pi)$$

is minimum.

Depending upon the encoding scheme we choose, smaller values of $\textsc{Cost}(C_{i,t}, \pi)$ 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. [43] for the purpose of minimizing the cost of delta-encoded lists in inverted indexes. The BIMLOGA problem generalizes other known optimization problems, such as minimum logarithmic arrangement (MLOGA) and minimum logarithmic gap arrangement (MLOGGAPA) [44] problems, which are themselves modifications of the classic minimum linear arrangement problem

(MLA) (sometimes called the optimal linear arrangement or optimal linear ordering problem); all of these problems are known to be NP-hard [45,46,44,43]. For the BIMLOGA problem, Dhulipala et al. introduce an efficient heuristic for obtaining practically good orderings [43]. Note that BIMLOGA is a special case of MPA: that for $r = 1$ (no partitions) and $\textsc{Cost}(C_{i,t}, \pi)$ 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.

The problem looks difficult even if we opt for a different cost function that does not depend on the permutation π. For example, consider $\textsc{Cost}(C_{i,t}, \pi) = |\mathcal{R}_i|$ bits, that is, each partial color $C_{i,t}$ is encoded with the characteristic bit-vector of the set, which takes $|\mathcal{R}_i|$ bits. In this case, a solution to MPA could be determined by considering each possible set partition. The number of possible partitions of a set of size N is the N-th Bell number, B_N. We know that $B_N \leq e^{N^2/2}$, considering B_N as the N-th moment of a Poisson distribution with parameter 1. Since $\textsc{Cost}(C_{i,t}, \pi)$ can be computed in $O(|C_{i,t}|)$, $\textsc{Cost}(\mathcal{C}, \mathcal{P}, \pi)$ can be computed in $O(Z)$ time given a partition $\mathcal{P}$, where Z be the total number of integers in the color sets $\mathcal{C} = \{C_1, \ldots, C_M\}$ (we scan each color C_j and de-duplicate the partial color sets $C_{i,t}$ using hashing). Hence, a solution to MPA would naïvely take $O(e^{N^2/2}Z)$ time. In conclusion, while the MPA problem is NP-hard in general, it seems that even the problem of finding the optimal partition is difficult when an ordering of the references is fixed. A natural question is whether or not this variant of the problem is NP-hard too.

4.3 The SCPO framework

In this section we propose a construction algorithm for the Mac-dBG, 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 macro steps: (1) *Sketching*, (2) *Clustering*, (3) *Partitioning*, and (4) *Ordering* (SCPO), and can be though as a heuristic for the MPA optimization problem (Problem 2).

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 2.2) that each reference $R_i \in \mathcal{R}$ can be spelled by a proper concatenation (a "tiling") of the unitigs of the underlying compacted de Bruijn graph. If these unitigs are assigned unique identifiers by SSHash, it follows that each R_i can be seen as a list of unitig identifiers (with possible duplicates). The idea is that these integer lists are much shorter and take less space than the actual DNA references. The similarity of two references can then be determined by the number of shared integers in their unitig list representation. These integer lists can then be fed as input of a clustering algorithm (*Clustering* step 2 below.)

But we can do one more pre-processing step prior to the actual clustering: compute a *sketch* of each unitig list with the purpose of reducing even more the space/time cost of the clustering algorithm. There exist many different types of sketches based on hashing, such as *min-wise* [47] and its flavors [48] *hyper-log-log* [49], *set-sketch* [50], etc., each with slightly different properties and trade-offs. However, one property that is common to them all is that the similarity of two sets can be assessed in terms of the similarity between the two sketches, admitting some degree of approximation. These lossy sketches will effectively trade-off similarity accuracy for better computational costs but we argue that the trade-off is worth in order to scale up to large reference collections.

We remark that sketching for genomics has been already advocated as a very useful technique to assess sequence similarity [51,52]. Differently from these approaches that fill the sketches with the k-mers of the sequences, here we can afford much *smaller* sketches (or equivalently – for the same space budget – much *more accurate* sketches) since we fill the sketches with the identifiers of the unitigs. Since there are way fewer unitigs than k-mers, the computed sketches will be very

economical, hence permitting to scale to very large collection within the memory limit of a modest server machine.

2. Clustering. The computed sketches are then fed as input of a clustering algorithm. For example, we could use the well-known K-means algorithm, where the parameter K determines the number of desired clusters. In our case, this parameter will be equal to r – the number of partitions. Alternatively, we could use a divisive, hierarchical, approach that does not need an *a-priori* number of clusters to be supplied as input.

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 into $\mathcal{R}_1, \ldots, \mathcal{R}_r$, as well as $\mathcal{P}$, is uniquely determined.

4. Ordering. Finally, once the partitions $\mathcal{P}$ have been assigned, one may *reorder* the references that fall within a particular partition. While the goal of clustering and partitioning was to factor out repeated sub-patterns within the color table (i.e., the inverted index $\mathcal{L}$), the goal of the ordering step is to assign the references *within* each partition identifiers such that references that tend to co-occur within the partial colors of these partitions are assigned nearby identifiers. Ultimately, the utility of this reassignment is that, when the underlying inverted index uses delta coding, or similar methods that require fewer bits to represent lists of nearby numbers, the reordering enables more efficient encoding of these lists. There are many formulations related to the problem we wish to solve here, and one may view it as a seriation problem [53] or a minimum linear arrangement problem [45,54], or more directly in the inverted indexing context as the document identifier assignment problem [55]. Most formulations of these problems have been shown to be NP-hard to solve optimally, but in many cases, good heuristics or approximation algorithms exist [43], and in some cases when restrictions are placed on the input distances (e.g., being representable via a tree), efficient algorithms exist to obtain optimal orderings [56].

We note that the importance of the ordering step may depend greatly on the type of references being indexed. In the case of pangenomes, clustering the references into an appropriate number of partitions may provide a substantial benefit, and the additional benefit obtained by reassigning the reference identifiers within each partition may be marginal. On the other hand, when indexing more diverse collections of genomes (e.g., the Gut Bacteria collection mentioned below), the ordering itself may provide the majority of the compression benefit. We leave thorough experimentation of the best reordering approach – or at least the approach that provides the most desirable quality-to-runtime trade-off – to future work. However, we demonstrate below that, when available, a simple ordering by taxonomic lineage provides an ordering that substantially reduces the space required by the inverted index for a diverse metagenome collection.

(Optional) Further compression of partial colors with *super colors*. Even within the reordered partitions of partial colors we expect there to be substantial redundancy *between* the distinct partial colors. That is, while all partial colors are distinct within their partition, there may exist many partial colors that are *similar* (i.e., differ by the inclusion or exclusion of only a small number of references). Therefore, while we have not yet exploited this potential in the current index description, we expect these similarities may help to further compress the representation of these partial colors.

For example, *referential encoding* schemes, such as those adopted by Almodaresi et al. [40] or Karasikov et al. [36], are likely to be promising. Within each partition, we may define a small set of "super colors" – partial colors that are nearby, in terms of their Hamming distance, to many other partial colors within this partition. Subsequently, we can write down these super colors directly, and encode other similar colors by simply recording their differences with respect to their most similar super color. This process can even be done recursively, as in [40], to form a tree structure. A given partial color can then be decoded by traversing from the node corresponding to this identifier

up the tree until a fully-recorded color is encountered, and applying the differences collected on the edges along the traversal. As opposed to [40], where such a tree is based on a minimum spanning tree (MST) of a subgraph of the color graph, we note that one may also lift the restriction that the super colors themselves actually appear within the set of partial colors within a partition. In this case, one could possibly construct an even lower-weight Steiner tree [57,58] by placing vertices (i.e., super colors) that are close to many present partial colors within the partition. For now, we leave the question of assessing the added benefit of such a super color encoding, as well as the particular referential encoding scheme that works best in practice, to future work.

Finally in this section, it is worth noting that the approach we describe here for constructing Mac-dBGs bears a conceptual resemblance to the phylogenetic compression framework recently introduced by Břinda et al. [59]. At a high level, this owes to the fact that both approaches take advantage of well-known concepts in compression and Information Retrieval – namely that clustering and reordering are practical and effective heuristics for boosting compression. However, despite of the conceptual similarities (as well as similarities of both approaches to prior work exploiting clustering and reordering for improving compression), there are also critical distinctions between these approaches. First, while Břinda et al. [59] describe a general framework (applicable to a range of different tasks), we are primarily concerned with a specific optimization problem, leading our approach to be less general but likely more effective for the purpose of indexing. Additionally, while the approach by Břinda et al. focuses on clustering and organizing references so as to improve the construction of collections of disparate dictionaries, our approach adopts a single k-mer dictionary and instead induces a logical partitioning over the set of colors. This layout, as already pointed out in Section 4.1, allows to avoid having to record k-mers that appear in multiple partitions more than once. As a result, while the phylogenetic compression framework aims to scale to immense and highly-diverse collections of references, it anticipates a primarily disk-based index in which reference partitions are loaded, decompressed, and searched for matches, in a manner similar to a database (or BLAST [60]) search. On the other hand, the Mac-dBG approach we present here places a premium on query time, and aims to enable *in-memory indexing* with interactive lookups for the purpose of fast mapping of large collections of reads against the index.

5 Preliminary results

In this section we present some preliminary results conducted to assess the potential of the Mac-dBG, i.e., its space effectiveness and query efficiency. Throughout this section, we compare the Mac-dBG to state-of-the-art c-dBG indexes, namely Fulgor [23] and Themisto [32].

Datasets. We build Mac-dBGs with the proposed SCPO framework on the following three example pangenomes: 3,862 *E. Coli* genomes[5]; 10,000 and 50,000 *S. Enterica* genomes[6] from the collection by Blackwell et al. [61]. Additionally, we also include a much more diverse collection of 30,691 genomes assembled from human gut samples, originally published by Hiseni et al. [62].

Hardware and software. All experiments were run on a machine equipped with Intel Xeon Platinum 8276L CPUs (clocked at 2.20GHz), 500 GB of RAM running Ubuntu 18.04.6 LTS (GNU/Linux 4.15.0). The prototype Mac-dBG implementation is written in `C++17`, and available at https://github.com/jermp/fulgor/tree/dev. We compiled the codebase with gcc 11.1.0. For Themisto, we use the shipped compiled binaries (v3.1.1).

We report some implementation details. We relied on the existing Fulgor codebase to implement the Mac-dBG. Hence, we adopt the same compression methods we used in Fulgor to compress the partial colors; for the meta colors, instead, we just use $\log_2(N_p)$ bits per meta color; lastly, we use SSHash [38,39] as a k-mer dictionary data structure.

[5] https://zenodo.org/record/6577997

[6] http://ftp.ebi.ac.uk/pub/databases/ENA2018-bacteria-661k

Table 1. Index space in GB of Mac-dBGs for $k = 31$ and $r = 16$ partitions, on three example pangenomes and a more heterogeneous metagenome, in comparison to that of Fulgor and Themisto. The Gut Bacteria dataset has no partitions, but the Mac-dBG representation has reference identifiers assigned by taxonomic lineage. (In parentheses we indicate how many times the index space is smaller compared to Themisto's.)

	3,862 *E. Coli*	10,000 *S. Enterica*	50,000 *S. Enterica*	30,691 Gut Bacteria
Themisto	2.97	4.52	37.96	139.41
Fulgor	1.65 (1.80×)	1.83 (2.47×)	18.30 (2.07×)	36.77 (3.79×; 10.23 bits/int for colors)
Mac-dBG	0.64 (4.64×)	0.60 (7.53×)	4.40 (8.63×)	26.99 (5.17×; 4.23 bits/int for colors)

5.1 Index size

The result in Table 1 shows the total index size of Mac-dBGs on the tested datasets, for: $k = 31$, $r = 16$ partitions, and hyper-log-log sketches of size 2^{10} bytes each. On the largest pangenome, consisting of 50,000 *S. Enterica* genomes, the sketches take just $50{,}000 \times 2^{10}/10^6 = 51.2$ MB. We use the classic K-means algorithm [63,64], as implemented in scikit-learn [65], to perform the clustering. Under this configuration, the whole SCPO framework requires only a few minutes to run on the largest pangenome (but it would obviously take more time for larger collections). Although we do not discuss the construction algorithm in this note, we point out that the Mac-dBG data structure can be efficiently built from a Fulgor index, with no additional disk space.

Additionally, Table 1 also reports the total size of the Mac-dBG on a metagenome consisting of 31 bacteria from the human gut. Here, no partitioning is performed; rather, the reference identifiers are assigned based on the estimated taxonomic lineage (assigned by the original pipeline of Hiseni et al. [62]) of each reference. Briefly, we consider the root-to-leaf path of each reference in the estimated taxonomy, and turn each path into a tuple. Then, these tuples are sorted lexicographically to assign the reference identifier to the corresponding reference sequences. This particular approach is therefore "reference-guided". However, we anticipate that reference-free (i.e., fully unsupervised) approaches will work similarly well, as we simply wish to assign close identifiers to references that are similar in terms of their sequences, and therefore likely to co-occur within partial colors. This experiment shows the potential of re-assigning reference identifiers in a principled manner: even on heterogeneous collections like the Gut Bacteria dataset, the space for the colors improves from 10.23 bits/int to 4.23 bits/int (2.4× improvement).

While Fulgor outperforms Themisto by a large margin (being always ≈2× smaller, in consistency with the analysis in [23]), the net result of the experiment is that the prototype Mac-dBG is *even* $\approx 2-4\times$ *smaller than Fulgor.* We believe that this encouraging result represents a significant improvement on the problem of efficient representations for colored de Bruijn graphs.

Fig. 3 shows index space breakdowns for Mac-dBG in comparison to Fulgor. We remark that the dictionary component (the SSHash data structure) is common to both Mac-dBG's and Fulgor's implementations. As evident, the fraction of space for the colors is significantly reduced in Mac-dBG compared to Fulgor – sometimes, even dramatically. Consider, for example, the *S. Enterica* 10,000 pangenome: while the colors take 77% of the whole Fulgor index space, this percentage is now reduced to $(15 + 16)\% = 31\%$ for Mac-dBG, surprisingly making SSHash the "heaviest" component of the index. Another important consideration regards how the total space for the colors is divided among partial and meta colors. Meta colors almost take as much space as partial colors – surely a consequence of the simple $\log_2(N_p)$-bit encoding we are currently using. More succinct encodings can reduce the space for the meta colors substantially.

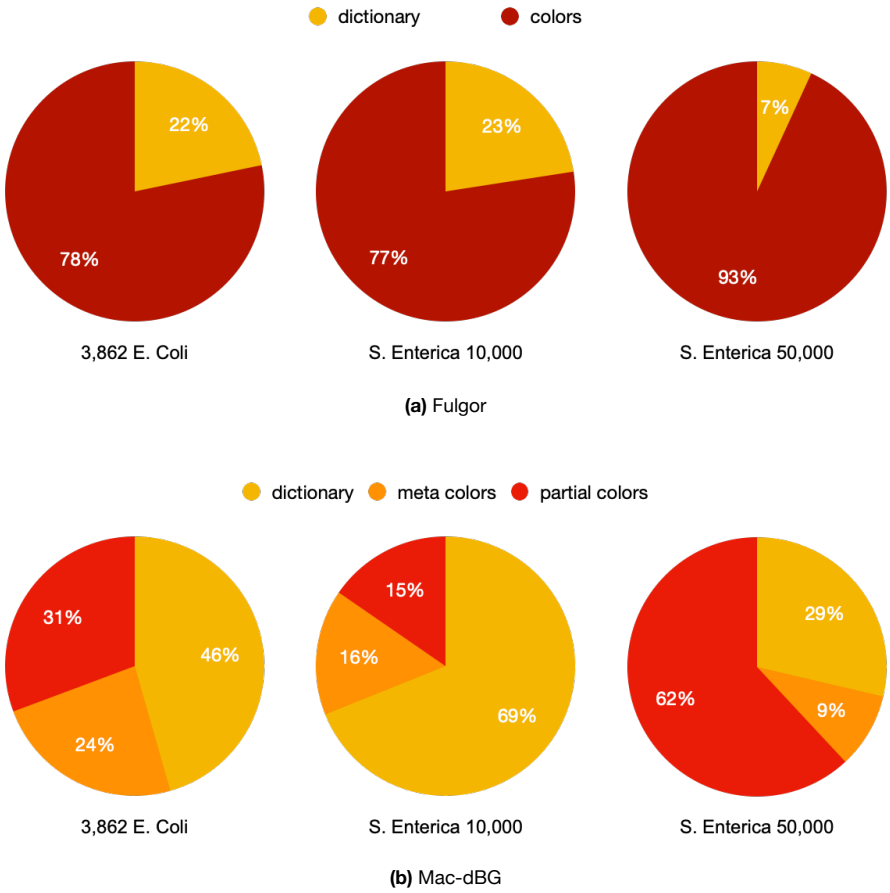


Fig. 3. Index space breakdowns.

5.2 Query speed

We now consider the speed of the Mac-dBG index when answering pseudoalignment queries using the *full-intersection* algorithm (see [23, Section 4.1] for details). We use high-hit workloads to benchmark the indexes. Specifically, we use the first reads from the following pair-end libraries as queries, in FASTQ format: SRR1928200[7] for *E. Coli*, SRR801268[8] for *S. Enterica*, and ERR321482[9] for Gut Bacteria. These files contain several million reads. For these workloads, we obtain hit rates (fraction of mapped read over the total number of queried reads) of 99.1%, 90.4%, 91.2%, and 92.9% on *E. Coli*, *S. Enterica* 10,000, *S. Enterica* 50,000, and Gut Bacteria, respectively.

The query speed results are reported in Table 2. As expected, when meta color lists are not very long (recall, from Section 4.1, that we have meta colors of length at most $r = 16$ in this experiment), Mac-dBGs do not sacrifice query efficiency compared to the fastest index, Fulgor. Indeed, they achieve the same performance of the larger *S. Enterica* collections but impose a minor slowdown on the *E. Coli* pangenome (probably due to its much higher hit-rate). The Mac-dBG built on the Gut Bacteria obviously does not introduce any penalty compared to Fulgor since

[7] https://www.ebi.ac.uk/ena/browser/view/SRR1928200
[8] https://www.ebi.ac.uk/ena/browser/view/SRR801268
[9] https://www.ebi.ac.uk/ena/browser/view/ERR321482

Table 2. Total query time as elapsed time reported by `/usr/bin/time`, using 16 processing threads for all indexes. The read-mapping output is written to `/dev/null` for this experiment. Results are relative to the full-intersection query mode. All reported timings are relative to a second run of the experiment, when the index is loaded faster from the disk cache.

	3,862 *E. Coli*	10,000 *S. Enterica*	50,000 *S. Enterica*	30,691 Gut Bacteria
	mm:ss	mm:ss	mm:ss	mm:ss
Themisto	05:50	07:18	41:25	02:45
Fulgor	02:16	02:20	19:15	01:16
Mac-dBG	03:30	02:34	19:17	01:00

it uses one partition only (we reported its query time for completeness). It is expected, however, that using more partitions and more succinct encodings for meta colors could raise the query overhead of Mac-dBGs compared to Fulgor but reduce the index space. This trade-off is yet to be explored.

On the other hand, we also note that, due to the manner in which the partitions factorize the space of references, Mac-dBGs can offer *even faster* query times than traditional c-dBGs if a two-level intersection algorithm is employed. First, only meta colors are intersected (hence, without any need to access the partial colors) to determine the partitions in common to all inverted lists being intersected. Second, the corresponding partial colors are decoded only for these common partitions. This optimization could be very beneficial when accessing meta colors relative to unitigs belonging to a specific partition. We remark that the query times reported in Table 2 are achieved *without* this optimization that we plan to implement.

6 Research questions

We conclude this note with a rich list of research questions (both theoretical and practical) about the introduced Mac-dBG representation that we intend to answer in the near future.

RQ 1. *Do different sketching/clustering algorithms have a meaningful impact on the index size?*
In our preliminary results from Section 5 we use the classic K-means clustering algorithm over hyper-log-log sketches, but other clustering/sketching algorithm combinations could be tested, and may yield meaningfully different solutions to the optimization problem.

RQ 2. *How should reference identifiers be assigned to references within the same cluster $\mathcal{R}_i$?*
In our preliminary results, apart from the Gut Bacteria metagenome, we assigned identifiers at random within the same cluster. (In the Gut Bacteria dataset, we assigned reference order based on imputed taxonomic lineages.) However, one can likely optimize these assignments generally, and in a reference-free/unsupervised manner, to reduce the index size adopting, e.g., techniques similar to optimal leaf re-ordering [56] or recursive bipartite graph partitioning [43] based on the reference similarities already computed from the sketches used for partitioning.

RQ 3. *How should one automatically determine a proper value of r, and how strong is the dependency of the index size upon r?*
As already explained, we expect a proper value of r to be not too small (so as to increase the likeliness of having identical sub-sequences in the partial colors) and not too large (so as to reduce the storage space of the meta-colors). Therefore, the choice of r appears to be fundamental for the final space effectiveness of the Mac-dBG. While evaluating the exact effect of a particular choice of r on the final index may be computationally intensive,

there may be a sufficiently faithful and easy-to-evaluate proxy function, that would allow efficiently optimizing over this parameter.

RQ 4. *Can we reduce the space even more by exploiting the fact that partial colors (i.e., colors within the same partition) are similar?*
There is strong evidence from prior work [40,36] that coherence or similarity *between* distinct colors allows for effective compression in a manner that is different from what is obtained via the current SCPO framework. A natural question is the degree to which such approaches can be effectively "stacked", so that one could apply the SCPO framework, and then, within each partition, further compress the partial colors using a referential encoding like the minimum spanning tree-based encoding of Almodaresi et al. [40] or the delta encoding of Karasikov et al. [36]. Unlike the current meta-color scheme, these approaches can reduce the query speed, and so the potential space-time tradeoffs are worth exploring. Further, one should understand whether it is more convenient to: (1) compress the partial colors with an entirely separate method, or (2) use the Mac-dBG representation *recursively* on each color partition $\mathcal{C}_i$.

RQ 5. *Are there any specific properties of the colors that we can leverage to make the SCPO framework even more efficient?*
For example, we exploit that fact that we can build sketches directly over the unitig identifiers as given by the SSHash dictionary to reduce their memory usage. It would be interesting to design an algorithm that clusters these sketches even *faster* than K-means.

RQ 6. *How do different compression methods impact the space/time trade-offs of Mac-dBGs?*
Clearly, there is no single compression algorithm that works best for all needs; rather, one should choose a space/time trade-off suitable for the application at hand. In principle, one could completely ignore query efficiency and just support decoding of the colors in order (i.e., first C_1, then C_2, and so on) with the purpose of optimizing index space. This extreme trade-off is gaining popularity in recent years, with several proposals available [66,18], given the massive size of genomic collections. The goal of such algorithms is therefore to save disk space/energy and make more economical the sharing of large datasets.

RQ 7. *What is the information-theoretic space lower bound on the representation of any Mac-dBG?*
This is a question of great importance. On one hand, lower bounds exist for the (arguably simpler) case of un-colored de Bruijn graphs, both for more general membership data structures [67] and navigational data structures [68]. Moreover, practical data structures exist to represent de Bruijn graphs using only a few bits per k-mer and supporting membership [35,38,33] even coming close to the navigational space lower bound [33]. On the other hand, there is a distinct lack of similar information theoretic lower bounds on the representation of the colored compacted de Bruijn graph. Yet, this appears to be a needed tool to understand how much improvement might be expected, and what size of representation might be achieved.

Acknowledgements

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. Bo Liu, Hongzhe Guo, Michael Brudno, and Yadong Wang. deBGA: read alignment with de bruijn graph-based seed and extension. *Bioinformatics*, 32(21):3224–3232, July 2016.
2. Nicolas L Bray, Harold Pimentel, Páll Melsted, and Lior Pachter. Near-optimal probabilistic rna-seq quantification. *Nature biotechnology*, 34(5):525–527, 2016.
3. L Schaeffer, H Pimentel, N Bray, P Melsted, and L Pachter. Pseudoalignment for metagenomic read assignment. *Bioinformatics*, 33(14):2082–2088, 02 2017.
4. Fatemeh Almodaresi, Hirak Sarkar, Avi Srivastava, and Rob Patro. A space and time-efficient index for the compacted colored de Bruijn graph. *Bioinformatics*, 34(13):i169–i177, 2018.
5. Mark Reppell and John Novembre. Using pseudoalignment and base quality to accurately quantify microbial community composition. *PLOS Computational Biology*, 14(4):1–23, 04 2018.
6. Tommi Mäklin, Teemu Kallonen, Sophia David, Christine J Boinett, Ben Pascoe, Guillaume Méric, David M Aanensen, Edward J Feil, Stephen Baker, Julian Parkhill, 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.
7. Fatemeh Almodaresi, Mohsen Zakeri, and Rob Patro. PuffAligner: a fast, efficient and accurate aligner based on the pufferfish index. *Bioinformatics*, 37(22):4048–4055, June 2021.
8. Giorgos Skoufos, Fatemeh Almodaresi, Mohsen Zakeri, Joseph N. Paulson, Rob Patro, Artemis G. Hatzigeorgiou, and Ioannis S. Vlachos. AGAMEMNON: an accurate metaGenomics and MEtatranscriptoMics quaNtificatiON analysis suite. *Genome Biology*, 23(1), January 2022.
9. Ilia Minkin and Paul Medvedev. Scalable multiple whole-genome alignment and locally collinear block construction with SibeliaZ. *Nature Communications*, 11(1), December 2020.
10. Ilia Minkin and Paul Medvedev. Scalable pairwise whole-genome homology mapping of long genomes with BubbZ. *iScience*, 23(6):101224, June 2020.
11. Siavash Sheikhizadeh, M. Eric Schranz, Mehmet Akdel, Dick de Ridder, and Sandra Smit. PanTools: representation, storage and exploration of pan-genomic data. *Bioinformatics*, 32(17):i487–i493, August 2016.
12. Alan Cleary, Thiruvarangan Ramaraj, Indika Kahanda, Joann Mudge, and Brendan Mumey. Exploring Frequented Regions in Pan-Genomic Graphs. *IEEE/ACM Transactions on Computational Biology and Bioinformatics*, 16(5):1424–1435, September 2019.
13. Buwani Manuweera, Joann Mudge, Indika Kahanda, Brendan Mumey, Thiruvarangan Ramaraj, and Alan Cleary. Pangenome-Wide Association Studies with Frequented Regions. In *Proceedings of the 10th ACM International Conference on Bioinformatics, Computational Biology and Health Informatics*. ACM, September 2019.
14. Kadir Dede and Enno Ohlebusch. Dynamic construction of pan-genome subgraphs. *Open Computer Science*, 10(1):82–96, April 2020.
15. John A. Lees, T. Tien Mai, Marco Galardini, Nicole E. Wheeler, Samuel T. Horsfield, Julian Parkhill, and Jukka Corander. Improved Prediction of Bacterial Genotype-Phenotype Associations Using Interpretable Pangenome-Spanning Regressions. *mBio*, 11(4), August 2020.
16. Roland Wittler. Alignment- and reference-free phylogenomics with colored de Bruijn graphs. *Algorithms for Molecular Biology*, 15(1), April 2020.
17. Nina Luhmann, Guillaume Holley, and Mark Achtman. BlastFrost: fast querying of 100, 000s of bacterial genomes in bifrost graphs. *Genome Biology*, 22(1), January 2021.
18. Amatur Rahman, Yoann Dufresne, and Paul Medvedev. Compression algorithm for colored de bruijn graphs. *bioRxiv*, pages 2023–05, 2023.
19. Shoshana Marcus, Hayan Lee, and Michael C Schatz. Splitmem: a graphical algorithm for pan-genome analysis with suffix skips. *Bioinformatics*, 30(24):3476–3483, 2014.
20. Uwe Baier, Timo Beller, and Enno Ohlebusch. Graphical pan-genome analysis with compressed suffix trees and the burrows–wheeler transform. *Bioinformatics*, 32(4):497–504, 2016.
21. Justin Zobel and Alistair Moffat. Inverted files for text search engines. *ACM Computing Surveys (CSUR)*, 38(2):6–es, 2006.
22. Giulio Ermanno Pibiri and Rossano Venturini. Techniques for inverted index compression. *ACM Computing Surveys (CSUR)*, 53(6):125:1–125:36, 2021.

23. Jason Fan, Noor Pratap Singh, Jamshed Khan, Giulio Ermanno Pibiri, and Rob Patro. Fulgor: A fast and compact k-mer index for large-scale matching and color queries. In *23-rd International Workshop on Algorithms in Bioinformatics. To appear*, 2023.
24. Jamshed Khan and Rob Patro. Cuttlefish: fast, parallel and low-memory compaction of de Bruijn graphs from large-scale genome collections. *Bioinformatics*, 37(Supplement_1):i177–i186, 2021.
25. Jamshed Khan, Marek Kokot, Sebastian Deorowicz, and Rob Patro. Scalable, ultra-fast, and low-memory construction of compacted de Bruijn graphs with Cuttlefish 2. *Genome Biology*, 23(1):190, 2022.
26. Amatur Rahman and Paul Medvedev. Assembler artifacts include misassembly because of unsafe unitigs and underassembly because of bidirected graphs. *Genome Research*, 32(9):1746–1753, July 2022.
27. Guillaume Holley and Páll Melsted. Bifrost: highly parallel construction and indexing of colored and compacted de Bruijn graphs. *Genome biology*, 21(1):1–20, 2020.
28. Jason Fan, Jamshed Khan, Giulio Ermanno Pibiri, and Rob Patro. Spectrum preserving tilings enable sparse and modular reference indexing. In *Research in Computational Molecular Biology*, pages 21–40, 2023.
29. Ilia Minkin, Son Pham, and Paul Medvedev. TwoPaCo: an efficient algorithm to build the compacted de Bruijn graph from many complete genomes. *Bioinformatics*, 33(24):4024–4032, 2017.
30. Andrea Cracco and Alexandru I Tomescu. Extremely fast construction and querying of compacted and colored de bruijn graphs with GGCAT. *Genome Research*, pages gr–277615, 2023.
31. Camille Marchet, Christina Boucher, Simon J Puglisi, Paul Medvedev, Mikaël Salson, and Rayan Chikhi. Data structures based on k-mers for querying large collections of sequencing data sets. *Genome Research*, 31(1):1–12, 2021.
32. Jarno N Alanko, Jaakko Vuohtoniemi, Tommi Mäklin, and Simon J Puglisi. Themisto: a scalable colored k-mer index for sensitive pseudoalignment against hundreds of thousands of bacterial genomes. *Bioinformatics*, 39(Supplement_1):i260–i269, June 2023.
33. Jarno N. Alanko, Simon J. Puglisi, and Jaakko Vuohtoniemi. Small searchable k-spectra via subset rank queries on the spectral burrows-wheeler transform. *SIAM Conference on Applied and Computational Discrete Algorithms (ACDA23)*, pages 225–236, 2023.
34. Mikhail Karasikov, Harun Mustafa, Amir Joudaki, Sara Javadzadeh-no, Gunnar Rätsch, and André Kahles. Sparse Binary Relation Representations for Genome Graph Annotation. *Journal of Computational Biology*, 27(4):626–639, April 2020.
35. Alexander Bowe, Taku Onodera, Kunihiko Sadakane, and Tetsuo Shibuya. Succinct de Bruijn graphs. In *International Workshop on Algorithms in Bioinformatics (WABI)*, pages 225–235. Springer, 2012.
36. Mikhail Karasikov, Harun Mustafa, Gunnar Rätsch, and André Kahles. Lossless indexing with counting de Bruijn graphs. *Genome Research*, 32(9):1754–1764, May 2022.
37. Daniel Lemire, Owen Kaser, Nathan Kurz, Luca Deri, Chris O'Hara, François Saint-Jacques, and Gregory Ssi-Yan-Kai. Roaring bitmaps: Implementation of an optimized software library. *Software: Practice and Experience*, 48(4):867–895, 2018.
38. Giulio Ermanno Pibiri. Sparse and skew hashing of k-mers. *Bioinformatics*, 38(Supplement_1):i185–i194, 06 2022.
39. Giulio Ermanno Pibiri. On weighted k-mer dictionaries. *Algorithms for Molecular Biology*, 18(3), 2023.
40. Fatemeh Almodaresi, Prashant Pandey, Michael Ferdman, Rob Johnson, and Rob Patro. An Efficient, Scalable, and Exact Representation of High-Dimensional Color Information Enabled Using de Bruijn Graph Search. *Journal of Computational Biology*, 27(4):485–499, April 2020.
41. Giulio Ermanno Pibiri and Rossano Venturini. Clustered Elias-Fano indexes. *ACM Transactions on Information Systems*, 36(1):2:1–2:33, 2017.
42. Giulio Ermanno Pibiri, Matthias Petri, and Alistair Moffat. Fast dictionary-based compression for inverted indexes. In *Proceedings of the Twelfth ACM International Conference on Web Search and Data Mining*, pages 6–14, 2019.
43. Laxman Dhulipala, Igor Kabiljo, Brian Karrer, Giuseppe Ottaviano, Sergey Pupyrev, and Alon Shalita. Compressing graphs and indexes with recursive graph bisection. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 1535–1544, 2016.
44. Flavio Chierichetti, Ravi Kumar, Silvio Lattanzi, Michael Mitzenmacher, Alessandro Panconesi, and Prabhakar Raghavan. On compressing social networks. In *Proceedings of the 15th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 219–228, 2009.

45. L. H. Harper. Optimal assignments of numbers to vertices. *Journal of the Society for Industrial and Applied Mathematics*, 12(1):131–135, March 1964.
46. Michael R. Garey and David S. Johnson. *Computers and Intractability; A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., USA, 1990.
47. Andrei Z Broder. On the resemblance and containment of documents. In *Proceedings. Compression and Complexity of SEQUENCES 1997 (Cat. No. 97TB100171)*, pages 21–29. IEEE, 1997.
48. Edith Cohen. Min-hash sketches. In *Encyclopedia of Algorithms*, pages 1282–1287. Springer New York, 2016.
49. Philippe Flajolet, Éric Fusy, Olivier Gandouet, and Frédéric Meunier. Hyperloglog: the analysis of a near-optimal cardinality estimation algorithm. In *Discrete Mathematics and Theoretical Computer Science*, pages 137–156. Discrete Mathematics and Theoretical Computer Science, 2007.
50. Otmar Ertl. Setsketch: Filling the gap between minhash and hyperloglog. *arXiv preprint arXiv:2101.00314*, 2021.
51. Daniel N Baker and Ben Langmead. Dashing: fast and accurate genomic distances with hyperloglog. *Genome biology*, 20:1–12, 2019.
52. N Tessa Pierce, Luiz Irber, Taylor Reiter, Phillip Brooks, and C Titus Brown. Large-scale sequence comparisons with sourmash. *F1000Research*, 8, 2019.
53. Innar Liiv. Seriation and matrix reordering methods: An historical overview. *Statistical Analysis and Data Mining*, pages n/a–n/a, 2010.
54. D. Adolphson and T. C. Hu. Optimal Linear Ordering. *SIAM Journal on Applied Mathematics*, 25(3):403–423, November 1973.
55. D. Blandford and G. Blelloch. Index compression through document reordering. In *Data Compression Conference*. IEEE Comput. Soc, 2002.
56. Ziv Bar-Joseph, David K. Gifford, and Tommi S. Jaakkola. Fast optimal leaf ordering for hierarchical clustering. *Bioinformatics*, 17(suppl_1):S22–S29, June 2001.
57. S. E. Dreyfus and R. A. Wagner. The Steiner problem in graphs. *Networks*, 1(3):195–207, 1971.
58. Anatoly Yur'evich Levin. Algorithm for the shortest connection of a group of graph vertices. In *Doklady Akademii Nauk*, volume 200, pages 773–776. Russian Academy of Sciences, 1971.
59. Karel Břinda, Leandro Lima, Simone Pignotti, Natalia Quinones-Olvera, Kamil Salikhov, Rayan Chikhi, Gregory Kucherov, Zamin Iqbal, and Michael Baym. Efficient and Robust Search of Microbial Genomes via Phylogenetic Compression. April 2023.
60. Stephen F Altschul, Warren Gish, Webb Miller, Eugene W Myers, and David J Lipman. Basic local alignment search tool. *Journal of Molecular Biology*, 215(3):403–410, 1990.
61. Grace A. Blackwell, Martin Hunt, Kerri M. Malone, Leandro Lima, Gal Horesh, Blaise T. F. Alako, Nicholas R. Thomson, and Zamin Iqbal. Exploring bacterial diversity via a curated and searchable snapshot of archived DNA sequences. *PLOS Biology*, 19(11):1–16, 11 2021.
62. Pranvera Hiseni, Knut Rudi, Robert C Wilson, Finn Terje Hegge, and Lars Snipen. HumGut: a comprehensive human gut prokaryotic genomes collection filtered by metagenome data. *Microbiome*, 9(1):1–12, 2021.
63. S. Lloyd. Least squares quantization in PCM. *IEEE Transactions on Information Theory*, 28(2):129–137, March 1982.
64. Charles Elkan. Using the triangle inequality to accelerate k-means. In *Proceedings of the Twentieth International Conference on International Conference on Machine Learning*, ICML'03, page 147–153. AAAI Press, 2003.
65. F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.
66. Amatur Rahman, Rayan Chikhi, and Paul Medvedev. Disk compression of k-mer sets. *Algorithms for Molecular Biology*, 16(1):10, 2021.
67. Thomas C. Conway and Andrew J. Bromage. Succinct data structures for assembling large genomes. *Bioinformatics*, 27(4):479–486, January 2011.
68. Rayan Chikhi, Antoine Limasset, Shaun Jackman, Jared T. Simpson, and Paul Medvedev. On the Representation of De Bruijn Graphs. *Journal of Computational Biology*, 22(5):336–352, May 2015.