An Index for Sequencing Reads Based on the Colored de Bruijn Graph

Diego Díaz-Domínguez[1,2(✉)]

[1] Department of Computer Science, University of Chile, Santiago, Chile
diediaz@dcc.uchile.cl
[2] CeBiB — Center for Biotechnology and Bioengineering, University of Chile, Santiago, Chile

Abstract. In this article, we show how to transform a colored de Bruijn graph (dBG) into a practical index for processing massive sets of sequencing reads. Similar to previous works, we encode an instance of a colored dBG of the set using *BOSS* and a color matrix C. To reduce the space requirements, we devise an algorithm that produces a smaller and more sparse version of C. The novelties in this algorithm are (i) an incomplete coloring of the graph and (ii) a greedy coloring approach that tries to reuse the same colors for different strings when possible. We also propose two algorithms that work on top of the index; one is for reconstructing reads, and the other is for contig assembly. Experimental results show that our data structure uses about half the space of the plain representation of the set (1 Byte per DNA symbol) and that more than 99% of the reads can be reconstructed just from the index.

Keywords: de Bruijn graphs · DNA sequencing · Compact data structures

1 Introduction

A set of *sequencing reads* is a massive collection $R = \{R_1, \ldots, R_n\}$ of n overlapping short strings that together encode the sequence of a DNA sample. Analyzing this kind of data allows scientists to uncover complex biological processes that otherwise could not be studied. There are many ways for extracting information from a set of reads (see [27] for review). However, in most of the cases, the process can be reduced to build a *de Bruijn graph* (dBG) of the collection and then search for graph paths that spell segments of the source DNA (see [6,15,28] for some examples).

Briefly, a dBG is a directed labeled graph that stores the transitions of the substrings of size k, or *kmers*, in R. Constructing it is relatively simple, and

Partially supported by Basal Funds FB0001, Conicyt, Chile; by a Conicyt Ph.D. Scholarship; by Fondecyt Grants 1-171058 and 1-170048, Chile; and by the European Union's Horizon 2020 research and innovation programme under the Marie Sklodowska-Curie [grant agreement No. 690941].

N. R. Brisaboa and S. J. Puglisi (Eds.): SPIRE 2019, LNCS 11811, pp. 304–321, 2019.
https://doi.org/10.1007/978-3-030-32686-9_22

the resulting graph usually uses less space than the input text. Nevertheless, this data structure is lossy, so it is not always possible to know if the label of a path matches a substring of the source DNA. The only paths that fulfill this property are those in which all nodes, except the first and last, have indegree and outdegree one [16]. Still, they represent just a fraction of the complete dBG.

More branched parts of the graph are also informative, but traverse them requires extra information to avoid spelling incorrect sequences. A simple solution is to augment the dBG with colors, in other words, we assign a particular color c_i to every string $R_i \in R$, and then we store the same c_i in every edge that represents a kmer of R_i. In this way, we can walk over the graph always following the successor node colored with the same color of the current node.

The idea of coloring dBGs was first proposed by Iqbal et al. [15]. Their data structure, however, contemplated a union dBG built from several string collections, with colors assigned to the collections rather than particular strings. Considering the potential applications of colored dBGs, Boucher et al. [4] proposed a succinct version of the data structure of Iqbal et al. In their index, called VARI, the topology of the graph is encoded using *BOSS* [5], and the colors are stored separately from the dBG in a binary matrix C, in which the rows represent the kmers and the columns represent the colors. Since the work of Boucher et al., some authors have tried to compress and manipulate C even further; including that of [2,13,25], while others, such as [21] and [22] have proposed methods to store compressed and dynamic versions C.

An instance of a colored dBG for a single set R can also be encoded using a color matrix. The only difference though is that the number of columns is proportional to the number of sequences in R. Assigning a particular color to every sequence is not a problem if the collection is of small or moderated size. However, massive datasets are rather usual in Bioinformatics, so even using a succinct representation of C might not be enough. One way to reduce the number of columns is to reuse colors for those sequences that do not share any kmer in the dBG. Alpanahi et al. [1] addressed this problem, and showed that it is unlikely that the minimum-size coloring can be approximated in polynomial time.

Alpanahi et al. also proposed a heuristic for recoloring the colored dBG of a set of sequences that, in practice, dramatically reduces the number of colors when R is a set of sequencing reads. Their coloring idea, however, might still produce incorrect sequences, so the applications of their version of the colored dBG are still limited.

Our Contributions. In this article, we show how to use a colored dBG to store and analyze a collection of sequencing reads succinctly. Similarly to VARI, we use *BOSS* and the color matrix C to encode the data. However, we reduce the space requirements by partially coloring the dBG and greedily reusing the same colors for different reads when possible. We also propose two algorithms that work on top of the data structure, one for reconstructing the reads directly from the dBG and other for assembling contigs. We believe that these two algorithms can serve as a base to perform Bioinformatics analyses in compressed space.

Our experimental results show that on average, the percentage of nodes in *BOSS* that need to be colored is about 12.4%, the space usage of the whole index is about half the space of the plain representation of R (1 Byte/DNA symbol), and that more than 99% of the original reads can be reconstructed from the index.

2 Preliminaries

DNA Strings. A DNA sequence R is a string over the alphabet $\Sigma = \{\mathtt{a}, \mathtt{c}, \mathtt{g}, \mathtt{t}\}$ (which we map to $[2..5]$), where every symbol represents a particular nucleotide in a DNA molecule. The DNA *complement* is a permutation $\pi[2..\sigma]$ that reorders the symbols in Σ exchanging `a` with `t` and `c` with `g`. The *reverse complement* of R, denoted R^{rc}, is a string transformation that reverses R and then replaces every symbol $R[i]$ by its complement $\pi(R[i])$. For technical convenience we add to Σ the so-called *dummy* symbol `$`, which is always mapped to 1.

de Bruijn Graph. A de Bruijn graph (dBG) [7] of the string collection $R = \{R_1, \ldots, R_n\}$ is a labeled directed graph $G = (V, E)$ that encodes the transitions between the substrings of size k of R, where k is a parameter. Every node $v \in V$ is labeled with a unique $k-1$ substring of R. Two nodes v and u are connected by a directed edge $(v, u) \in E$ if the $k-2$ suffix of v overlaps the $k-2$ prefix of u and the k-string resulted from the overlap exists as substring in R. The label of the edge is the last symbol of the label of node u.

Rank and Select Data Structures. Given a sequence $B[1..n]$ of elements over the alphabet $\Sigma = [1..\sigma]$, $\mathtt{rank}_b(B, i)$ with $i \in [1..n]$ and $b \in \Sigma$, returns the number of times the element b occurs in $B[1..i]$, while $\mathtt{select}_b(B, i)$ returns the position of the ith occurrence of b in B. For binary alphabets, B can be represented in $n + o(n)$ bits so that `rank` and `select` are solved in constant time [9]. When B has $m \ll n$ 1s, a compressed representation using $m \lg \frac{n}{m} + \mathcal{O}(m) + o(n)$ bits, still solving the operations in constant time, is of interest [26].

BOSS Index. The *BOSS* data structure [5] is a succinct representation for dBGs based on the *Burrows-Wheeler Transform* (*BWT*) [8]. In this index, the labels of the nodes are regarded as rows in a matrix and sorted in reverse lexicographical order, i.e., strings are read from right to left. Suffixes and prefixes in R of size below $k-1$ are also included in the matrix by padding them with `$` symbols in the right size (for suffixes) or the left side (for prefixes). These padded nodes are also called *dummy*. The last column of the matrix is stored as an array $K[1..\sigma]$, with $K[i]$ being the number of labels lexicographically smaller than any other label ending with character i. Additionally, the symbols of the outgoing edges of every node are sorted and then stored together in a single array E. A bit vector $B[1..|E|]$ is also set to mark the position in E of the first outgoing symbol of each node. The complete index is thus composed of the vectors E, K, and B. It can be stored in $|E|(\mathcal{H}_0(E) + \mathcal{H}_0(B))(1 + o(1)) + \mathcal{O}(\sigma \log n)$ bits,

where $\mathcal{H}_0$ is the zero-order empirical entropy [23, Sec 2.3]. This space is reached with a Huffman-shaped Wavelet Tree [18] for E, a compressed bitmap [26] for B (as it is usually very dense), and a plain array for K. Bowe et al. [5] defined the following operations over $BOSS$ to navigate the graph:

- `outdegree`(v): number of outgoing edges of v.
- `forward`(v, a): node reached by following an edge from v labeled with a.
- `indegree`(v): number of incoming edges of v.
- `backward`(v): list of the nodes with an outgoing edge to v.
- `nodeLabel`(v): label of node v.

The first four operations can be answered in $\mathcal{O}(\log \sigma)$ time while the last one takes $\mathcal{O}(k \log \sigma)$ time. For our purposes, we also define the following operations:

- `forward_r`(v, r): node reached by following the r-*th* outgoing edge of v in lexicographical order.
- `label2Node`(S): identifier in $BOSS$ of the node labeled with the $(k-1)$-string S.

The function `forward_r` is a small variation `forward`, and it maintains the original time, while the function `label2Node` is the opposite of `nodeLabel`, but it also maintain its complexity in $\mathcal{O}(k \log \sigma)$ time.

Graph Coloring. The problem of coloring a graph $G = (V, E)$ consists of assigning an integer $c(v) \in [1..\omega]$ to each node $v \in V$ such that (i) no adjacent nodes have the same color and (ii) ω is minimal. The coloring is *complete* if all the nodes of the graph are assigned with one color, and it is *proper* if constraint (i) is met for each node. The chromatic number of a graph G, denoted by $\chi(G)$, is the minimum number of colors required to generate a coloring that is complete and proper. A coloring using exactly $\chi(G)$ colors is considered to be optimal. Determining if there is a feasible ω-coloring for G is well known to be NP-complete, while the problem of inferring $\chi(G)$ is NP-hard [17].

Colored dBG. The first version of the colored dBG [15] was described as a union graph G built from several dBGs of different string collections. The edges in G that encode the kmers of the i-*th* collection are assigned the color i. The compacted version of this graph [4] represents the topology of G using the $BOSS$ index and the colors using a binary matrix C, where the position $C[i, j]$ is set to true if the kmer represented by the i-*th* edge in the ordering of $BOSS$ is assigned color j. The rows of C are then stored using the compressed representation for bit vectors of [26], or using Elias-Fano encoding [10,11,24] if the rows are very sparse. In the single set version of the colored dBG, the colors are assigned to every string. Therefore, the number of columns in C grows with the size of the collection. Alipanahi et al. [1] noticed that we could reduce the space of C by using the same colors in those strings that have no common kmers. This new problem was named the *CDBG-recoloring*, and formally stated

as follows; given a set R of strings and its dBG G, find the minimum number of colors such that (i) every string $R_i \in R$ is assigned one color and (ii) strings having two or more kmers in common in G cannot have the same color. Alipanahi et al. [1] showed that an instance $I(G')$ of the *Graph-Coloring* problem can be reduced in polynomial time to another instance $I'(G)$ of the *CDBG-recoloring* problem. Thus, any algorithm that finds $\chi(G')$, also finds the minimum number of colors for dBG G. However, they also proved that the decision version of *CDBG-Recoloring* is NP-complete.

3 Definitions

Let $R = \{R_1, R_2, ..., R_n\}$ be a collection of n DNA sequencing *reads*, and let $R' = \{R_1, R_1^{rc}..R_n, R_n^{rc}\}$ be a collection of size $2n$ that contains the strings in R along with their DNA reverse complements. The dBG of order k constructed from R' is defined as $G^k_{R'} = (V, E)$, and an instance of $BOSS$ for $G^k_{R'}$ is denoted as $BOSS(G^k_{R'}) = (V', E')$, where V' and E' include the dummy nodes and their edges. For simplicity, we will refer to $BOSS(G^k_{R'})$ just as $BOSS(G)$. A node in V' is considered an *starting* node if its $k-1$ label is of the form $\$A$, where $\$$ is a dummy symbol and A is a $k-2$ prefix of one or more sequences in R'. Equivalently, a node is considered an *ending* node if its $k-1$ label is of the form $A\$$, with $\$$ being a dummy and A being a $k-2$ suffix of one or more sequences in R'. Nodes whose labels do not contain dummy symbols are considered *solid*, and solid nodes with at least one predecessor node with outdegree more than one are considered *critical*. For practical reasons, we define two extra functions, `isStarting` and `isEnding` that are used to check if a node is starting or ending respectively.

A *walk* P over the dBG of $BOSS(G)$ is a sequence $(v_0, e_0, v_1...v_{t-1}, e_t, v_t)$ where $v_0, v_1, ...v_{t-1}, v_t$ are nodes and $e_1..e_t$ are edges, e_i connecting v_{i-1} with v_i. P is a *path* if all the nodes are different, except possibly the first and the last. In such case, P is said to be a *cycle*. A sequence $R_i \in R$ is *unambiguous* if there is a path in $BOSS(G)$ whose label matches the sequence of R_i and if no pair of colored nodes in $(u, v) \in P$ share a predecessor node $v' \in P$. In any other case, R_i is *ambiguous*. Finally, the path P_i that spells the sequence of R_i is said to be *safe* if every one of its branching nodes has only one successor colored with the color of R_i.

We assume that R is a *factor-free* set, i.e., no $R_i \in R$ is also a substring of another sequence R_j, with $i \neq j$.

4 Coloring a dBG of Reads

In this section, we define a coloring scheme for $BOSS(G)$ that generates a more succinct color matrix, and that allows us to reconstruct and assemble unambiguous sequences of R'. We use the dBG of R' because most of the Bioinformatic analyses require the inspection of the reverse complements of the reads. Unlike previous works, the rows in C represent the nodes in $BOSS(G)$ instead of the edges.

A Partial Coloring. We make C more sparse by coloring only those nodes in the graph that are *strictly* necessary for reconstructing the sequences. We formalize this idea with the following lemma:

Lemma 1. *For the path of an unambiguous sequence $R_i \in R'$ to be safe we have to color the starting node s_i that encodes the $k-2$ prefix of R_i, the ending node e_i that encodes the $k-2$ suffix of R_i and the critical nodes in the path.*

Proof. We start a walk from s_i using the following rules: (i) if the current node v in the walk has outdegree one, then we follow its only outgoing edge, (ii) if v is a branching node, i.e., it has outdegree more than one, then we inspect its successor nodes and follow the one colored with the same color of s_i and (iii) if v is equal to e_i, then we stop the traversal. □

Note that the successor nodes of a branching node are critical by definition, so they are always colored. On the other hand, nodes with outdegree one do not require a color inspection because they have only one possible way out.

Coloring the nodes s_i and e_i for every R_i is necessary; otherwise, it would be difficult to know when a path starts or ends. Consider, for example, using the solid nodes that represent the $k-1$ prefix and the $k-1$ suffix of R_i as starting and ending points respectively. It might happen that the starting point of R_i can also be a critical point of another sequence R_j. If we start a reconstruction from s_i and pick the color of R_j, then we will generate an incomplete sequence. A similar argument can be used for ending nodes. The concepts associated with our coloring idea are depicted in Figs. 2A and B.

Unsafe Coloring. As explained in Sect. 2, we can use the recoloring idea of [1] to reduce the number of columns in C. Still, using the same colors for unrelated strings is not safe for reconstructing unambiguous sequences.

Lemma 2. *Using the same color c for two unambiguous sequences $R_i, R_j \in R'$ that do not share any $k-1$ substring might produce an unsafe path for R_i or R_j.*

Proof. Assume there is another pair of sequences $R_x, R_y \in R'$ that do not share any $k-1$ subsequence either, to which we assign them color c'. Suppose that the paths of R_x and R_j crosses the paths of R_i and R_j such that the resulting dBG topology resembles a grid. In other words, if R_i has the edge (u, u') and R_j has the edge (v, v'), then R_x has the edge (u, v) and R_y has the edge (u', v'). In this scenario, v will have two successors, node v' from the path of R_j and some other node v'' from the path of R_x. Both v' and v'' are critical by definition so they will be colored with c and c' respectively. The problem is that node v' is also a critical node for R_y, so it will also have color c'. The reason is that u', a node that precedes v', appears in R_i and R_y. As a consequence, the path of R_x is no longer safe because one of its nodes (v in this example) has to successors colored with c'. A similar argument can be made for R_i and color c. Figure 1 depicts the idea of this proof. □

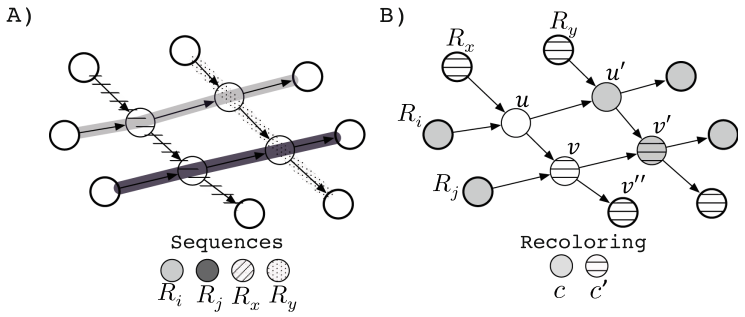
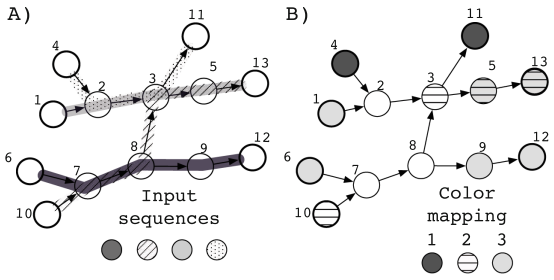


Fig. 1. Example of unsafe paths produced by a graph recoloring. (A) The dBG generated from the unambiguous sequences R_i, R_j, R_x and R_y. Every texture represents the path of a specific string. (B) Recolored dBG. Sequences R_i and R_j are assigned the same color c (light gray) as they do not share any $k-1$ substring. Similarly, sequences R_x and R_y are assigned another color c' (horizontal lines) as they do not share any $k-1$ sequence neither. Nodes u, u', v, v' and v'' are those mentioned in the Proof of Lemma 2. The sequences of R_i and R_x cannot be reconstructed as their paths become unsafe after the recoloring.

When spurious edges connect paths of unrelated sequences that are assigned the same color (as in the proof of Lemma 2), we can generate chimeric strings if, by error, we follow one of those edges. In the algorithm, we solve this problem by assigning different colors to those strings with sporadic edges, even if they do not share any $k-1$ substring.

Safer and Greedy Coloring. Our greedy coloring algorithm starts by marking in a bitmap $N = [1..|V'|]$ the p nodes of $BOSS(G)$ that need to be colored (starting, ending and critical). After that, we create an array M of p entries. Every $M[j]$ with $j \in [1..p]$ will contain a dynamic vector that stores the colors of the *j-th* colored node in the $BOSS$ ordering. We also add $\texttt{rank}_1$ support to N to map a node $v \in V$ to its array of colors in M. Thus, its position can be inferred as $\texttt{rank}_1(N, v)$.

The only inputs we need for the algorithm are N, R' and $BOSS(G)$. For every $R_i \in R'$ we proceed as follows; we append a dummy symbol at the ends of the string, and then use the function `label2Node` to find the node v labeled with the $k-1$ prefix of R_i. Note that this prefix will map a starting node as we append dummies to R_i. From v, we begin a walk on the graph and follow the edges whose symbols match the characters in the suffix $R_i[k..|R_i|]$. Note now that the last node v' we visit in this walk is an ending node that maps the $k-1$ suffix of R_i. As we move through the edges, we store in an array W_i the starting, ending, and critical nodes associated with R_i. Additionally, we push into another array I_i the neighbor nodes of the walk that need to be inspected to assign a color to R_i. The rules for pushing elements into I_i are as follows; (i) if v is a node in the path of R_i with outdegree more than one, then we push all its successor



C)

	Color					
ID	**1**	**2**	**3**	N	M'	F
1	0	0	1	1	3	1
2	0	0	0	0	2	1
3	0	1	0	1	1	1
4	1	0	0	1	2	1
5	0	1	1	1	1	0
6	0	0	1	1	3	1
7	0	0	0	0	3	1
8	0	0	0	0	2	1
9	0	0	1	1	1	1
10	0	1	0	1	3	1
11	1	0	0	1	2	1
12	0	0	1	1	1	0
13	0	1	1	1		

Fig. 2. Succinct colored dBG. (A) The topology of the graph. Colors and textures represent the paths that spell the input sequences of the dBG. Numbers over the nodes are their identifiers. Nodes 4,1,6 and 10 are starting nodes (darker borders). Nodes 11,13 and 12 are ending nodes and nodes 3,9,11 and 5 are critical. (B) Our greedy coloring algorithm. (C) The binary matrix C that encodes the colors of Fig. B. The left side is C in its uncompressed form and the right side is our succinct version of C using the arrays N,M', and F.

nodes into I_i, (ii) if v is a node in the path of R_i with indegree more than one, then we visit every predecessor node v' of v, and if v' has outdegree more than one, then we push into I_i the successor nodes of v'. Once we finish the traversal, we create a hash map H_i and fill it with the colors that were previously assigned to the nodes in I_i and W_i. After that, we pick the smallest color c' that is not in the keys of H_i, and push it to every array $M[\texttt{rank}_1(N, j)]$ with $j \in W_i$. After we process all the sequences in R', the final set of colors is represented by the values in M. The whole processing of coloring a R_i is described in detail by the procedure `greedyCol` in Pseudocode (Algorithm 1).

The construction of the sets W_i and I_i is independent for every string in R', so it can be done in parallel. However, the construction of the hash map H_i and the assignment of the color c' to the elements of W_i has to be performed sequentially as all the sequences in R' need concurrent access to M.

Ambiguous Sequences. Our scheme, however, cannot safely retrieve sequences that are ambiguous.

Lemma 3. *Ambiguous sequences of R' cannot be reconstructed safely from the color matrix C and $BOSS(G)$.*

Proof. Assume that collection R is composed just by one string $R_1 = XbXc$, where X is a repeated substring and b, c are two different symbols in Σ. Consider also that the kmer size for $BOSS(G)$ is $k = |X| + 1$. The instance of $BOSS(G)$ will have a node v labeled with X, with two outgoing edges, whose symbols are b and c. Given our coloring scheme, the successor nodes of v will be both colored with the same color. As a consequence, if during a walk we reach node v, then

we will get stuck because there is not enough information to decide which is the correct edge to follow (both successor nodes have the same color). □

A sequence R_i will be ambiguous if it has the same $k-1$ pattern in two different contexts. Another case in which R_i is ambiguous is when a spurious edge connects an uncolored node of R_i with two or more critical nodes in the same path. Note that unlike unambiguous sequences with spurious edges, an ambiguous sequence will always be encoded by an unsafe path, regardless of the recoloring algorithm. In general, the number of ambiguous sequences will depend on the value we use for k.

5 Compressing the Colored dBG

The pair (M, N) can be regarded as a compact representation of C, where the empty rows were discarded. Every $M[i]$, with $i \in [1..|M|]$, is a row with at least one value, and every color $M[i][j]$, with $j \in [1..|M[i]|]$, is a column. However, M is not succinct enough to make it practical. We are still using a computer word for every color of M. Besides, we need $|M|$ extra words to store the pointers for the lists in M.

We compress M by using an idea similar to the one implemented in *BOSS* to store the edges of the dBG. The first step is to sort the colors of every list $M[i]$. Because the greedy coloring generates a set of unique colors for every node, each $M[i]$ becomes an array of strictly increasing elements after the sorting. Thus, instead of storing the values explicitly, we encode them as deltas, i.e., $M[i][j] = M[i][j] - M[i][j-1]$. After transforming M, we concatenate all its values into one single list M' and create a bit map $F = [1..|M'|]$ to mark the first element of every $M[i]$ in M'. We store M' using Elias-Fano encoding [10,11] and F using the compressed representation for bit maps of [26]. Finally, we add $\texttt{select}_1$ support to F to map a range of elements in M' to an array in M. The complete representation of the color matrix now becomes $C = N + F + M'$ (see Fig. 2C). The complete index of the colored dBG is thus composed of our version of C and $BOSS(G)$. We now formalize the idea of retrieving the colors of a node from the succinct representation of C.

– $\texttt{getColors}(v)$: list of colors assigned to node v.

Theorem 1. *the function* $\texttt{getColors}(v)$ *computes in* $\mathcal{O}(c)$ *time the* c *colors assigned to node* v.

Proof. We first compute the rank r of node v within the colored nodes. This operation is carried out with $r = \texttt{rank}_1(N, v)$. After retrieving r, we obtain the range $[i..j]$ in M' where the values of v lie. For this purpose, we perform two $\texttt{select}_1$ operations over F, $[i, j] = (\texttt{select}_1(F, r), \texttt{select}_1(F, r+1) - 1)$. Finally, we scan the range $[i..j]$ in M', and as we read the values, we incrementally reconstruct the colors from the deltas. All the `rank` and `select` operations takes $\mathcal{O}(1)$, and reading the $c = j - i + 1$ entries from M' takes $\mathcal{O}(c)$, because retrieving an element from an Elias-Fano-encoded array also takes $\mathcal{O}(1)$. In conclusion, computing the colors of v takes $\mathcal{O}(c)$. □

6 Algorithms for the Colored dBG

Reconstructing Unambiguous Sequences. We describe now an online algorithm that works on top of our index and that reconstructs all the unambiguous sequences in R'. We cannot tell, however, if a reconstructed string R_i was present in the original set R or if it was its reverse complement R_i^{rc}. This is not really a problem, because a sequence and its reverse complement are equivalent in most of the Bioinformatic analyses.

The algorithm receives as input a starting node v. It first computes an array A with the colors assigned to v using the function `getColors` (see Sect. 5), and then sets a string $S =$ `nodeLabel`(v). For every color $a \in A$, the algorithm performs the following steps; initializes two temporary variables, an integer $v' = v$ and string $S' = S$, and then begins a graph walk from v'. If the outdegree of v' is one, then the next node in the walk is the successor node $v' =$ `outgoing_r`$(v', 1)$. On the other hand, if the outdegree of v' is more than one, then the algorithm inspects all the successor nodes of v' to check which one of them is the node v'' colored with a. If there is only one such v'', then it sets $v' = v''$. This procedure continues until v' becomes an ending node. During the walk, the edge symbols are pushed into S'. When an ending node is reached, the algorithm reports $S'[1..|S'| - 1]$ as the reconstructed sequence.

If at some point during a walk, the algorithm reaches a node with outdegree more than one, and with more than one successor colored with a, then aborts the reconstruction of the string as the path is unsafe for color a. Then, it returns to v and continues with the next sequence. The complete procedure is detailed in the function `buildSeqs` of Algorithm 2.

Assembling Contigs. Our coloring scheme for the dBG allows us to report sequences that represent the overlap of two or more strings of R'. There are several ways in which a set of sequences can be arranged such that they form valid overlaps, but in practice, we are not interested in all such combinations. What we want is to compute only those union strings that describe real segments of the underlying genome of R', a.k.a *contig* sequences. In this work we do not go deep into the complexities of contig assembly (see [14,16,19,20] for some review). Instead, we propose a simple heuristic, that work on top of our index, and that it is aimed to produce contigs that are longer than those produced by uncolored dBGs.

Similar to `buildSeqs`, this method traverses the graph to reconstruct the contigs. During the process, it uses the color information to weight the outgoing edges *on the fly*, and thus, inferring which is the most probable path that matches a real segment of the source DNA.

The algorithm receives as input a starting node v and initializes a set L and hash map Q. Both data structures are used to store information about the strings that belong to the contig of v. A read $R_i \in R'$ is identified in the index as a pair (c, v'), where c is a color assigned to R_i and v' is the starting node of its path. L contains the reads already traversed while Q contains the active reads.

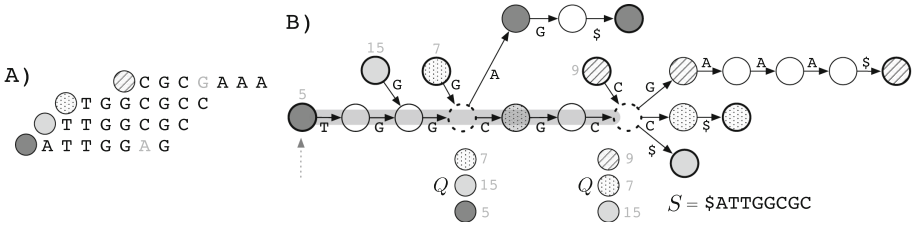


Fig. 3. Example of the assembly of a contig using our index. (A) Inexact overlap of four sequences. The circle to the left of every string represents its color in the dBG. Light gray symbols are mismatches in the overlap. (B) The colored dBG of the sequences. Circles with darker borders are starting and ending nodes. Light gray values over the starting nodes are their identifiers. The contig assembly begins in node 5 (denoted with a dashed arrow) and the threshold x to continue the extension is set to 0.5. The state of the hash map Q when the walk reaches a branching node (dashed circles) is depicted below the graph. The assembly ends in the right-most branching node as it has not a successor node that contains at least 50% of the colors in Q. The final contig is shown as a light grey path over the graph, and its sequence is stored in S.

The algorithm also initializes a string $S = \texttt{nodeLabel}(v)$ and pushes every pair $Q[c_i] = v$ with $c_i \in \texttt{getColor}(v)$. After that, it begins a walk from v and pushes into S the symbols of the edges it visits. For every new node v' reached during the walk, the algorithm checks if one of its predecessor nodes, say u, is a starting node. If so, then for every $c_i \in \texttt{getColors}(u)$ sets $Q[c_i] = u$ if (c_i, u) does not exist in L. On the other hand, if one of the successors of v', say u', is an ending node, then for every $c_i \in \texttt{getColors}(u')$ sets $L[(c, Q[c_i])]$ and then removes the entry $Q[c_i]$. After updating Q and L, it selects one of the outgoing edges of v' to continue the walk. For this purpose, the algorithm uses the following rules; (i) if v' has outdegree one, then it takes its only outgoing edge, (ii) if v' has outdegree more than one, then it inspects how the colors in Q distribute among the successors of v'. If there is only one successor node of v', say v'', colored with at least x fraction of the colors of Q, where x is a parameter, then the algorithm follows v'', and removes from Q the colors of the other successor nodes of v'.

The algorithm will stop if; (i) there is no such v'' that meet the x threshold, (ii) there is more than one successor of v' with the same color or (iii) v' has outdegree one, but the successor node is an ending node. After finishing the walk, the substring $S[2..|S|]$ is reported as the contig. The procedure `contigAssm` in Pseudocode (Algorithm 3) describes in detail the contig assembly algorithm, and a graphical example is shown in Fig. 3.

7 Experiments

We use a set of reads generated from the E.coli genome[1] to test the ideas described in this article. The raw file was in `FASTQ` format and contained

[1] http://spades.bioinf.spbau.ru/spades_test_datasets/ecoli_mc.

14,214,324 reads of 100 characters long each. We preprocessed the file by removing sequencing errors using the tool of [3], and discarding reads with `N` symbols. The preprocessing yielded a data set of 8,655,214 reads (a `FASTQ` file of 2 GB). Additionally, we discarded sequencing qualities and the identifiers of the reads as they are not considered in our data structure. From the resulting set R (a text file of 833.67 MB), we created another set R' that considers the elements in R and their reverse complements.

Our version of the colored dBG, the algorithm for greedy coloring and the algorithms for reconstructing and assembly reads were implemented in `C++`[2], on top of the `SDSL-lite` library [12]. In our implementation, arrays M' and F are precomputed beforehand to store the colors directly to them, because using the dynamic list M is not cache-friendly. Additionally, all our code, except the algorithm for contig assembly, can be executed using multiple threads.

We built six instances of our index using R' as input. We choose different values for k, from 25 to 50 in steps of five. The coloring of every one of these instances was carried out using eight threads. Statistics about the graph topologies are shown in Table 1, and statistics about the coloring process are shown in Table 2. In every instance, we reconstructed the unambiguous reads (see Table 2). Additionally, we generated an FM-index of R' to locate the reconstructed reads and check that they were real sequences. All the tests were carried out on a machine with Debian 4.9, 252 GB of RAM and processor Intel(R) Xeon(R) Silver @ 2.10 GHz, with 32 cores.

8 Results

The average compression rate achieved by our index is 1.89, meaning that, in all the cases, the data structure used about half the space of the plain representation of the reads (see Table 1). We also note that the smaller the value for k, the greater the size of the index. This behavior is expected as the dBG becomes denser when we decrease k. Thus, we have to store a higher number of colors per node.

The number of colors of every instance is several orders of magnitude smaller than the number of reads, being $k = 25$ the instance with more colors (6552) and $k = 50$ the instance with the fewest (1689). Even though the fraction of colored nodes in every instance is small, the percentage of the index space used by the color matrix is still high (73% on average). Regarding the time for coloring the graph, it seems to be reasonable for practical purposes if we use several threads. In fact, building, filling and compacting C took 5,015 s on average, and the value decreases if we increment k. The working space, however, is still considerable. We had memory peaks ranging from 3.03 GB to 4.3 GB, depending on the value for k (see Table 2).

The process of reconstructing the reads yielded a small number of ambiguous sequences in all the instances (2,760 sequences on average), and decreases with higher values of k, especially for values above 40 (see Table 2).

[2] https://bitbucket.org/DiegoDiazDominguez/colored_bos/src/master.

Table 1. Statistics about the different colored dBGs generated in the experiments. The index size is expressed in MB and considers the space of $BOSS(G)$ plus the space of our succinct version of C. The compression rate was calculated as the space of the plain representation of the reads (833.67 MB) divided by the index size.

k	Total number of nodes	Number of solid nodes	Number of edges	Index size	Compression rate
25	106,028,714	11,257,781	120,610,151	446.38	1.86
30	142,591,410	11,425,646	157,186,548	443.82	1.87
35	179,167,289	11,561,630	193,773,251	441.18	1.88
40	215,751,326	11,667,364	230,365,635	438.23	1.90
45	252,337,929	11,743,320	266,958,709	435.30	1.91
50	288,925,674	11,791,640	303,552,318	432.13	1.92

Table 2. Statistics about our greedy coloring algorithm. The column "Color space usage" refers to the percentage of the index space used by our succinct version of C. Elapsed time and memory peak are expressed in seconds and MB, respectively, and both consider only the process of building, filling, and compacting the color matrix.

k	Number of colored nodes	Number of colors	Color space usage	Ambiguous sequences	Elapsed time	Memory peak
25	21,882,874	6,552	83.03	1904	5,835	4,391
30	21,907,324	4,944	79.14	1502	5,551	4,119
35	21,926,687	2,924	75.27	1224	5,131	3,847
40	21,942,083	2,064	71.40	1054	4,872	3,575
45	21,954,138	1,888	67.51	714	4,507	3,303
50	21,964,947	1,689	63.58	176	4,199	3,030

9 Conclusions and Further Work

Experimental results shows our data structure is succinct, and that has a practical use. Still, we believe that a more careful algorithm for constructing the index is still necessary to reduce the memory peaks during the coloring. Further compaction of the color matrix can be achieved by using more elaborated compression techniques. However, this extra compression can increase the construction time of the colored dBG and produce a considerable slow down in the algorithms that work on top of it for extracting information from the reads. Comparison of our results with other similar data structures is difficult for the moment. Most of the indexes based on colored dBGs were not designed to handle huge sets of colors like ours and the greedy recoloring of [1] does not scale well

and needs extra information for reconstructing the reads. Still, it is a promising approach that, with further work, can be used in the future as a base for performing Bioinformatics analyses in compressed space.

A Appendix

A.1 Pseudocodes

Algorithm 1. Function `greedyCol`

```
1: procedure greedyCol(G,N,R_i,M)   ▷ G is a dBG, N is a bitmap, R_i is a string and M is array of lists
2:     R_i ← $R_i$                  ▷ append dummy symbols at the ends of R_i
3:     v ← string2node(R_i[1..k − 1])
4:     W_i ← ∅
5:     I_i ← I_i ∪ rank_1(N, v)
6:     for each r ∈ R_i[k − 1..|R_i|] do        ▷ traverse the dBG path of R_i
7:         o ← outdegree(G, v)
8:         if o > 1 then
9:             for j ← 1 to o do
10:                I_i ← I_i ∪ rank_1(N, forward_r(G, v, j))
11:        i ← indegree(G, v)
12:        if i > 1 then
13:            for j ← 1 to i do
14:                v' ← incomming_r(G, v, j)
15:                o' ← outdegree(G, v'))
16:                if o' > 1 then
17:                    for j ← 1 to o' do
18:                        I_i ← I_i ∪ rank_1(N, forward_r(G, v', j))
19:        if N[v] is true then
20:            W_i ← W_i ∪ rank_1(N, v)
21:        v ← forward(G, v, r)
22:    W_i ← W_i ∪ rank_1(N, v)
23:    I_i ← I_i ∪ rank_1(N, v)
24:    for each n ∈ I_i do              ▷ compute the colors already used
25:        for each c ∈ M[n] do
26:            H_i[c] ← true
27:    c' ← minimum color not in H_i
28:    for each n ∈ W_i do                          ▷ color the nodes
29:        M[n] ← M[n] ∪ c'
```

Algorithm 2. Function `buildSeqs`

```
1: procedure buildSeqs(G,v) ▷ G is a colored dBG and v is a starting node
2:     L ← ∅                                              ▷ list of rebuilt sequences
3:     A ← getColors(G,v)
4:     S ← nodeLabel(G, v)                ▷ initialize an string with the label of v
5:     for each a ∈ C do
6:         v′ ← v                                         ▷ temporal dBG node
7:         S′ ← S, amb ← false
8:         while isEnding(G,v′) is false and amb is false do
9:             o ← outdegree(G,v′)
10:            if o is 1 then
11:                S′ ← S′ ∪ edgeSymbol(G, v′, 1) ▷ push the new symbol into S′
12:                v′ ← forward_r(G, 1)
13:            else
14:                m ← 0
15:                for u ← 1 to o do  ▷ check which successors of v′ has color a
16:                    if a ∈ getColors(forward_r(G, v′, u)) then
17:                        v′ ← forward_r(G, v′, u)
18:                        m ← m + 1
19:                if m > 1 then        ▷ more than one successor v′ has color a
20:                    amb ← true
21:        if amb not true then
22:            L ← L ∪ S[2..|S| − 1]
23:    return L
```

Algorithm 3. Function `contigAssm`

```
1: procedure contigAssm(G,v,x)  ▷ v is a starting node and x is a threshold
2:     L ← ∅
3:     S ← nodeLabel(G, v)
4:     for each c_i ∈ getColors(v) do
5:         Q[c_i] ← v
6:     while true do
7:         if indegree(G, v) > 1 then
8:             v' ← backward_r(G, v, 1))
9:             if isStarting(v') then                ▷ add new reads to the contig
10:                for each c_i ∈ getColors(v') do
11:                    if L[(c_i, v')] is not true then
12:                        Q[c_i] ← v'
13:        if o ← outdegree(G, v) > 1 then
14:            t ← v, v ← 0
15:            for i ← 1 to o do    ▷ compute the most probable successor node
16:                v' ← forward_r(G, t, i)
17:                if isEnding(v') then              ▷ discard reads ending at v
18:                    for each c_i ∈ getColors(v') do
19:                        L[(c_i, Q[c_i])] ← true
20:                    Q ← Q \ A
21:                else
22:                    A ← getColors(v')
23:                    w ← (Q ∩ A)/|Q|               ▷ weight the successor node
24:                    if w ≥ x then
25:                        v ← v'
26:                        Q ← A
27:                        S ← S ∪ edgeSymbol(G, t, i)
28:                        break
29:            if v is 0 then break   ▷ no successor has the minimum weight x
30:        else
31:            v ← forward_r(G, v, 1)
32:            if isEnding(v) then break
33:            S ← S ∪ edgeSymbol(G, v, 1)
34:    return S[2..|S|]
```

References

1. Alipanahi, B., Kuhnle, A., Boucher, C.: Recoloring the colored de Bruijn graph. In: Proceedings of 25th International Symposium on String Processing and Information Retrieval (SPIRE), pp. 1–11 (2018). https://doi.org/10.1007/978-3-030-00479-8_1

2. Almodaresi, F., Pandey, P., Patro, R.: Rainbowfish: a succinct colored de Bruijn graph representation. In: Proceedings of 17th International Workshop on Algorithms in Bioinformatics (WABI). Article 18 (2017). https://doi.org/10.4230/LIPIcs.WABI.2017.18
3. Bankevich, A., et al.: SPAdes: a new genome assembly algorithm and its applications to single-cell sequencing. J. Comput. Biol. **19**(5), 455–477 (2012). https://doi.org/10.1089/cmb.2012.0021
4. Boucher, C., Bowe, A., Gagie, T., Puglisi, S.J., Sadakane, K.: Variable-order de Bruijn graphs. In: Proceedings of 25th Data Compression Conference (DCC), pp. 383–392 (2015). https://doi.org/10.1109/DCC.2015.70
5. Bowe, A., Onodera, T., Sadakane, K., Shibuya, T.: Succinct de Bruijn graphs. In: Proceedings of 12th International Workshop on Algorithms in Bioinformatics (WABI), pp. 225–235 (2012). https://doi.org/10.1007/978-3-642-33122-0_18
6. Bray, N., Pimentel, H., Melsted, P., Pachter, L.: Near-optimal probabilistic RNA-seq quantification. Nat. Biotechnol. **34**(5), 525–527 (2016). https://doi.org/10.1038/nbt.3519
7. de Bruijn, N.G.: A combinatorial problem. Koninklijke Nederlandse Akademie v. Wetenschappen **49**(49), 758–764 (1946)
8. Burrows, M., Wheeler, D.: A block sorting lossless data compression algorithm. Technical report 124, Digital Equipment Corporation (1994)
9. Clark, D.: Compact PAT trees. Ph.D. thesis, University of Waterloo, Canada (1996)
10. Elias, P.: Efficient storage and retrieval by content and address of static files. J. ACM **21**(2), 246–260 (1974). https://doi.org/10.1145/321812.321820
11. Fano, R.M.: On the number of bits required to implement an associative memory. Massachusetts Institute of Technology (1971)
12. Gog, S., Beller, T., Moffat, A., Petri, M.: From theory to practice: plug and play with succinct data structures. In: Proceedings of 13th International Symposium on Experimental Algorithms (SEA), pp. 326–337 (2014). https://doi.org/10.1007/978-3-319-07959-2_28
13. Holley, G., Wittler, R., Stoye, J.: Bloom filter trie - a data structure for pan-genome storage. In: Proceedings of 15th International Workshop on Algorithms in Bioinformatics (WABI), pp. 217–230 (2015). https://doi.org/10.1007/978-3-662-48221-6_16
14. Idury, R.M., Waterman, M.S.: A new algorithm for DNA sequence assembly. J. Comput. Biol. **2**(2), 291–306 (1995). https://doi.org/10.1089/cmb.1995.2.291
15. Iqbal, Z., Caccamo, M., Turner, I., Flicek, P., McVean, G.: De novo assembly and genotyping of variants using colored de Bruijn graphs. Nat. Genet. **44**(2), 226–232 (2012). https://doi.org/10.1038/ng.1028
16. Kececioglu, J.D., Myers, E.W.: Combinatorial algorithms for DNA sequence assembly. Algorithmica **13**(1), 7–51 (1995). https://doi.org/10.1007/BF01188580
17. Lewis, R.: A Guide to Graph Colouring. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-25730-3
18. Mäkinen, V., Navarro, G.: Succinct suffix arrays based on run-length encoding. Nordic J. Comput. **12**(1), 40–66 (2005). https://doi.org/10.1007/11496656_5
19. Medvedev, Paul, Georgiou, Konstantinos, Myers, Gene, Brudno, Michael: Computability of Models for Sequence Assembly. In: Giancarlo, Raffaele, Hannenhalli, Sridhar (eds.) WABI 2007. LNCS, vol. 4645, pp. 289–301. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-74126-8_27

20. Medvedev, P., Pham, S., Chaisson, M., Tesler, G., Pevzner, P.: Paired de bruijn graphs: a novel approach for incorporating mate pair information into genome assemblers. J. Comput. Biol. **18**(11), 1625–1634 (2011). https://doi.org/10.1089/cmb.2011.0151
21. Mustafa, H., Kahles, A., Karasikov, M., Raetsch, G.: Metannot: a succinct data structure for compression of colors in dynamic de Bruijn graphs. bioRxiv, Article 236711 (2017). https://doi.org/10.3929/ethz-b-000236153
22. Mustafa, H., Schilken, I., Karasikov, M., Eickhoff, C., Rätsch, G., Kahles, A.: Dynamic compression schemes for graph coloring. Bioinformatics **35**(3), 407–414 (2018). https://doi.org/10.1093/bioinformatics/bty632
23. Navarro, G.: Compact Data Structures: A Practical Approach. Cambridge University Press, Cambridge (2016). https://doi.org/10.1017/CBO9781316588284
24. Okanohara, D., Sadakane, K.: Practical entropy-compressed rank/select dictionary. In: Proceedings of 9th Workshop on Algorithm Engineering and Experiments (ALENEX), pp. 60–70 (2007). https://doi.org/10.1137/1.9781611972870.6
25. Pandey, P., Almodaresi, F., Bender, M.A., Ferdman, M., Johnson, R., Patro, R.: Mantis: a fast, small, and exact large-scale sequence-search index. Cell Syst. **7**(2), 201–207 (2018). https://doi.org/10.1016/j.cels.2018.05.021
26. Raman, R., Raman, V., Satti, S.R.: Succinct indexable dictionaries with applications to encoding k-ary trees, prefix sums and multisets. ACM Trans. Algorithms **3**(4), Article 43 (2007). https://doi.org/10.1145/1290672.1290680
27. Reuter, J., Spacek, D., Snyder, M.: High-throughput sequencing technologies. Mol. Cell **58**(4), 586–597 (2015). https://doi.org/10.1016/j.molcel.2015.05.004
28. Salmela, L., Walve, R., Rivals, E., Ukkonen, E.: Accurate self-correction of errors in long reads using de Bruijn graphs. Bioinformatics **33**(6), 799–806 (2016). https://doi.org/10.1093/bioinformatics/btw321