# Space-Efficient Merging of Succinct de Bruijn Graphs

Lavinia Egidi[1], Felipe A. Louza[2(✉)], and Giovanni Manzini[1,3]

[1] University of Eastern Piedmont, Alessandria, Italy
{lavinia.egidi,giovanni.manzini}@uniupo.it
[2] Faculty of Electrical Engineering, Universidade Federal de Uberlândia,
Uberlândia, Brazil
louza@ufu.br
[3] IIT CNR, Pisa, Italy

Abstract. We propose a new algorithm for merging succinct representations of *de Bruijn* graphs introduced in [Bowe *et al.* WABI 2012]. Our algorithm is based on the lightweight BWT merging approach by Holt and McMillan [Bionformatics 2014, ACM-BCB 2014]. Our algorithm has the same asymptotic cost of the state of the art tool for the same problem presented by Muggli *et al.* [bioRxiv 2017, Bioinformatics 2019], but it uses less than half of its working space. A novel important feature of our algorithm, not found in any of the existing tools, is that it can compute the *Variable Order* succinct representation of the union graph within the same asymptotic time/space bounds.

Keywords: de Bruijn graphs · Succinct data structures · Merging · Variable-order · Colored graphs · External memory algorithms

1 Introduction

The *de Bruijn* graph for a collection of strings is a key data structure in genome assembly [19]. After the seminal work of Bowe *et al.* [5], many succinct representations of this data structure have been proposed in the literature [2–4,18] offering more and more functionalities still using a fraction of the space required to store the input collection uncompressed. In this paper we consider the problem of merging two existing succinct representations of de Bruijn graphs built for different collections. Since the de Bruijn graph is a lossy representation and from it we cannot recover the original input collection, the alternative to merging is storing a copy of each collection to be used for building new de Bruijn graphs from scratch.

Recently, Muggli *et al.* [16,17] have proposed a merging algorithm for colored de Bruijn graphs and have shown the effectiveness of the merging approach for the construction of de Bruijn graphs for very large datasets. The algorithm in [16] is based on an MSD Radix Sort procedure of the graph edges and its running

time is $\mathcal{O}(mk)$, where m is the total number of edges and k is the order of the de Bruijn graph.

A fundamental parameter of any construction algorithm for succinct data structures is its *space usage* since this parameter determines the size of the largest dataset that can be handled by a machine with a given amount of memory. For a graph with m edges and n nodes the merging algorithm by Muggli *et al.* uses, in addition to the input and the output, $2(m\log\sigma + m + n)$ bits plus $\mathcal{O}(\sigma)$ words of working space, where σ is the alphabet size. This value represents a three fold improvement over previous results, but it is still larger than the size of the resulting de Bruijn graph which is upper bounded by $2(m\log\sigma + m) + o(m)$ bits.

In this paper, we present a new merging algorithm that still runs in $\mathcal{O}(mk)$ time, but only uses $4n$ bits plus $\mathcal{O}(\sigma)$ words of working space. For genome collections ($\sigma = 5$) our algorithm uses less than half the space of Muggli *et al.*'s: our advantage grows with the size of the alphabet and with the average outdegree m/n. Notice that the working space of our algorithm is always less than the space of the resulting de Bruijn graph. In Sect. 4 we will discuss the practical significance of this space reduction.

Our new merging algorithm is based on a mixed LSD/MSD Radix Sort algorithm which is inspired by the lightweight BWT merging algorithm introduced by Holt and McMillan [11,12] and later improved in [8,9]. In addition to its small working space, our algorithm has the remarkable feature that it can compute as a by-product, with no additional cost, the LCS (Longest Common Suffix) between the node labels, thus making it possible to construct succinct Variable Order de Bruijn graph representations [4], a feature not shared by any other merging algorithm.

The rest of the paper is organized as follows. After reviewing succinct de Bruijn graphs in Sect. 2, we describe our algorithm in Sect. 3. In Sect. 4 we describe the implementation details and compare our result to the state of the art. In Sect. 5 we discuss the case of colored or variable order de Bruijn graphs. In Sect. 6 we show that combining an external memory version of our merging algorithm with recent results on external memory de Bruijn graph construction [6,7] we get a space efficient external memory procedure for building succinct representations of de Bruijn graphs for very large collections.

2   Notation and Background

Given the alphabet $\Sigma = \{1, 2, \ldots, \sigma\}$ and a collection of strings $\mathcal{C} = s_1, \ldots, s_d$ over Σ, we prepend to each string s_i k copies of a symbol $\$ \notin \Sigma$ which is lexicographically smaller than any other symbol. The order-k *de Bruijn graph* $G(V, E)$ for the collection $\mathcal{C}$ is a directed edge-labeled graph containing a node v for every **unique k-mer** appearing in one of the strings of $\mathcal{C}$. For each node v we denote by $\overrightarrow{v} = v[1, k]$ its associated k-mer, where $v[1] \ldots v[k]$ are symbols. The graph G contains an edge (u, v), with label $v[k]$, iff one of the strings in $\mathcal{C}$ contains a **$(k + 1)$-mer** with prefix $\overrightarrow{u}$ and suffix $\overrightarrow{v}$. The edge (u, v) therefore represents the $(k+1)$-mer $u[1, k]v[k]$. Note that each node has at most $|\Sigma|$ outgoing edges and all edges incoming to node v have label $v[k]$.
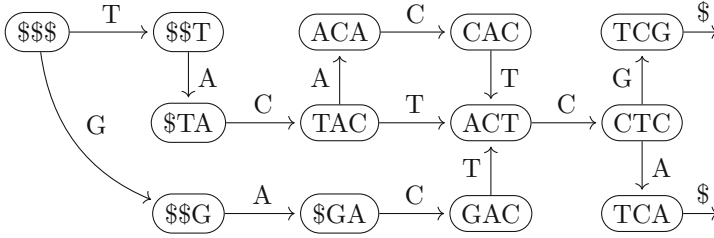
Fig. 1. de Bruijn graph for $\mathcal{C} = \{\text{TACACT, TACTCG, GACTCA}\}$.

BOSS Succinct Representation. In 2012, Bowe *et al.* [5] introduced a succinct representation for the de Bruijn graph, usually referred to as BOSS representation, for the authors' initials. The authors showed how to represent the graph in small space supporting fast navigation operations. The BOSS representation of the graph $G(V, E)$ is defined by considering the set of nodes $v_1, v_2, \ldots v_n$ sorted according to the colexicographic order of their associated k-mer. Hence, if $\overleftarrow{v} = v[k] \ldots v[1]$ denotes the string $\overrightarrow{v}$ reversed, the nodes are ordered so that

$$\overleftarrow{v_1} \prec \overleftarrow{v_2} \prec \cdots \prec \overleftarrow{v_n} \tag{1}$$

By construction the first node is $\overleftarrow{v_1} = \$^k$ and all $\overleftarrow{v_i}$ are distinct. For each node v_i, $i = 1, \ldots, n$, we define W_i as the sorted sequence of symbols on the edges leaving from node v_i; if v_i has out-degree zero we set $W_i = \$$. Let $\mathsf{Node}[i]$ denote the node label for W_i. Finally, we define

1. $W[1, m]$ as the concatenation $W_1 W_2 \cdots W_n$;
2. $W^-[1, m]$ as the bitvector such that $W^-[i] = \mathbf{1}$ iff $W[i]$ corresponds to the label of the edge (u, v) such that $\overleftarrow{u}$ has the smallest rank among the nodes that have an edge going to node v;
3. $\mathsf{last}[1, m]$ as the bitvector such that $\mathsf{last}[i] = 1$ iff $i = m$ or the outgoing edges corresponding to $W[i]$ and $W[i + 1]$ have different source nodes.
4. $\mathsf{C}[1, \sigma]$ as the integer array, such that $\mathsf{C}[c]$ stores the number of symbols smaller than $c \in \Sigma \cup \{\$\}$ in the last symbol of Node.

The length m of the arrays W, W^-, and last is equal to the number of edges plus the number of nodes with out-degree 0. In addition, the number of $\mathbf{1}$'s in last is equal to the number of nodes n, and the number of $\mathbf{1}$'s in W^- is equal to the number of nodes with positive in-degree, which is $n - 1$ since $v_1 = \$^k$ is the only node with in-degree 0. Array C can be obtained by scanning W, W^- and last, therefore, array $\mathsf{Node}[1, m]$ is not stored explicitly.

Note that there is a natural one-to-one correspondence, called LF for historical reasons, between the indices i such that $W^-[i] = \mathbf{1}$ and the the set $\{2, \ldots, n\}$: in this correspondence $LF(i) = j$ iff v_j is the destination node of the edge associated to $W[i]$. See example in Figs. 1 and 2.
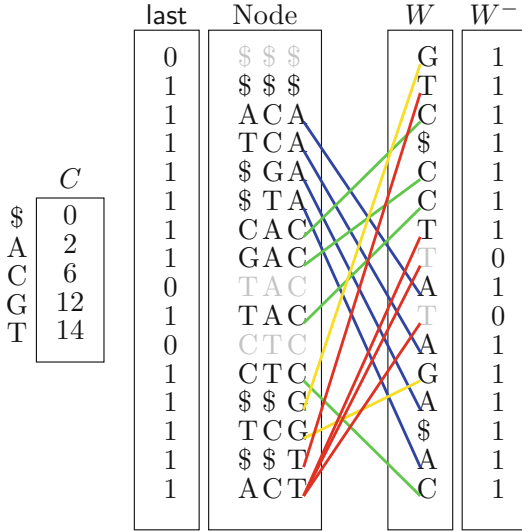
last	Node				W	W^-
0	$ $ $				G	1
1	$ $ $				T	1
1	A C A				C	1
1	T C A				$	1
1	$ G A				C	1
1	$ T A				C	1
1	C A C				T	1
1	G A C				T	0
0	T A C				A	1
1	T A C				T	0
0	C T C				A	1
1	C T C				G	1
1	$ $ G				A	1
1	T C G				$	1
1	$ $ T				A	1
1	A C T				C	1

C	
$	0
A	2
C	6
G	12
T	14

Fig. 2. BOSS representation of the graph in Fig. 1. The colored lines connect each label in W to its destination node; edges of the same color have the same label. Note that edges of the same color do not cross because of Property 1. (Color figure online)

Property 1. The LF map is order preserving in the following sense: if $W^-[i] = W^-[j] = \mathbf{1}$ then

$$W[i] < W[j] \implies LF(i) < LF(j),$$
$$(W[i] = W[j]) \wedge (i < j) \implies LF(i) < LF(j). \tag{2}$$

□

In [5] it is shown that given array C, enriching the arrays W, W^-, and last with the data structures from [10,20] supporting constant time rank and select operations, we can efficiently navigate the graph G. The cost to store array C is $\mathcal{O}(\sigma \log n)$ bits. The overall cost of encoding the three arrays and the auxiliary data structures is bounded by $m \log \sigma + 2m + o(m)$ bits, with the usual time/space tradeoffs available for rank/select data structures.

Colored BOSS. The colored de Bruijn graph [13] is an extension of the de Bruijn graphs for a multiset of individual graphs, where each edge is associated with a set of "colors" that indicates which graphs contain that edge.

The BOSS representation for a set of graphs $\mathcal{G} = \{G_1, \ldots, G_t\}$ contains the union of all individual graphs. In its simplest representation, the colors of all edges $W[i]$ are stored in a two-dimensional binary array $\mathcal{M}$, such that $\mathcal{M}[i, j] = 1$ iff the i-th edge is present in graph G_j. There are different compression alternatives for the color matrix $\mathcal{M}$ that support fast operations [2,15,18]. Recently, Alipanah *et al.* [1] presented a different approach to reduce the size of $\mathcal{M}$ by recoloring.

Variable-Order BOSS. The order k (dimension) of a de Bruijn graph is an important parameter for genome assembling algorithms. The graph can be very small and uninformative when k is small, whereas it can become too large or disconnected when k is large. To add flexibility to the BOSS representation, Boucher *et al.* [4] suggest to enrich the BOSS representation of an order-k de Bruijn graph with the length of the longest common suffix (LCS) between the k-mers of consecutive nodes $v_1, v_2, \ldots, v_n$ sorted according to (1). These lengths are stored in a wavelet tree using $O(n \log k)$ additional bits. The authors show that this enriched representation supports navigation on all de Bruijn graphs of order $k' \leq k$ and that it is even possible to vary the order k' of the graph on the fly during the navigation up to the maximum value k.

The LCS between $\overrightarrow{v_i}$ and $\overrightarrow{v_{i+1}}$ is equivalent to the length of the longest common prefix (LCP) between their reverses $\overleftarrow{v_i}$ and $\overleftarrow{v_{i+1}}$. The LCP (or LCS) between the nodes $v_1, v_2, \cdots, v_n$ can be computed during the k-mer sorting phase. In the following we denote by VO-BOSS the **variable order** succinct de Bruijn graph consisting of the BOSS representations enriched with the LCS/LCP information.

3   Merging Plain BOSS Representations

Suppose we are given the BOSS representations of two de Bruijn graphs $\langle W_0, W_0^-, \mathsf{last}_0 \rangle$ and $\langle W_1, W_1^-, \mathsf{last}_1 \rangle$ obtained respectively from the collections of strings $\mathcal{C}_0$ and $\mathcal{C}_1$. In this section we show how to compute the BOSS representation for the union collection $\mathcal{C}_{01} = \mathcal{C}_0 \cup \mathcal{C}_1$. The procedure does not change in the general case when we are merging an arbitrary number of graphs. Let G_0 and G_1 denote respectively the (uncompressed) de Bruijn graphs for $\mathcal{C}_0$ and $\mathcal{C}_1$, and let

$$v_1, \ldots, v_{n_0} \qquad \text{and} \qquad w_1, \ldots, w_{n_1}$$

denote their respective set of nodes sorted in colexicographic order. Hence, with the notation of the previous section we have

$$\overleftarrow{v_1} \prec \cdots \prec \overleftarrow{v_{n_0}} \qquad \text{and} \qquad \overleftarrow{w_1} \prec \cdots \prec \overleftarrow{w_{n_1}} \tag{3}$$

We observe that the k-mers in the collection $\mathcal{C}_{01}$ are simply the union of the k-mers in $\mathcal{C}_0$ and $\mathcal{C}_1$. To build the de Bruijn graph for $\mathcal{C}_{01}$ we need therefore to: (1) merge the nodes in G_0 and G_1 according to the colexicographic order of their associated k-mers, (2) recognize when two nodes in G_0 and G_1 refer to the same k-mer, and (3) properly merge and update the bitvectors W_0^-, last_0 and W_1^-, last_1.

3.1   Phase 1: Merging k-mers

The main technical difficulty is that in the BOSS representation the k-mers associated to each node $\overrightarrow{v} = v[1, k]$ are not directly available. Our algorithm

will reconstruct them using the symbols associated to the graph edges; to this end
the algorithm will consider only the edges such that the corresponding entries in
W_0^- or W_1^- are equal to $\mathbf{1}$. Following these edges, first we recover the last symbol
of each k-mer, following them a second time we recover the last two symbols of
each k-mer and so on. However, to save space we do not explicitly maintain the
k-mers; instead, using the ideas from [11, 12] our algorithm computes a bitvector
$Z^{(k)}$ representing how the k-mers in G_0 and G_1 should be merged according to
the colexicographic order.

To this end, our algorithm executes $k-1$ iterations of the code shown in Fig. 3
(note that lines 8–10 and 17–22 of the algorithm are related to the computation
of the B array that is used in the following section). For $h = 2, 3, \ldots, k$, during
iteration h, we compute the bitvector $Z^{(h)}[1, n_0 + n_1]$ containing n_0 $\mathbf{0}$'s and n_1
$\mathbf{1}$'s such that $Z^{(h)}$ satisfies the following property

Property 2. For $i = 1, \ldots, n_0$ and $j = 1, \ldots n_1$ the i-th $\mathbf{0}$ precedes the j-th $\mathbf{1}$ in
$Z^{(h)}$ if and only if $\overleftarrow{v_i}[1, h] \preceq \overleftarrow{w_j}[1, h]$. $\qquad\square$

Property 2 states that if we merge the nodes from G_0 and G_1 according
to the bitvector $Z^{(h)}$ the corresponding k-mers will be sorted according to the
lexicographic order restricted to the first h symbols of each reversed k-mer. As
a consequence, $Z^{(k)}$ will provide us the colexicographic order of all the nodes in
G_0 and G_1. To prove that Property 2 holds, we first define $Z^{(1)}$ and show that
it satisfies the property, then we prove that for $h = 2, \ldots, k$ the code in Fig. 3
computes $Z^{(h)}$ that still satisfies Property 2.

For $c \in \Sigma$ let $\ell_0(c)$ and $\ell_1(c)$ denote respectively the number of nodes in
G_0 and G_1 whose associated k-mers end with symbol c. These values can be
computed with a single scan of W_0 (resp. W_1) considering only the symbols $W_0[i]$
(resp. $W_1[i]$) such that $W_0^-[i] = \mathbf{1}$ (resp. $W_1^-[i] = \mathbf{1}$). By construction, it is

$$n_0 = 1 + \sum_{c \in \Sigma} \ell_0(c), \qquad n_1 = 1 + \sum_{c \in \Sigma} \ell_1(c)$$

where the two 1's account for the nodes v_1 and w_1 whose associated k-mer is $\$^k$.
We define

$$Z^{(1)} = \underline{\mathbf{01}} \ \underline{\mathbf{0}^{\ell_0(1)}\mathbf{1}^{\ell_1(1)}} \ \underline{\mathbf{0}^{\ell_0(2)}\mathbf{1}^{\ell_1(2)}} \cdots \underline{\mathbf{0}^{\ell_0(\sigma)}\mathbf{1}^{\ell_1(\sigma)}} \ . \tag{4}$$

The first pair $\mathbf{01}$ in $Z^{(1)}$ accounts for v_1 and w_1; for each $c \in \Sigma$ group $\mathbf{0}^{\ell_0(c)}\mathbf{1}^{\ell_1(c)}$
accounts for the nodes ending with symbol c. Note that, apart from the first two
symbols, $Z^{(1)}$ can be logically partitioned into σ subarrays one for each alphabet
symbol. For $c \in \Sigma$ let

$$\mathsf{start}(c) = 3 + \sum_{i < c}(\ell_0(i) + \ell_1(i))$$

then the subarray corresponding to c starts at position $\mathsf{start}(c)$ and has size
$\ell_0(c) + \ell_1(c)$. As a consequence of (3), the i-th $\mathbf{0}$ (resp. j-th $\mathbf{1}$) belongs to the
subarray associated to symbol c iff $\overleftarrow{v_i}[1] = c$ (resp. $\overleftarrow{w_j}[1] = c$).

To see that $Z^{(1)}$ satisfies Property 2, observe that the i-th $\mathbf{0}$ precedes j-th $\mathbf{1}$ iff the i-th $\mathbf{0}$ belongs to a subarray corresponding to a symbol not larger than the symbol corresponding to the subarray containing the j-th $\mathbf{1}$; this implies $\overleftarrow{v_i}[1,1] \preceq \overleftarrow{w_j}[1,1]$.

The bitvectors $Z^{(h)}$ computed by the algorithm in Fig. 3 can be logically divided into the same subarrays we defined for $Z^{(1)}$. In the algorithm we use an array $F[1,\sigma]$ to keep track of the next available position of each subarray. Because of how the array F is initialized and updated, we see that every time we read a symbol c at line 14 the corresponding bit $b = Z^{(h-1)}[k]$, which gives us the graph containing c, is written in the portion of $Z^{(h)}$ corresponding to c (line 16). The only exception are the first two entries of $Z^{(h)}$ which are written at line 6 which corresponds to the nodes v_1 and w_1. We treat these nodes differently since they are the only ones with in-degree zero. For all other nodes, we implicitly use the one-to-one correspondence (2) between entries $W[i]$ with $W^-[i] = \mathbf{1}$ and nodes v_j with positive in-degree.

The following Lemma proves the correctness of the algorithm in Fig. 3.

Lemma 1. *For $h = 2, \ldots, k$, the array $Z^{(h)}$ computed by the algorithm in Fig. 3 satisfies Property 2.*

Proof. To prove the "if" part of Property 2 let $1 \leq f < g \leq n_0 + n_1$ denote two indexes such that $Z^{(h)}[f]$ is the i-th $\mathbf{0}$ and $Z^{(h)}[g]$ is the j-th $\mathbf{1}$ in $Z^{(h)}$ for some $1 \leq i \leq n_0$ and $1 \leq j \leq n_1$. We need to show that $\overleftarrow{v_i}[1,h] \preceq \overleftarrow{w_j}[1,h]$.

Assume first $\overleftarrow{v_i}[1] \neq \overleftarrow{w_j}[1]$. The hypothesis $f < g$ implies $\overleftarrow{v_i}[1] < \overleftarrow{w_j}[1]$, since otherwise during iteration h the j-th $\mathbf{1}$ would have been written in a subarray of $Z^{(h)}$ preceding the one where the i-th $\mathbf{0}$ is written. Hence $\overleftarrow{v_i}[1,h] \preceq \overleftarrow{w_j}[1,h]$ as claimed.

Assume now $\overleftarrow{v_i}[1] = \overleftarrow{w_j}[1] = c$. In this case during iteration h the i-th $\mathbf{0}$ and the j-th $\mathbf{1}$ are both written to the subarray of $Z^{(h)}$ associated to symbol c. Let f', g' denote respectively the value of the main loop variable p in the procedure of Fig. 3 when the entries $Z^{(h)}[f]$ and $Z^{(h)}[g]$ are written. Since each subarray in $Z^{(h)}$ is filled sequentially, the hypothesis $f < g$ implies $f' < g'$. By construction $Z^{(h-1)}[f'] = \mathbf{0}$ and $Z^{(h-1)}[g'] = \mathbf{1}$. Say f' is the i'-th $\mathbf{0}$ in $Z^{(h-1)}$ and g' is the j'-th $\mathbf{1}$ in $Z^{(h-1)}$. By the inductive hypothesis on $Z^{(h-1)}$ it is

$$\overleftarrow{v_{i'}}[1, h-1] \preceq \overleftarrow{w_{j'}}[1, h-1]. \tag{5}$$

By construction there is an edge labeled c from $v_{i'}$ to v_i and from $w_{j'}$ to w_j hence

$$\overrightarrow{v_i}[1,h] = \overrightarrow{v_{i'}}[1,h-1]c, \qquad \overrightarrow{w_j}[1,h] = \overrightarrow{w_{j'}}[1,h-1]c;$$

therefore

$$\overleftarrow{v_i}[1,h] = c\overleftarrow{v_{i'}}[1,h-1], \qquad \overleftarrow{w_j}[1,h] = c\overleftarrow{w_{j'}}[1,h-1];$$

using (5) we conclude that $\overleftarrow{v_i}[1,h] \preceq \overleftarrow{w_j}[1,h]$ as claimed.

For the "only if" part of Property 2, assume $\overleftarrow{v_i}[1,h] \preceq \overleftarrow{w_j}[1,h]$ for some $i \geq 1$ and $j \geq 1$. We need to prove that in $Z^{(h)}$ the i-th **0** precedes the j-th **1**. If $\overleftarrow{v_i}[1] \neq \overleftarrow{w_j}[1]$ the proof is immediate. If $c = \overleftarrow{v_i}[1] = \overleftarrow{w_j}[1]$ then

$$\overleftarrow{v_i}[2,h] \preceq \overleftarrow{w_j}[2,h].$$

Let i' and j' be such that $\overleftarrow{v_{i'}}[1,h-1] = \overleftarrow{v_i}[2,h]$ and $\overleftarrow{w_{j'}}[1,h-1] = \overleftarrow{w_j}[2,h]$. By induction hypothesis, in $Z^{(h-1)}$ the i'-th **0** precedes the j'-th **1**.

During phase h, the i-th **0** in $Z^{(h)}$ is written to position f when processing the i'-th **0** of $Z^{(h-1)}$, and the j-th **1** in $Z^{(h)}$ is written to position g when processing the j'-th **1** of $Z^{(h-1)}$. Since in $Z^{(h-1)}$ the i'-th **0** precedes the j'-th **1** and since f and g both belong to the subarray of $Z^{(h)}$ corresponding to the symbol c, their relative order does not change and the i-th **0** precedes the j-th **1** as claimed.   □

```
1:  for c ← 1 to σ do
2:      F[c] ← start(c)                                        ▷ Init F array
3:      Block_id[c] ← −1                                      ▷ Init Block_id array
4:  end for
5:  i_0 ← i_1 ← 1                                  ▷ Init counters for W_0 and W_1
6:  Z^(h) ← 01                        ▷ First two entries correspond to v_1 and w_1
7:  for p ← 1 to n_0 + n_1 do
8:      if B[p] ≠ 0 and B[p] ≠ h then
9:          id ← p                          ▷ A new block of Z^(h−1) is starting
10:     end if
11:     b ← Z^(h−1)[p]                                ▷ Get bit b from Z^(h−1)
12:     repeat                                  ▷ Current node is from graph G_b
13:         if W_b^−[i_b] = 1 then
14:             c ← W_b[i_b]                      ▷ Get symbol from outgoing edges
15:             q ← F[c]++              ▷ Get destination for b according to symbol c
16:             Z^(h)[q] ← b                            ▷ Copy bit b to Z^(h)
17:             if Block_id[c] ≠ id then
18:                 Block_id[c] ← id               ▷ Update block id for symbol c
19:                 if B[q] = 0 then                   ▷ Check if already marked
20:                     B[q] ← h            ▷ A new block of Z^(h) will start here
21:                 end if
22:             end if
23:         end if
24:     until last_b[i_b++] ≠ 1                          ▷ Exit if c was last edge
25: end for
```

Fig. 3. Main procedure for merging succinct de Bruijn graphs. Lines 8–10 and 17–22 are related to the computation of the B array introduced in Sect. 3.2.

3.2   Phase 2: Recognizing Identical k-mers

Once we have determined, via the bitvector $Z^{(h)}[1, n_0 + n_1]$, the colexicographic order of the k-mers, we need to determine when two k-mers are identical since in this case we have to merge their outgoing and incoming edges. Note that two identical k-mers will be consecutive in the colexicographic order and they will necessarily belong one to G_0 and the other to G_1.

Following Property 2, and a technique introduced in [8], we identify the i-th **0** in $Z^{(h)}$ with $\overleftarrow{v_i}$ and the j-th **1** in $Z^{(h)}$ with $\overleftarrow{w_j}$. Property 2 is equivalent to state that we can logically partition $Z^{(h)}$ into $b(h) + 1$ h-blocks

$$Z^{(h)}[1, \ell_1], \ Z^{(h)}[\ell_1 + 1, \ell_2], \ \ldots, \ Z^{(h)}[\ell_{b(h)} + 1, n_0 + n_1] \tag{6}$$

such that each block corresponds to a set of k-mers which are prefixed by the same length-h substring. Note that during iterations $h = 2, 3, \ldots, k$ the k-mers within an h-block will be rearranged, and sorted according to longer and longer prefixes, but they will stay within the same block.

In the algorithm of Fig. 3, in addition to $Z^{(h)}$, we maintain an integer array $B[1, n_0 + n_1]$, such that at the end of iteration h it is $B[i] \neq 0$ if and only if a block of $Z^{(h)}$ starts at position i. Initially, for $h = 1$, since we have one block per symbol, we set

$$B = \underline{10} \, \underline{10^{\ell_0(1)+\ell_1(1)-1}} \, \underline{10^{\ell_0(2)+\ell_1(2)-1}} \cdots \underline{10^{\ell_0(\sigma)+\ell_1(\sigma)-1}}.$$

During iteration h, new block boundaries are established as follows. At line 9 we identify each existing block with its starting position. Then, at lines 17–22, if the entry $Z^{(h)}[q]$ has the form $c\alpha$, while $Z^{(h)}[q - 1]$ has the form $c\beta$, with α and β belonging to different blocks, then we know that q is the starting position of an h-block. Note that we write h to $B[q]$ only if no other value has been previously written there. This ensures that $B[q]$ is the smallest position in which the strings corresponding to $Z^{(h)}[q - 1]$ and $Z^{(h)}[q]$ differ, or equivalently, $B[q] - 1$ is the LCP between the strings corresponding to $Z^{(h)}[q - 1]$ and $Z^{(h)}[q]$. The above observations are summarized in the following Lemma, which is a generalization to de Bruijn graphs of an analogous result for BWT merging established in Corollary 4 in [8].

Lemma 2. *After iteration k of the merging algorithm for $q = 2, \ldots, n_0 + n_1$ if $B[q] \neq 0$ then $B[q] - 1$ is the LCP between the reverse k-mers corresponding to $Z^{(k)}[q - 1]$ and $Z^{(k)}[q]$, while if $B[q] = 0$ their LCP is equal to k, hence such k-mers are equal.*   □

The above lemma shows that using array B we can establish when two k-mers are equal and consequently the associated graph nodes should be merged.

3.3   Phase 3: Building BOSS Representation for the Union Graph

We now show how to compute the succinct representation of the union graph $G_0 \cup G_1$, consisting of the arrays $\langle W_{01}, W_{01}^-, \mathsf{last}_{01} \rangle$, given the succinct representations of G_0 and G_1 and the arrays $Z^{(k)}$ and B.

The arrays W_{01}, W_{01}^-, last_{01} are initially empty and we fill them in a single sequential pass. For $q = 1, \ldots, n_0 + n_1$ we consider the values $Z^{(k)}[q]$ and $B[q]$. If $B[q] = 0$ then the k-mer associated to $Z^{(k)}[q-1]$, say $\overleftarrow{v_i}$ is identical to the k-mer associated to $Z^{(k)}[q]$, say $\overleftarrow{w_j}$. In this case we recover from W_0 and W_1 the labels of the edges outgoing from v_i and w_j, we compute their union and write them to W_{01} (we assume the edges are in the lexicographic order), writing at the same time the representation of the out-degree of the new node to last_{01}. If instead $B[q] \neq 0$, then the k-mer associated to $Z^{(k)}[q-1]$ is unique and we copy the information of its outgoing edges and out-degree directly to W_{01} and last_{01}.

When we write the symbol $W_{01}[i]$ we simultaneously write the bit $W_{01}^-[i]$ according to the following strategy. If the symbol $c = W_{01}[i]$ is the first occurrence of c after a value $B[q]$, with $0 < B[q] < k$, then we set $W_{01}^-[i] = \mathbf{1}$, otherwise we set $W_{01}^-[i] = \mathbf{0}$. The rationale is that if no values $B[q]$ with $0 < B[q] < k$ occur between two nodes, then the associated (reversed) k-mers have a common LCP of length $k-1$ and therefore if they both have an outgoing edge labelled with c they reach the same node and only the first one should have $W_{01}^-[i] = \mathbf{1}$.

4   Implementation Details and Analysis

Let $n = n_1 + n_0$ denote the sum of number of nodes in G_0 and G_1, and let $m = |W_0| + |W_1|$ denote the sum of the number of edges. The k-mer merging algorithm as described executes in $\mathcal{O}(m)$ time a first pass over the arrays W_0, W_0^-, and W_1, W_1^- to compute the values $\ell_0(c) + \ell_1(c)$ for $c \in \Sigma$ and initialize the arrays $F[1, \sigma]$, $\mathsf{start}[1, \sigma]$, $\mathsf{Block\_id}[1, \sigma]$ and $Z^{(1)}[1, n]$ (Phase 1). Then, the algorithm executes $k-1$ iterations of the code in Fig. 3 each iteration taking $\mathcal{O}(m)$ time. Finally, still in $\mathcal{O}(m)$ time the algorithm computes the succinct representation of the union graph (Phases 2 and 3). The overall running time is therefore $\mathcal{O}(m\,k)$.

We now analyze the space usage of the algorithm. In addition to the input and the output, our algorithm uses $2n$ bits for two instances of the $Z^{(\cdot)}$ array (for the current $Z^{(h)}$ and for the previous $Z^{(h-1)}$), plus $n\lceil \log k \rceil$ bits for the B array. Note, however, that during iteration h we only need to check whether $B[i]$ is equal to 0, h, or some value within 0 and h. Similarly, for the computation of W_{01}^- we only need to distinguish between the cases where $B[i]$ is equal to 0, k or some value $0 < B[i] < k$. Therefore, we can save space replacing $B[1, n]$ with an array $B_2[1, n]$ containing two bits per entry representing the four possible states $\{0, 1, 2, 3\}$. During iteration h, the values in B_2 are used instead of the ones in B as follows: An entry $B_2[i] = 0$ corresponds to $B[i] = 0$, an entry $B_2[i] = 3$ corresponds to an entry $0 < B[i] < h - 1$. In addition, if h is even, an entry $B_2[i] = 2$ corresponds to $B[i] = h$ and an entry $B_2[i] = 1$ corresponds to $B[i] = h - 1$; while if h is odd the correspondence is $2 \to h - 1$, $1 \to h$. The reason for this apparently involved scheme, first introduced in [6], is that during phase h, an entry in B_2 can be modified either before or after we have read it at Line 9. Using this technique, the working space of the algorithm, i.e., the space in addition to the input and the output, is $4n$ bits plus $3\sigma + \mathcal{O}(1)$ words of RAM for the arrays start, F, and $\mathsf{Block\_id}$.

Theorem 1. *The merging of two succinct representations of two order-k de Bruijn graphs can be done in $\mathcal{O}(m\,k)$ time using $4n$ bits plus $\mathcal{O}(\sigma)$ words of working space.*   □

We stated the above theorem in terms of working space, since the total space depends on how we store the input and output, and for such storage there are several possible alternatives. The usual assumption is that the input de Bruijn graphs, i.e. the arrays $\langle W_0, W_0^-, \mathsf{last}_0 \rangle$ and $\langle W_1, W_1^-, \mathsf{last}_1 \rangle$, are stored in RAM using overall $m \log \sigma + 2m$ bits. Since the three arrays representing the output de Bruijn graph are generated sequentially in one pass, they are usually written directly to disk without being stored in RAM, so they do not contribute to the total space usage. Also note that during each iteration of the algorithm in Fig. 3, the input arrays are all accessed sequentially. Thus we could keep them on disk reducing the overall RAM usage to just $4n$ bits plus $\mathcal{O}(\sigma)$ words; the resulting algorithm would perform additional $\mathcal{O}(k(m \log \sigma + 2m)/D)$ I/Os where D denotes the disk page size in bits.

Comparison with the State of the Art. The de Bruijn graph merging algorithm by Muggli *et al.* [16,17] is similar to ours in that it has a *planning phase* consisting of the colexicographic sorting of the $(k + 1)$-mers associated to the edges of G_0 and G_1. To this end, the algorithm uses a standard MSD radix sort. However only the most significant symbol of each $(k+1)$-mer is readily available in W_0 and W_1. Thus, during each iteration the algorithm computes also the next symbol of each $(k+1)$-mer that will be used as a sorting key in the next iteration. The overall space for such symbols is $2m\lceil \log \sigma \rceil$ bits, since for each edge we need the symbol for the current and next iteration. In addition, the algorithm uses up to $2(n + m)$ bits to maintain the set of intervals consisting in edges whose associated reversed $(k+1)$-mer have a common prefix; these intervals correspond to the blocks we implicitly maintain in the array B_2 using only $2n$ bits.

Summing up, the algorithm by Muggli *et al.* runs in $\mathcal{O}(mk)$ time, and uses $2(m\lceil \log \sigma \rceil + m + n)$ bits plus $\mathcal{O}(\sigma)$ words of working space. Our algorithm has the same time complexity but uses less space: even for $\sigma = 5$ as in bioinformatics applications, our algorithm uses less than half the space ($4n$ bits vs. $6.64m + 2n$ bits). This space reduction significantly influences the size of the largest de Bruijn graph that can be built with a given amount of RAM. For example, in the setting in which the input graphs are stored on disk and all the RAM is used for the working space, our algorithm can build a de Bruijn graph whose size is twice the size of the largest de Bruijn graph that can be built with the algorithm of Muggli *et al.*.

We stress that the space reduction was obtained by substantially changing the sorting procedure. Although both algorithms are based on radix sorting they differ substantially in their execution. The algorithm by Muggli *et al.* follows the traditional MSD radix sort strategy; hence it establishes, for example, that $ACG \prec ACT$ when it compares the third 'digits' and finds that $G < T$. In our algorithm we use a mixed LSD/MSD strategy: in the above example we also find that $ACG \prec ACT$ during the third iteration, but this is established without comparing directly G and T, which are not explicitly available. Instead, during

the second iteration the algorithm finds that $CG \prec CT$ and during the third iteration it uses this fact to infer that $ACG \prec ACT$: this is indeed a remarkable sorting trick first introduced in [12] and adapted here to de Bruijn graphs.

5   Merging Colored and VO-BOSS Representations

Our algorithm can be easily generalized to merge colored and VO (variable-order) BOSS representations. Note that the algorithm by Muggli *et al.* can also merge colored BOSS representations, but in its original formulation, it cannot merge VO representations.

   Given the colored BOSS representation of two de Bruijn graphs G_0 and G_1, the corresponding color matrices $\mathcal{M}_0$ and $\mathcal{M}_1$ have size $m_0 \times c_0$ and $m_1 \times c_1$. We initially create a new color matrix $\mathcal{M}_{01}$ of size $(m_0 + m_1) \times (c_0 + c_1)$ with all entries empty. During the merging of the union graph (Phase 3), for $q = 1, \ldots, n$, we write the colors of the edges associated to $Z^{(h)}[q]$ to the corresponding line in $\mathcal{M}_{01}$ possibly merging the colors when we find nodes with identical k-mers in $\mathcal{O}(c_{01})$ time, with $c_{01} = c_0 + c_1$. To make sure that color ids from $\mathcal{M}_0$ are different from those in $\mathcal{M}_1$ in the new graph we add the constant c_0 (the number of distinct colors in G_0) to any color id coming from the matrix $\mathcal{M}_1$.

Theorem 2. *The merging of two succinct representations of colored de Bruijn graphs takes $\mathcal{O}(m \max(k, c_{01}))$ time and $4n$ bits plus $\mathcal{O}(\sigma)$ words of working space, where $c_{01} = c_0 + c_1$.* □

   We now show that we can compute the variable order VO-BOSS representation of the union of two de Bruijn graphs G_0 and G_1 given their *plain*, eg. non variable order, BOSS representations. For the VO-BOSS representation we need the LCS array for the nodes in the union graph $\langle W_{01}, W_{01}^-, \mathsf{last}_{01} \rangle$. Notice that after merging the k-mers of G_0 and G_1 with the algorithm in Fig. 3 (Phase 1) the values in $B[1, n]$ already provide the LCP information between the reverse labels of all consecutive nodes (Lemma 2). When building the union graph (Phase 3), for $q = 1, \ldots, n$, the LCS between two consecutive nodes, say v_i and w_j, is equal to the LCP of their reverses $\overleftarrow{v_i}$ and $\overleftarrow{w_j}$, which is given by $B[q] - 1$ whenever $B[q] > 0$ (if $B[q] = 0$ then $\overleftarrow{v_i} = \overleftarrow{w_j}$ and nodes v_i and v_j should be merged). Hence, our algorithm for computing the VO representation of the union graph consists exactly of the algorithm in Fig. 3 in which we store the array B in $n \log k$ bits instead of using the 2-bit representation described in Sect. 4. Hence the running time is still $\mathcal{O}(mk)$ and the working space becomes the space for the bitvectors $Z^{(h-1)}$ and $Z^{(h)}$ (recall we define the working space as the space used in addition to the space for the input and the output).

Theorem 3. *Merging two succinct representations of variable order de Bruijn graphs takes $\mathcal{O}(mk)$ time and $2n$ bits plus $\mathcal{O}(\sigma)$ words of working space.* □

6   External Memory Construction

In this section we show that using our merging algorithm we can design a complete external memory algorithm to construct succinct de Bruijn graphs.

We preliminary observe that at each iteration of the algorithm in Fig. 3 not only the arrays $\langle W_0, W_0^-, \mathsf{last}_0 \rangle$ and $\langle W_1, W_1^-, \mathsf{last}_1 \rangle$ but also $Z^{(h-1)}$ and B_2 are read sequentially from beginning to end. At the same time, the arrays $Z^{(h)}$ and B_2 are written sequentially but into σ different partitions whose starting positions are the values in $\mathsf{start}[1, \sigma]$ which are the same for each iteration. Thus, if we split $Z^{(\cdot)}$ and B_2 into σ different files, all accesses are sequential and our algorithm runs in external memory in $\mathcal{O}(mk)$ time, doing $\mathcal{O}(mk)$ sequential I/Os and using only $\mathcal{O}(\sigma)$ words of RAM.

Assume now we are given a string collection $\mathcal{C} = s_1, \dots, s_d$ of total length N, the desired order k, and the amount of available RAM M. First, we split $\mathcal{C}$ into smaller subcollections $r_i = s_j, \dots, s_{j'}$, such that we can compute the BWT and LCP array of each subcollection in linear time in RAM using M bytes, using e.g. the suffix sorting algorithm gSACA-K [14]. For each subcollection we then compute, and write to disk, the BOSS representation of its de Bruijn graph using the algorithm described in [6, Section 5.3]. Since these are linear algorithms the overall cost of this phase is $\mathcal{O}(N)$ time and $\mathcal{O}(N)$ sequential I/Os.

Finally, we merge all de Bruijn graphs into a single BOSS representation of the union graph with the external memory variant just described. Since the number of subcollections is $\mathcal{O}(N/M)$, a total of $\log(N/M)$ merging rounds will suffice to get the BOSS representation of the union graph.

Theorem 4. *Given a strings collection $\mathcal{C} = s_1, \dots, s_d$ of total length N, we can build the corresponding order-k succinct de Bruijn graph in $\mathcal{O}(N\,k \log(N/M))$ time and $\mathcal{O}(N\,k \log(N/M))$ sequential I/Os using $\mathcal{O}(M)$ words of RAM.*   □

Note that our construction algorithm can be easily extended to generate the colored/variable order variants of the de Bruijn graph. For the colored variant it suffices to use gSACA-K to generate also the document array [14] and then use the colored merging variant. For the variable order representation, it suffices to store the LCP/LCS values during the very last merging phase, using the techniques described in [6, Section 3] to handle them in external memory.

References

1. Alipanahi, B., Kuhnle, A., Boucher, C.: Recoloring the colored de Bruijn graph. In: Gagie, T., Moffat, A., Navarro, G., Cuadros-Vargas, E. (eds.) SPIRE 2018. LNCS, vol. 11147, pp. 1–11. Springer, Cham (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: WABI. LIPIcs, vol. 88, pp. 18:1–18:15. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik (2017)

3. Belazzougui, D., Gagie, T., Mäkinen, V., Previtali, M., Puglisi, S.J.: Bidirectional variable-order de Bruijn graphs. Int. J. Found. Comput. Sci. **29**(08), 1279–1295 (2018)

4. Boucher, C., Bowe, A., Gagie, T., Puglisi, S.J., Sadakane, K.: Variable-order de Bruijn graphs. In: DCC, pp. 383–392. IEEE (2015)

5. Bowe, A., Onodera, T., Sadakane, K., Shibuya, T.: Succinct de Bruijn graphs. In: Raphael, B., Tang, J. (eds.) WABI 2012. LNCS, vol. 7534, pp. 225–235. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-33122-0_18

6. Egidi, L., Louza, F.A., Manzini, G., Telles, G.P.: External memory BWT and LCP computation for sequence collections with applications. In: WABI. LIPIcs, vol. 113, pp. 10:1–10:14. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik (2018)

7. Egidi, L., Louza, F.A., Manzini, G., Telles, G.P.: External memory BWT and LCP computation for sequence collections with applications. Algorithms Mol. Biol. **14**(1), 6:1–6:15 (2019)

8. Egidi, L., Manzini, G.: Lightweight BWT and LCP merging via the gap algorithm. In: Fici, G., Sciortino, M., Venturini, R. (eds.) SPIRE 2017. LNCS, vol. 10508, pp. 176–190. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-67428-5_15

9. Egidi, L., Manzini, G.: Lightweight merging of compressed indices based on BWT variants. CoRR (2019). http://arxiv.org/abs/1903.01465

10. Ferragina, P., Manzini, G., Mäkinen, V., Navarro, G.: Compressed representations of sequences and full-text indexes. ACM Trans. Algorithms **3**(2) (2007)

11. Holt, J., McMillan, L.: Constructing Burrows-Wheeler transforms of large string collections via merging. In: BCB, pp. 464–471. ACM (2014)

12. Holt, J., McMillan, L.: Merging of multi-string BWTs with applications. Bioinformatics **30**(24), 3524–3531 (2014)

13. 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)

14. Louza, F.A., Gog, S., Telles, G.P.: Inducing enhanced suffix arrays for string collections. Theor. Comput. Sci. **678**, 22–39 (2017)

15. Marcus, S., Lee, H., Schatz, M.C.: Splitmem: a graphical algorithm for pan-genome analysis with suffix skips. Bioinformatics **30**(24), 3476–3483 (2014)

16. Muggli, M.D., Alipanahi, B., Boucher, C.: Building large updatable colored de Bruijn graphs via merging. Bioinformatics **35**(14), i51–i60 (2019). https://doi.org/10.1093/bioinformatics/btz350

17. Muggli, M.D., Boucher, C.: Succinct de Bruijn graph construction for massive populations through space-efficient merging. bioRxiv (2017). https://doi.org/10.1101/229641

18. Muggli, M.D., et al.: Succinct colored de Bruijn graphs. Bioinformatics **33**(20), 3181–3187 (2017)

19. Pevzner, P.A., Tang, H., Waterman, M.S.: An Eulerian path approach to DNA fragment assembly. Proc. Natl. Acad. Sci. **98**(17), 9748–9753 (2001)
20. Raman, R., Raman, V., Rao, S.: Succinct indexable dictionaries with applications to encoding k-ary trees, prefix sums and multisets. ACM Trans. Algorithms **3**(4) (2007)