# Space efficient merging of de Bruijn graphs and Wheeler graphs*

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

[1]University of Eastern Piedmont, Alessandria, Italy
[2]Federal University of Uberlândia, Uberlândia, Brazil
[3]University of Pisa, Pisa, Italy

**Abstract**

The merging of succinct data structures is a well established technique for the space efficient construction of large succinct indexes. In the first part of the paper we propose a new algorithm for merging succinct representations of *de Bruijn graphs*. Our algorithm has the same asymptotic cost of the state of the art algorithm for the same problem 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. In the second part of the paper we consider the more general problem of merging succinct representations of *Wheeler graphs*, a recently introduced graph family which includes as special cases de Bruijn graphs and many other known succinct indexes based on the BWT or one of its variants. In this paper we provide a space efficient algorithm for Wheeler graph merging; our algorithm works under the assumption that the union of the input Wheeler graphs has an ordering that satisfies the Wheeler conditions and which is compatible with the ordering of the original graphs.

## 1 Introduction

A fundamental parameter of any construction algorithm for succinct data structures is its *space usage*: this parameter determines the size of the largest dataset that can be handled by a machine with a given amount of memory. Recent works [22, 33, 36, 42] have shown that the technique of building large indexing data structures by merging or updating smaller ones is one of the most effective for designing space efficient algorithms.

In the first part of the paper we consider the *de Bruijn* graph for a collection of strings, which is a key data structure for genome assembly [40]. After the seminal work of Bowe *et al.* [12], many succinct representations of this data structure have been proposed in the literature (*e.g.* [14, 5, 10, 11, 38, 9]) 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.* [36, 37] have proposed a merging algorithm for 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 [36] 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

*Postprint version. The final authenticated version is available online at `https://doi.org/10.1007/s00453-021-00855-2`. A preliminary version of the results in Section 3 appeared in the Proc. 26th Symposium on String Processing and Information Retrieval (SPIRE 2019) [17].

the de Bruijn graph. For a graph with $m$ edges and $n$ nodes the merging algorithm by Muggli *et al.* uses $2(m \log \sigma + m + n)$ bits plus $\mathcal{O}(\sigma)$ words of working space, where $\sigma$ is the alphabet size (the working space is defined as the space used by the algorithm in addition to the space used for the input and the output). This value represents a three fold improvement over previous results, but it is still larger than the size of the resulting succinct representation of the de Bruijn graph, which is upper bounded by $2(m \log \sigma + m) + o(m)$ bits.

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 out-degree $m/n$. Notice that the working space of our algorithm is always less than the space of the resulting succinct de Bruijn graph. Our new merging algorithm is based on a mixed LSD/MSD Radix Sort algorithm which is inspired by the lightweight BWT merging introduced by Holt and McMillan [30, 31] and later improved in [20, 21]. 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 in Bowe *et al.*'s representation, thus making it possible to construct succinct Variable Order de Bruijn graph [11], a feature not shared by any other merging algorithm.

In the second part of this paper, we address the issue of generalizing the results on de Bruijn graphs, and some previous results on succinct data structure merging [21, 31], to *Wheeler graphs*. The notion of Wheeler graph has been recently introduced in [28] to provide a unifying view of a large family of compressed data structures loosely based on the BWT [13] or one of its variants. Among the others, the FM-index [23], the XBWT [25], and the BOSS representation of de Bruijn graphs can all be seen as special cases of (succinct) Wheeler graphs. After their introduction, Wheeler graphs have become objects of independent study and several authors have shown they have some remarkable properties (*e.g.* [1, 3, 7, 15, 27, 29]).

A space efficient algorithm for merging Wheeler graphs would automatically provide a merging algorithm for the many practical succinct data structures, present and future, which have the Wheeler graph structure. Unfortunately, because of their generality, the problem of merging Wheeler graphs appears to be much harder than the problem of merging specific data structures. As we discuss in Section 4, the correct setting for Wheeler graph merging is to consider the language $\mathcal{L}$, defined as the union of the languages recognized by the input graphs when considered as Nondeterministic Finite Automata, and then to build a Wheeler graph recognizing $\mathcal{L}$ (assuming one exists, see [2, Lemma 3.3]). In this paper we address a slightly simpler problem: we consider the union graph, a graph guaranteed to recognize $\mathcal{L}$, and ask whether there is an ordering of its nodes that makes it a Wheeler graph, with the additional restriction that such ordering must be compatible with the Wheeler orderings of the graphs that contribute to the union. By compatible, we mean that the relative order of the nodes from each single graph are preserved. Although determining if a graph is a Wheeler graph is NP-complete in the general case [29], for the special case of the union graph, and with the additional restriction mentioned above, we show that the problem can be solved in quadratic time via a reduction to the 2-SAT problem. We also describe a space efficient algorithm that is guaranteed to return a Wheeler graph recognizing the language $\mathcal{L}$ under the condition that the union graph has a compatible Wheeler ordering (and sometimes even when this condition is not satisfied, as discussed in Section 5.1). If the union graph has $n$ nodes, our algorithm takes $\mathcal{O}(n^2)$ time and only uses $4n + o(n)$ bits of working space. A fine point is that sometimes our algorithm does not return the union graph itself but a smaller Wheeler graph recognizing $\mathcal{L}$: this is not relevant for succinct data structures but it is a further indication that the problem should be studied looking at the properties of the union language $\mathcal{L}$.

To our knowledge we are the first to tackle the problem of Wheeler graphs merging. Although we do not solve this problem in its full generality, our results show that it is possible to perform non trivial operations on a succinct representation of Wheeler graphs using a small working space: extending this result would make them even more appealing as general tools for establishing properties of an important class of succinct data structures.

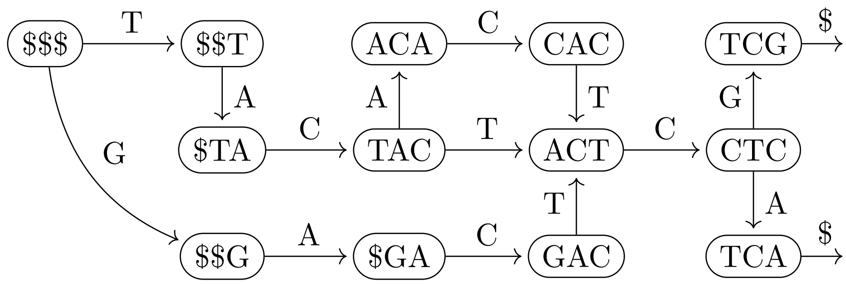



Figure 1: de Bruijn graph for $\mathcal{C} = \{\text{TACACT, TACTCG, GACTCA}\}$.

## 2 Background and notation

Let $\Sigma = \{1, 2, \ldots, \sigma\}$ denote the canonical alphabet of size $\sigma$. Let $s[1, n]$ denote a string of length $n$ over $\Sigma$. Given two strings $s_1$ and $s_2$, we write $s_1 \prec s_2$ to denote that $s_1$ is lexicographically smaller than $s_2$. Given a string $s[1, n]$ and $c \in \Sigma$, we write $\mathsf{rank}_c(s, i)$ to denote the number of occurrences of $c$ in $s[1, i]$, and $\mathsf{select}_c(s, j)$ to denote the position of the $j$-th $c$ in $s$. We define $\mathsf{select}_c(s, 0) = 0$. In this paper we assume a RAM model with word size $w$ with $\sigma = w^{\mathcal{O}(1)}$. This ensures that we can represent any string $s$ in $|s| \log \sigma + o(|s|)$ bits, or even $H_0(s) + o(|s|)$ bits, and support $\mathsf{rank}$ and $\mathsf{select}$ queries in constant time [8, Theorem 7], where $H_0(s)$ is the empirical *zero-order* entropy of $s$.

### 2.1 de Bruijn graphs

Given a collection of strings $\mathcal{C} = s_1, \ldots, s_d$ over $\Sigma$, 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 $v[1] \cdots v[k]$ 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 $v[k]$.

In 2012, Bowe *et al.* [12] 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 $\overleftarrow{v} = v[k] \cdots v[1]$ 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 $W_i = \$$. Finally, we define (see examples in Figs. 1 and 2):

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.

The length $m$ of the arrays $W$, $W^-$, and $\mathsf{last}$ 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 $\mathsf{last}$ is equal to the number of nodes

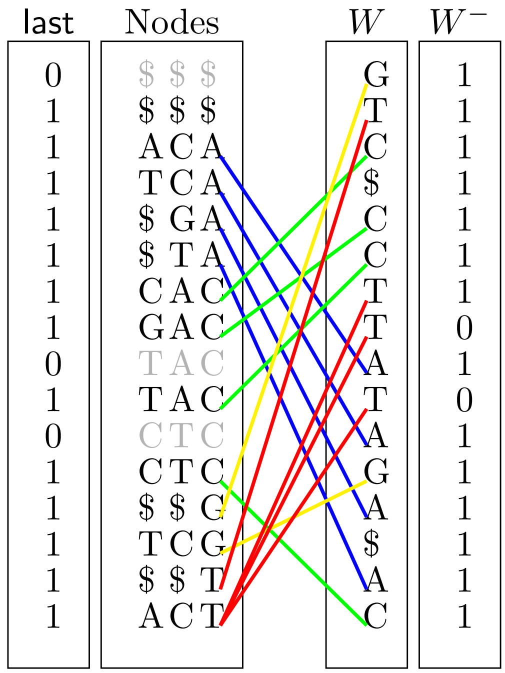



Figure 2: BOSS representation of the graph in Fig. 1. The colored lines connect each label in $W$ to its destination node; edges of the same color have the same label. Note that edges with the same label/color reach distinct ranges of nodes, and that edges with the same label/color do not cross.

$n$, 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. Bowe *et al.* observed 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 set $\{2, \ldots, n\}$: in this correspondence $LF(i) = j$ iff $v_j$ is the destination node of the edge associated to $W[i]$. The $LF$ correspondence is order preserving in the sense that if $W^-[i] = W^-[j] = \mathbf{1}$ then

$$\begin{aligned} W[i] < W[j] \quad &\Longrightarrow \quad LF(i) < LF(j), \\ (W[i] = W[j]) \wedge (i < j) \quad &\Longrightarrow \quad LF(i) < LF(j). \end{aligned} \tag{2}$$

Bowe *et al.* have shown that enriching the arrays $W$, $W^-$, and $\mathsf{last}$ with the data structures from [24, 41] supporting constant time rank and select operations, we can efficiently navigate the de Bruijn graph $G$. The overall cost of encoding the three arrays and the auxiliary data structures is bounded by $m \log \sigma + 2m + \sigma \log n + o(m)$ bits, with the usual time/space tradeoffs available for rank/select data structures (see [12] for details).

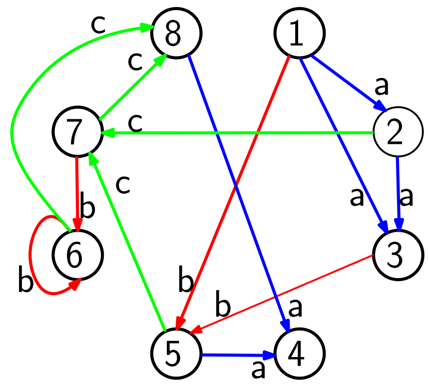
## 2.2 Wheeler graphs

**Definition 1.** A directed labeled graph $G = (V, E)$ is a *Wheeler graph* if there is an ordering of the nodes such that nodes with in-degree 0 precede those with positive in-degree and, for any pair of edges $e = (u, v)$ and $e' = (u', v')$ labeled with $a$ and $a'$ respectively, the following monotonicity properties hold:

$$a < a' \Longrightarrow v < v'\,, \tag{3a}$$

$$(a = a') \wedge (u < u') \Longrightarrow v \leq v'\,. \tag{3b}$$

□




$I = 10100100100100100100 1$

$O = 00010010110010010010 1$

$L =$ aab ac b ac bc ac a

$C' = 1000010001000$

Figure 3: An example of an eight node Wheeler graph (left) and its succinct representation (right). Node 1 has no incoming edges, nodes 2–4 have incoming edges labeled a, nodes 5–6 have incoming edges labeled b, nodes 7–8 have incoming edges labeled c. The binary arrays $I$ and $O$ are the unary representation of the nodes' in-degrees and out-degrees, the array $L$ stores the labels of the outgoing edges. The binary array $C'$ encodes the number of occurrences of any give symbol.

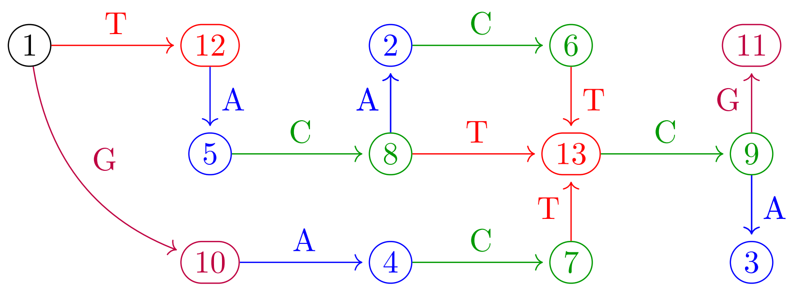
It is easy to see that a de Bruijn graph with the nodes sorted according to (1) is a Wheeler graph. Informally, for the graph of Figs. 1 and 2 property (3a) follows from the fact that edges with different labels/colors reach non interleaving ranges of nodes, and that edges with the same label/color do not cross. Note that property (1) coincides with (3a) and (3b) restricted to the case in which the destination nodes are distinct.

Gagie *et al.* [28] proposed the following compact representation for a Wheeler graph $G = (V, E)$ with $|V| = n$ and $|E| = m$. Let $x_1 < x_2 < \cdots < x_n$ denote the ordered set of nodes. For $i = 1, \ldots, n$ let $\ell_i$ and $k_i$ denote respectively the out-degree and in-degree of node $x_i$. Define the binary arrays of length $n + m$

$$O \;=\; \mathbf{0}^{\ell_1}\mathbf{1}\,\mathbf{0}^{\ell_2}\mathbf{1}\cdots\mathbf{0}^{\ell_n}\mathbf{1}, \qquad I \;=\; \mathbf{0}^{k_1}\mathbf{1}\,\mathbf{0}^{k_2}\mathbf{1}\cdots\mathbf{0}^{k_n}\mathbf{1}. \tag{4}$$

Note that $O$ (resp. $I$) consists of the concatenated unary representations of the out-degrees (resp. in-degrees). Let $L_i$ denote the sorted set of symbols on the edges leaving from $x_i$, and let $L[1..m]$ denote the concatenation $L = L_1 L_2 \cdots L_n$. Finally, let $C[1..\sigma]$ denote the array such that $C[c]$ is the number of edges with label smaller than $c \in \Sigma$ (we assume every distinct symbol labels some edge). As an alternative to $C[1..\sigma]$ one can use the binary array $C'[1..m]$ such that $C'[i] = \mathbf{1}$ iff $i = 1 + C[c]$ for some $c = 1..\sigma$. Fig. 3 shows an example of a Wheeler graph and its succinct representation; note that $C$ and $C'$ contains the same information as $L$, and are used only to speed up navigation. In [28] it is shown that we can efficiently navigate the Wheeler graph $G$ using the arrays $I, O, L$ and $C$ (or $C'$) and auxiliary data structures supporting constant time rank/select operations.

As an example of how navigation works, suppose that given node $v$ we want to compute the smallest $u$ such that $E$ contains the edge $(u, v)$, assuming $v$ has positive indegree. Both $u$ and $v$ are identified by their lexicographic rank. By construction, the desired edge corresponds to the first $\mathbf{0}$ in the unary representation of $v$'s indegree. Such $\mathbf{0}$ is in position $k = 1 + \mathsf{select}_1(I, v - 1)$ of $I$. As a running example, consider $v = 5$ in Fig. 3; we have $k = 1 + \mathsf{select}_1(I, 5 - 1) = 10$. The symbol on the edge is $c = \mathsf{rank}_1(C', \mathsf{rank}_0(I, k))$, since edges in $C'$ are ordered by their labels in increasing order. In our running example, $c = \mathsf{rank}_1(C', \mathsf{rank}_0(I, 10)) = 2$, that is, the second symbol in $\Sigma$, which is $\mathbf{b}$. The edge is the $j$-th one with label $c$ where $j = 1 + \mathsf{rank}_0(I, k) - \mathsf{select}_1(C', c)$ (there are $\mathsf{rank}_0(I, k)$ edges before that one, but $\mathsf{select}_1(C', c)$ have different labels). Hence, the symbol of that edge is the one in position $h = \mathsf{select}_c(L, j)$ in $L$ and therefore corresponds to the $h$-th $\mathbf{0}$ in $O$. In our running example, the $j$-th edge labeled with $c = \mathbf{b}$ is given by $j = 1 + \mathsf{rank}_0(I, 10) - \mathsf{select}_1(C', 2) = 1 + 6 - 6 = 1$, that is, $(u, v)$ is the first one with label $\mathbf{b}$, and $h = \mathsf{select}_b(L, 1) = 3$. The desired node $u$ is therefore the node whose outdegree unary




$I =$ `101010101010101010101010001`

$O =$ `001011010101010010010110101`

$L =$ `TG C  C C T T AT AG A  A C`

$C' =$ `10001000101000`

Figure 4: Wheeler graph representation for the de Bruijn graph presented in Fig. 1.

representation in $O$ contains the $h$-th bit $\mathbf{0}$ of $O$, that is $u = 1 + \mathsf{rank}_1(O, \mathsf{select}_0(O, h))$. This, in our running example, is $u = 1 + \mathsf{rank}_1(O, \mathsf{select}_0(O, 3)) = 1 + \mathsf{rank}_1(O, 3) = 1 + 0 = 1$. Therefore, the edge $(u, v) = (1, 5)$ is labeled with $\mathbf{b}$.

More in general, combining [28] with the succinct representations from [8] we get the following result.

**Lemma 2.** *It is possible to represent an $n$-node, $m$-edge Wheeler graph with labels over the alphabet $\Sigma$ in $2(n + m) + m \log \sigma + \sigma \log m + o(n + m \log \sigma)$ bits. The representation supports forward and backward traversing of the edges in $\mathcal{O}(1)$ time assuming $\sigma = w^{\mathcal{O}(1)}$, where $w$ is the word size.* □

Note that there are strong similarities between the BOSS representation and the Wheeler graph succinct representation, with the arrays $W$ and $\mathsf{last}$ corresponding respectively to $L$, and $O$, while the array $W^-$ is a permutation of $I$. This is not surprising since the latter was inspired by the former. The BOSS representation is more efficient since we can assume each (in)out-degree $d$ is positive so we unary encode $d - 1$ instead of $d$ saving $n$ bits in $W^-$ and $\mathsf{last}$. (For simplicity we omitted from the BOSS representation an array, called $F$ in [12], which corresponds to the $C$ array of Wheeler graphs).

Fig. 4 shows the de Bruijn graph of Fig. 1 as a Wheeler graph and its corresponding succinct representation. Note that in the Wheeler graph representation the outgoing edges labeled \$ from nodes 3 and 11 in are not necessary since, since the representation allows edges with out-degree zero.

The importance of Wheeler graphs comes from the observation in [28] that many succinct data structures supporting efficient substring queries [16, 23, 25, 26, 34, 39, 43] can be seen as Nondeterministic Finite Automata (NFA) whose states can be sorted so that the resulting graph is a Wheeler graph. We leave the details to [28] and formalize this notion with the following definitions.

**Definition 3.** A nondeterministic finite automata (NFA) is a quintuple $\mathcal{A} = (V, E, F, s, \Sigma)$, where $V$ is a set of states (or nodes), $\Sigma$ is the alphabet (set of labels), $E \subseteq V \times V \times \Sigma$ is a set of directed labeled edges, $F \subseteq V$ is a set of accepting states, and $s \in V$ is the start state. We additionally require that $s$ is the only state with in-degree 0, that each node is reachable from $s$, and that from each node we can reach a final state. □

**Definition 4.** A *Wheeler automaton* is a NFA without $\epsilon$-transitions for which there is an ordering of the states that makes the state diagram a Wheeler graph. □

The reader should refer to [1] to see that Definition 3 is not restrictive and for further details and properties of Wheeler automata. With a little abuse of notation in the following we will use the terms Wheeler graph and Wheeler automaton as synonymous.

# 3 Merging BOSS representations of de Bruijn graphs

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. In Section 3.5 we compare our solution with the state of the art algorithm for the same problem.

Let $G_0$ and $G_1$ denote respectively the (uncompressed) de Bruijn graphs for $\mathcal{C}_0$ and $\mathcal{C}_1$, and let

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

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

$$\overleftarrow{v_1} \prec \cdots \prec \overleftarrow{v_{n_0}} \qquad \text{and} \qquad \overleftarrow{w_1} \prec \cdots \prec \overleftarrow{w_{n_1}} \tag{5}$$

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 $W_0^-$, $\mathsf{last}_0$ and $W_1^-$, $\mathsf{last}_1$.

## 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 [30, 31] 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. 5 (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 $h = 2, 3, \dots, k$, 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 5.** For $i = 1, \dots, n_0$ and $j = 1, \dots n_1$ the $i$-th $\mathbf{0}$ precedes the $j$-th $\mathbf{1}$ in $Z^{(h)}$ if and only if $\overleftarrow{v_i}[1,h] \preceq \overleftarrow{w_j}[1,h]$. □

Property 5 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 5 holds, we first define $Z^{(1)}$ and show that it satisfies the property, then we prove that for $h = 2, \dots, k$ the code in Fig. 5 computes $Z^{(h)}$ that still satisfies Property 5.

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

$$Z^{(1)} = \underline{\mathbf{01}}\ \underline{\mathbf{0}^{\ell_0(1)}\mathbf{1}^{\ell_1(1)}}\ \underline{\mathbf{0}^{\ell_0(2)}\mathbf{1}^{\ell_1(2)}} \cdots \underline{\mathbf{0}^{\ell_0(\sigma)}\mathbf{1}^{\ell_1(\sigma)}}\ . \tag{6}$$

```
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
```

Figure 5: 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 Section 3.2.

The first pair $\mathbf{01}$ in $Z^{(1)}$ accounts for $v_1$ and $w_1$; for each $c \in \Sigma$ the 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 $\sigma$ 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 (5), 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 5, 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. 5 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 denotes the graph $G_b$ 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. 5.

**Lemma 6.** *For $h = 2, \ldots, k$, the array $Z^{(h)}$ computed by the algorithm in Fig. 5 satisfies Property 5.*

*Proof.* To prove the "if" part of Property 5 let $1 \leq f < g \leq n_0 + n_1$ denote two indexes such that $Z^{(h)}[f]$ is the $i$-th **0** and $Z^{(h)}[g]$ is the $j$-th **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 **1** would have been written in a subarray of $Z^{(h)}$ preceding the one where the $i$-th **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 **0** and the $j$-th **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. 5 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 **0** in $Z^{(h-1)}$ and $g'$ is the $j'$-th **1** in $Z^{(h-1)}$. By the inductive hypothesis on $Z^{(h-1)}$ it is

$$\overleftarrow{v_{i'}}[1,h-1] \ \preceq\ \overleftarrow{w_{j'}}[1,h-1]. \tag{7}$$

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

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

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

For the "only if" part of Property 5, 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. □ □

### 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 5, we identify the $i$-th **0** in $Z^{(h)}$ with $\overleftarrow{v_i}$ and the $j$-th **1** in $Z^{(h)}$ with $\overleftarrow{w_j}$. For $h = 2, \ldots, k$, let $b(h) + 1$ be the number of $h$-blocks. Property 5 is equivalent to state that we can logically partition $Z^{(h)}$ into $b(h) + 1$ $h$-blocks

$$Z^{(h)}[1, \ell_1],\ Z^{(h)}[\ell_1 + 1, \ell_2],\ \ldots,\ Z^{(h)}[\ell_{b(h)} + 1, n_0 + n_1] \tag{8}$$

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. 5, 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 $Z^{(h)}[q]$ corresponds to a $k$-mer that has the form $c\alpha$, while $Z^{(h)}[q-1]$ to one with form $c\beta$, with $\alpha$ and $\beta$ 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 [20].

**Lemma 7.** *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 $W_{01}$, $W^-_{01}$, $\mathsf{last}_{01}$ 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 lexicographic order), writing at the same time the representation of the out-degree of the new node to $\mathsf{last}_{01}$. 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 $\mathsf{last}_{01}$. 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 labeled with $c$ they reach the same node and only the first one should have $W^-_{01}[i] = \mathbf{1}$.

### 3.4 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. 5 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 $\{\mathit{0}, \mathit{1}, \mathit{2}, \mathit{3}\}$. During iteration $h$, the values in $B_2$ are used instead of the ones in $B$ as follows: An entry $B_2[i] = \mathit{0}$ corresponds to $B[i] = 0$, an entry $B_2[i] = \mathit{3}$ corresponds to an entry $0 < B[i] < h-1$. In addition, if $h$ is even, an entry $B_2[i] = \mathit{2}$ corresponds to $B[i] = h$ and an entry $B_2[i] = \mathit{1}$ corresponds to $B[i] = h-1$; while if $h$ is odd the correspondence is $\mathit{2} \rightarrow h-1$, $\mathit{1} \rightarrow h$. The reason for this apparently involved scheme, first introduced in [18], is that during phase $h$, an entry in $B_2$ can be modified either before or after we have read it at Line 9. To update $B_2$ it suffices to replace Lines 19–21 with instructions for setting to **1** the appropriate bit of $B_2[q]$. In two iterations these updates will correctly transform a value $B_2[i] = \mathit{0}$, meaning $B[i] = 0$, into the value $B_2[i] = \mathit{3}$, meaning $0 < B[i] < h-1$. For instance if, when $h$ is even, $B_2[i]$ is set to $\mathit{2}$, at the following iteration $h' = h+1$ (odd), $B_2[i] = \mathit{2}$ will stand for $B[i] = h'-1$, and set to $B_2[i] = \mathit{3}$. Then at the following iterations, $h'' > h'$, $B_2[i] = \mathit{3}$ stands for $0 < B[i] < h''-1$. 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 $\mathsf{start}$, $F$, and $\mathsf{Block\_id}$.

**Theorem 8.** *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. 5, 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.

### 3.5 Comparison with the state of the art

The de Bruijn graph merging algorithm by Muggli *et al.* [36, 37] 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 [31] and adapted here to de Bruijn graphs.

### 3.6 Merging colored and variable order representations

The *colored de Bruijn graph* [32] is an extension of de Bruijn graphs for a collection of 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 [5, 35, 38]. Recently, Alipanah *et al.* [4] presented a different approach to reduce the size of $\mathcal{M}$ by recoloring.

Another variant of de Bruijn graph is the *variable order succinct de Bruijn graph*. The order $k$ of a de Bruijn graph is an important parameter for genome assembling algorithms. When $k$ is small the graph can be too small and uninformative, whereas when $k$ is large the graph can become too large or disconnected. To add flexibility to the BOSS representation, Boucher *et al.* [11] 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.

In this section we show that our algorithm can be easily generalized to merge colored and VO-BOSS representations. Note that the merging algorithm by Muggli *et al.* can also merge colored BOSS representations, but in its original formulation, it cannot merge VO-BOSS 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 id's 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 9.** *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. 5 (Phase 1) the values in $B[1, n]$ already provide the LCP information between the reverse labels of all consecutive nodes (Lemma 7). 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. 5 in which we store the array $B$ in $n \log k$ bits instead of using the 2-bit representation described in Section 3.4. 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 10.** *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.* □

Note the LCP values can be written directly to disk, using for example the technique from [19].

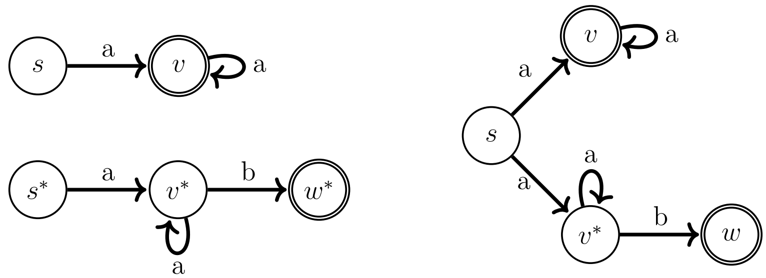



Figure 6: Two Wheeler automata (left) and their union (right). The accepting states are denoted, as usual, with a double circle. The union automaton is not Wheeler (see text), but if we make $v^*$ final the automaton on the lower left accepts the union language that is therefore Wheeler.

# 4 Merging Wheeler graphs via 2-SAT

Merging two de Bruijn graphs $G_0$ and $G_1$, or other succinct indices [21], amounts to building a new succinct data structure that supports the retrieval of the elements which are in $G_0$ or in $G_1$. Because of the correspondence between succinct data structures and Wheeler graphs, the natural generalization of the problem of merging succinct indices is the following problem:

**Problem 11.** Given two Wheeler automata $\mathcal{A}_0 = (V_0, E_0, F_0, s_0, \Sigma)$ and $\mathcal{A}_1 = (V_1, E_1, F_1, s_1, \Sigma)$ recognizing respectively the languages $\mathcal{L}_0$ and $\mathcal{L}_1$ find a Wheeler automaton $\mathcal{A}$ recognizing the union language $\mathcal{L} = \mathcal{L}_0 \cup \mathcal{L}_1$, or report that none exists.

Unfortunately, in the general case we cannot even guarantee that a Wheeler automaton recognizing $\mathcal{L}$ exists, since the property of languages of being recognizable by a Wheeler automata is not closed under union [2, Lemma 3.3]. Hence, instead of tackling the general problem, in this paper we reason in terms of automata rather than languages. Starting from $\mathcal{A}_0$ and $\mathcal{A}_1$ we define the union automaton $\mathcal{U}$ that naturally recognizes $\mathcal{L}$ and we consider the problem of determining if there exists an ordering of $\mathcal{U}$'s nodes that makes it a Wheeler graph (with an additional requirement detailed below). Formally, we define the union automaton (or graph) $\mathcal{U} = (V, E, F, s, \Sigma)$ as follows:

1. $V = (V_0 \setminus \{s_0\}) \cup (V_1 \setminus \{s_1\}) \cup \{s\}$;

2. $E = E_0^* \cup E_1^*$ where, for $i = 0, 1$, $E_i^*$ is $E_i$ where each edge leaving $s_i$ is replaced by an edge leaving $s$ with the same destination;

3. $F = (F_0 \setminus \{s_0\}) \cup (F_1 \setminus \{s_1\})$; with $s$ added to $F$ if $s_0 \in F_0$ or $s_1 \in F_1$.

It is immediate to see that $\mathcal{U}$ recognizes $\mathcal{L} = \mathcal{L}_0 \cup \mathcal{L}_1$ and still has the property that the initial state $s$ is the only one with in-degree 0.

**Definition 12.** A *Wheeler Compatible Order* (Wheeler C-order) for the union automaton $\mathcal{U}$ is an ordering of $V$ that makes $\mathcal{U}$ a Wheeler graph and that is compatible with the orderings of $V_0$ and $V_1$, in the sense that if $u, v \in V_0 \setminus \{s_0\}$ (resp. $V_1 \setminus \{s_1\}$) then $u < v$ in $V$ iff $u < v$ in $V_0$ (resp. $V_1$). □

In the rest of this section, we consider the problem of determining whether $\mathcal{U}$ has a Wheeler C-order and, if this is the case, to explicitly find it:

**Problem 13.** Given the union automaton $\mathcal{U}$ of two Wheeler automata, as defined above, find a Wheeler C-order of $\mathcal{U}$ in the sense of Definition 12 or report that none exists.

We observe that a Wheeler C-order does not necessarily exist even if the union language is Wheeler. Fig. 6 shows two Wheeler automata (left) and their union automata (right). The two input automata recognize respectively the languages $a^n$ and $a^n b$ with $n \geq 1$. The automaton on the right recognizes the union language but it is not a Wheeler automaton: clearly by (3a) it must be $s < w$ and the ranks of $v$ and $v^*$ must be between $s$ and $w$ but we cannot find ranks for $v$ and $v^*$ in the ordering that satisfy the Wheeler conditions. Indeed, $s < v^*$ and the edges $(s, v, a)$ and $(v^*, v^*, a)$ implies $v < v^*$; similarly $s < v$ implies $v^* < v$. However, the union language is Wheeler, since it is recognized by the second input automaton (left) with the state $v^*$ made final.

Having established its limitations, in the rest of this section we provide a solution to Problem 13. It is known [29] that in the general case determining if an automaton is a Wheeler automaton is NP-complete. However, for the union automaton $\mathcal{U}$ we show there exists a polynomial time algorithm to determine whether it has a Wheeler C-order. The algorithm provides a Wheeler C-order if one exists and works by transforming our problem into a 2-SAT instance that can then be solved using any polynomial time 2-SAT solver (*e.g.* [6]). The idea of using 2-SAT for recognizing Wheeler automata was introduced in [1]; however, in the general case this approach is viable only if each node has at most two outgoing edges labeled with the same symbol [1, Theorem 3.1]. We show that this limitation can be removed in the special case that the input is the union automaton of two Wheeler automata.

Let $s, u_1, u_2, \ldots, u_n, v_1, v_2, \ldots v_m$ denote the nodes of the union automata, where $u_1 < u_2 < \cdots < u_n$ are the ordered nodes of $\mathcal{A}_0$ and $v_1 < v_2 < \cdots < v_m$ are the ordered nodes of $\mathcal{A}_1$. We need to check if it is possible to "merge" the two sets of ordered nodes so that the Wheeler conditions are met. Our strategy consists in building a set of clauses, with at most 2 literals each, and to show that there exists a Wheeler C-order if and only if all clauses can be satisfied simultaneously.

We introduce $nm$ boolean variables $X_{i,j}$ for $i = 1, .., n$, $j = 1, .., m$, where $X_{i,j}$ represents the condition $(u_i < v_j)$. We introduce a first set of clauses:

$$i = 1, .., n,\ j = 1, .., m-1 \qquad X_{i,j} \Rightarrow X_{i,j+1} \tag{9}$$

$$i = 2, .., n,\ j = 1, .., m \qquad X_{i,j} \Rightarrow X_{i-1,j} \tag{10}$$

$$i = 1, .., n,\ j = 2, .., m \qquad \neg X_{i,j} \Rightarrow \neg X_{i,j-1} \tag{11}$$

$$i = 1, .., n-1,\ j = 1, .., m \qquad \neg X_{i,j} \Rightarrow \neg X_{i+1,j} \tag{12}$$

Informally, these clauses ensure the transitivity of the resulting order (note that (11) for example is equivalent to $(u_i > v_j) \Rightarrow (u_i > v_{j-1})$). Next, we introduce a second set of clauses which ensure that the resulting order makes $\mathcal{U}$ a Wheeler automaton. For each pair of nodes $u_i \in V_0$, $v_j \in V_1$ such that the edges entering $u_i$ are labeled $a$ and the edges entering $v_j$ are labeled $a'$ with $a \neq a'$ we add the clause

$$\begin{cases} X_{i,j} & \text{if } a < a' \\ \neg X_{i,j} & \text{if } a > a' \end{cases} \tag{13}$$

which is equivalent to (3a). Finally, for each symbol $a$ and for every pair of edges $(u_i, u_k) \in E_0$ and $(v_j, v_h) \in E_1$ both labeled $a$ we add the clauses

$$X_{i,j} \Rightarrow X_{k,h} \tag{14}$$

$$\neg X_{i,j} \Rightarrow \neg X_{k,h}. \tag{15}$$

which are equivalent to (3b) (note we cannot have $u_k = v_h$).

**Lemma 14.** *A truth assignment for the variables $X_{i,j}$ that satisfies (9)–(15) induces a Wheeler C-order for the nodes of $\mathcal{U}$. Viceversa, a Wheeler C-order for $\mathcal{U}$ provides a solution for the 2-SAT instance defined by clauses (9)–(15).*

*Proof.* Given an assignment satisfying (9)–(15) consider the ordering of $V$ defined as follows. Node $s$ has the smallest rank, the nodes in $V \cap V_0$ (resp. $V \cap V_1$) have the same order as in $V_0$ (resp. $V_1$) and for each pair $u_i \in V_0$, $v_j \in V_1$ it is $u_i < v_j$ iff $X_{i,j}$ is true. The resulting order is total: the only non trivial condition being the transitivity which is ensured by clauses (9)–(12). In addition, the order makes $\mathcal{U}$ a Wheeler automaton since conditions (3a)–(3b) follow by the hypothesis on $\mathcal{A}_0$ and $\mathcal{A}_1$ if the edges $e$ and $e'$ both belong to $E_0$ or $E_1$, and by (13)–(15) if not. Viceversa, given a Wheeler C-order for $V$ it is straightforward to verify that the assignment $X_{i,j} = (u_i < v_j)$ for $i = 1, \ldots, n$, $j = 1, \ldots, m$ satisfies the clauses (9)–(15). □ □

The following theorem summarizes the results of this section.

**Theorem 15.** *Given two Wheeler automata $\mathcal{A}_0$ and $\mathcal{A}_1$ in $\mathcal{O}(|E_0||E_1|)$ time we can find a Wheeler C-order for the union automata or report that no such order exists.*

*Proof.* The construction of the clauses takes constant time for clause and there are $\mathcal{O}(|V_0||V_1|)$ clauses of type (9)–(12), and $\mathcal{O}(|E_0||E_1|)$ clauses of type (13)–(15). The thesis follows observing that a 2-SAT instance can be solved in linear time in the number of clauses [6]. □ □

# 5 Computing a Wheeler C-order by iterative refining

The major drawback of the algorithm of Section 4 is its large working space. The explicit construction of 2-SAT clauses will take much more space than the succinct representation of the input/output automata. As discussed in Section 3.5 for de Bruijn graphs, space has a significant practical impact; hence a possible line of future research could be to maintain an implicit representation of the clauses and to devise a memory efficient 2-SAT solver.

In this section we present a different algorithm for computing a Wheeler C-order that takes $\mathcal{O}(|V|^2)$ time and only uses $4|V| + o(|V|)$ bits of working space. Our algorithm however does not always compute a Wheeler C-order for the union automata $\mathcal{U}$. Instead, under the assumption that a Wheeler C-order for $\mathcal{U}$ exists, our algorithm returns a Wheeler automaton $\mathcal{U}'$, possibly different from $\mathcal{U}$, that recognizes the same language as $\mathcal{U}$. The automaton $\mathcal{U}'$ is always smaller than or equal to $\mathcal{U}$ and the algorithm explicitly returns also the ordering that makes $\mathcal{U}'$ a Wheeler automaton. Interestingly it is even possible that our algorithm returns a Wheeler automaton recognizing the union language $\mathcal{L}_0 \cup \mathcal{L}_1$ even if no Wheeler C-order for $\mathcal{U}$ exists. This is a positive feature, but implies that our algorithm is not solving Problem 13, but rather it is providing an (unfortunately) incomplete solution to Problem 11. We will discuss this point in detail at the end of Section 5.1.

To describe our algorithm we introduce some additional notation.

**Definition 16.** Let $V$ denote the set of states of the union automata $\mathcal{U}$. An ordered partition $P_0, P_1, \dots P_k$ of $V$ into disjoint subsets is said to be *W-consistent* if $x \in P_i$, $y \in P_j$ with $i < j$ implies that for any Wheeler C-order it is $x < y$. □

In the above definition it is clear that the ordering of the sets in the partition is fundamental; however for simplicity in the following we usually leave implicit that we are talking about ordered partitions. Because of condition (3a), in a Wheeler graph all edges entering a given node $v$ must have the same label; this observation justifies the following definition.

**Definition 17.** If $v$ is a node in a Wheeler graph with positive in-degree, we denote by $\lambda(v)$ the symbol labelling every edge entering in $v$. □

With the above notation, the simplest example of a W-consistent partition is given by the following lemma.

**Lemma 18.** *Let $P_0 = \{s\}$, and for $i = 1, \dots, \sigma$, let $P_i = \{v \in V | \lambda(v) = i\}$. Then, $P_0, P_1, \dots, P_\sigma$ is a W-consistent partition.*

*Proof.* $P_0, P_1, \dots, P_\sigma$ is a well defined partition since $s$ is the only state with in-degree 0 and $\mathcal{A}_0$, $\mathcal{A}_1$ are Wheeler automaton. The partition is W-consistent because any Wheeler C-order for $\mathcal{U}$ must satisfy property (3a). □ □

The following lemmas illustrate some useful properties of W-consistent partitions.

**Lemma 19.** *Let $P_0, P_1, \dots P_k$ denote a W-consistent partition and let $v, v' \in P_h$ be such that $v \neq v'$ and $\lambda(v) = \lambda(v')$. If there exist two edges $(u, v)$ and $(u', v')$ with $u \in P_i$, $u' \in P_j$, $i < j$, then in any Wheeler C-order we must have $v < v'$.*

*Proof.* By Definition 16 in any Wheeler C-order we must have $u < u'$; the thesis follows by (3b). □ □

```
newP ← {P_0}                                    ▷ Init new partition with P_0
for i ← 1 to k do                               ▷ Consider P_1, ..., P_k
    S_0 ← P_i ∩ V_0                             ▷ Nodes coming from A_0
    S_1 ← P_i ∩ V_1                             ▷ Nodes coming from A_1
    L ← merge(S_0, S_1)                         ▷ Merge according to minmax pairs ordering
    Split L into subsets L_1, ..., L_t with identical minmax pairs.
    newP ← newP ∪ {L_1, ..., L_t}               ▷ Add P_i's refinement to newP
end for
return newP
```

Figure 7: Refinement step for a W-consistent partition $P_0, P_1, \ldots, P_k$. It returns a refined partition newP unless during the merging step two incompatible minmax pairs are found; in this case the algorithm terminates reporting that no Wheeler C-order exists for $\mathcal{U}$.

**Lemma 20.** *Let $P_0, P_1, \ldots P_k$ denote a W-consistent partition and let $v, v' \in P_h$ be such that $v \neq v'$ and $\lambda(v) = \lambda(v')$. Let $\ell$ (resp. $\ell'$) denote the smallest index such that there exists an edge from a node in $P_\ell$ (resp. $P_{\ell'}$) to $v$ (resp. $v'$). Similarly, let $m$ (resp. $m'$) denote the largest index such that there exists an edge from a node in $P_m$ (resp. $P_{m'}$) to $v$ (resp. $v'$). If it is not $\ell = m = \ell' = m'$, then, for any Wheeler C-order we have:*

$$m \leq \ell' \quad \Longrightarrow \quad v < v' \tag{16a}$$

$$m' \leq \ell \quad \Longrightarrow \quad v > v'. \tag{16b}$$

*In addition, if it is not $(m \leq \ell') \vee (m' \leq \ell)$, then a Wheeler C-order for the union automaton cannot exist.*

*Proof.* Consider the case $m \leq \ell'$ ($m' \leq \ell$ is symmetrical). If $m < \ell'$ then $v < v'$ by Lemma 19. If $m = \ell'$, since we are assuming it is not $\ell = m = \ell' = m'$ we must have either $\ell < \ell'$ or $m < m'$ (or both). In all cases the thesis follows again by Lemma 19.

Suppose now that it is not $(m \leq \ell') \vee (m' \leq \ell)$; then we must have $(m > \ell') \wedge (m' > \ell)$. Again by Lemma 19 a Wheeler C-order should satisfy simultaneously $v > v'$ and $v' > v$ which is impossible. □ □

In the following we call the $(\ell, m)$ pair defined in Lemma 20 a *minmax* pair.

**Definition 21.** We say that two minmax pairs $(\ell, m)$ and $(\ell', m')$ are *compatible* if $(m \leq \ell') \vee (m' \leq \ell)$. □

With the above definition, we can rephrase the second half of Lemma 20 saying that if the minmax pairs $(\ell, m)$ and $(\ell', m')$ are not compatible then the union automaton does not have a Wheeler C-order.

Given two *compatible* minmax pairs $(\ell, m)$ and $(\ell', m')$ if $m \leq \ell'$ we write

$$(\ell, m) \preceq (\ell', m'). \tag{17}$$

It is easy to see that if $(\ell, m)$ and $(\ell', m')$ are compatible then it is either $(\ell, m) \preceq (\ell', m')$ or $(\ell', m') \preceq (\ell, m)$ and both relations are true simultaneously if and only if $\ell = m = \ell' = m'$. Also note that the relation $\preceq$ is transitive in the sense that if $(\ell, m) \preceq (\ell', m')$ and $(\ell', m') \preceq (\ell'', m'')$ then $(\ell, m)$ is compatible with $(\ell'', m'')$ and $(\ell, m) \preceq (\ell'', m'')$.

## 5.1 The iterative refining algorithm

Our strategy consists in starting with the W-consistent partition defined in Lemma 18 and refining it iteratively obtaining finer W-consistent partitions. Refining a partition $P_0, P_1, \ldots, P_k$ here means that each set $P_i$ is partitioned into $P_{i,0}, P_{i,1}, \ldots, P_{i,t_i}$ so that at the next iteration the working partition becomes

$$P_{0,0}, \ldots, P_{0,t_0}, P_{1,0}, \ldots, P_{1,t_1} \ \ldots \ P_{i,0}, \ldots, P_{i,t_i} \ \ldots \ P_{k,0}, \ldots, P_{k,t_k}.$$

To simplify the notation we use $P_0, P_1, \ldots P_k$ to denote the current partition assuming that after each refinement step the sets are properly renumbered.

At each iteration the refinement is done with the algorithm outlined in Fig. 7 that produces a finer partition or reports that no Wheeler C-order exists. We first observe that since we start with the partition given by Lemma 18, at any time during the algorithm each set $P_h$ is a subset of some $\{v \in V | \lambda(v) = i\}$, so we can apply Lemmas 19 and 20 to any pair of distinct nodes $v, v' \in P_h$.

In the main loop of the algorithm in Fig. 7, given a set $P_i$ we define $S_0 = P_i \cap V_0$ and $S_1 = P_i \cap V_1$. Since $\mathcal{A}_0$ (resp. $\mathcal{A}_1$) is a Wheeler graph the elements in $S_0$ (resp. $S_1$) are all pairwise compatible and they are already ordered according to relation $\preceq$ defined by (17). Next we merge $S_0$ and $S_1$ according to $\preceq$. We start with an empty result list $L$ and we compare the nodes currently at the top of $S_0$ and $S_1$, say $v \in S_0$ and $v' \in S_1$. If the corresponding minmax pairs $(\ell, m)$ and $(\ell', m')$ are not compatible the whole algorithm fails (no Wheeler C-order exists by Lemma 20). Assuming the pairs are compatible, if $(\ell, m) \preceq (\ell', m')$ we remove $v$ from $S_0$ and append it to $L$; otherwise we remove $v'$ from the $S_1$ and append it to $L$. In either case, we then continue the merging of the elements still in $S_0$ and $S_1$. At the end of the merging procedure all elements in $P_i$ are in $L$ ordered according to the $\preceq$ relation. We refine $P_i$ splitting $L$ into (maximal) subsets $L_1, L_2, \ldots, L_t$ so that all elements in the same subset $L_k$ have identical minmax pairs $(\ell, m)$. In other words, if $v \in L_j$ and $v' \in L_k$, with $j < k$, the corresponding pairs $(\ell, m)$ and $(\ell', m')$ are such that $(\ell, m) \preceq (\ell', m')$ but it is not $\ell = m = \ell' = m'$. By (16a) this implies that in any Wheeler C-order we must have $v < v'$ and this ensures that if we split $P_i$ into the subsets $L_1, L_2, \ldots, L_t$ the resulting partition is still W-consistent. Repeating the above procedure for every set $P_i$ we end up with either a refined partition or the indication that the union automaton $\mathcal{U}$ has no Wheeler C-order. In the former case it is also possible that the new partition is identical to the current one: this happens when for $i = 1, \ldots, k$ at line 6 it is $t = 1$ since all nodes in $L$ have the same minmax pairs. In this case no further iteration will modify the partition so we stop the refinement phase.

Fig. 8 shows an example of a refinement step. $P_1$ on the left is refined yielding $P_1 - P_3$ on the right, and $P_2$ on the left is refined yielding $P_4 - P_6$ on the right. All other $P_i$'s are unchanged. The partition on the right cannot be further refined.

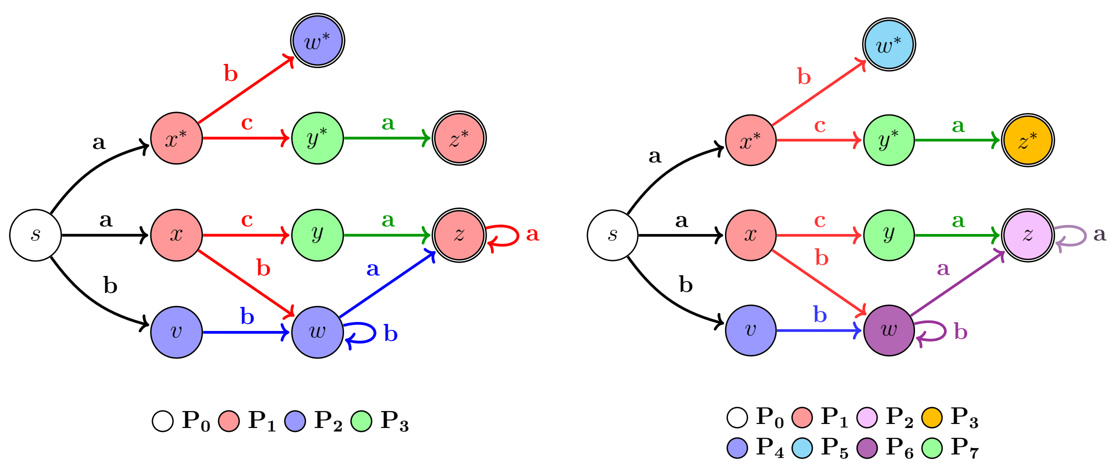



Figure 8: Example of a refinement step for the union automaton displayed above obtained from the Wheeler automata $\mathcal{A}_0$ and $\mathcal{A}_1$, with $V_0 = \{s, x^*, y^*, w^*, z^*\}$ and $V_1 = \{s, x, y, v, w, z\}$ On the left, nodes are colored to highlight the initial partition from Lemma 18: $P_0 = \{s\}$, $P_1 = \{x^*, z^*, x, z\}$, $P_2 = \{w^*, v, w\}$, $P_3 = \{y^*, y\}$. There are edges going from $P_0$ to $P_1$ and $P_2$, from $P_1$ to $P_2$ and $P_3$, from $P_2$ to $P_1$ and $P_2$, and from $P_3$ to $P_1$ (colors in the figure highlight the partition set from which edges originate). As a result, at the end of the refinement step, the new partition is $P_0 = \{s\}$, $P_1 = \{x^*, x\}$, $P_2 = \{z\}$, $P_3 = \{z^*\}$, $P_4 = \{v\}$, $P_5 = \{w^*\}$ $P_6 = \{w\}$ $P_7 = \{y^*, y\}$, as shown on the right side. An additional refinement step does not further modify the partition since all sets are either singletons or all their nodes have the same minmax pairs.

**Bounding the number of iterations.** Recall we are assuming that each node in the union automata is reachable from the initial state $s$. For each node $v \in V$ we define $d_v$ as the length of the *shortest path* from $s$ to $v$. Let then

$$\delta = \max_{v \in V} d_v \tag{18}$$

be the maximum distance from $s$ to nodes in $V$ In the following we show that, unless our algorithm reports that there is no Wheeler C-order, after at most $\delta + 2$ refinement iterations we reach a W-consistent partition that is not further refined by the algorithm in Fig. 7.

**Lemma 22.** *If, at the beginning of the $j$-th iteration, a partition element $P_i$ is not a singleton then either: $(i)$ for every $v \in P_i$ all paths from $s$ to $v$ have length at least $j$ or $(ii)$ there exists $j' < j$ such that for every $v \in P_i$ all paths from $s$ to $v$ have length $j'$.*

*Proof.* We prove the result by induction on $j$. For $j = 1$, immediately before the first iteration for $i > 1$ each $P_i$ is defined as $P_i = \{v|\lambda(v) = i\}$ so they all satisfy property $(i)$. For $j > 1$, let $P_0, \ldots, P_k$ denote the current partition at the beginning of the $(j-1)$-st iteration. During the refinement step each set $P_i$ is split into subsets $L_1, L_2, \ldots, L_t$ as described above. Each subset $L_h$ will become a partition element for the $j$-th iteration, so to prove the lemma we need to show that $L_h$ satisfies $(i)$ or $(ii)$. By construction, if $L_h$ is not a singleton then all nodes $v \in L_h$ have the same minmax pair $(\ell, m)$ with $\ell = m$. It follows that *all* edges reaching the nodes in $L_h$ must originate by the same set $P_\ell$. If $\ell = 0$ then $L_h$ satisfies property $(ii)$ with $j' = 1$. If $\ell > 0$, by induction $P_\ell$ must satisfy either $(i)$ or $(ii)$. If $P_\ell$ satisfies $(i)$ and all paths from $s$ to $P_\ell$ have length at least $j-1$, then $L_h$ also satisfies $(i)$ with paths of length at least $j$; if $P_\ell$ satisfies $(ii)$ with length $j'$, then $L_h$ satisfies $(ii)$ with length $j' + 1$. □ □

**Lemma 23.** *If, at the beginning of the $j$-th iteration, a partition element $P_i$ is not a singleton and satisfies property $(ii)$ of Lemma 22, then it will not be split by all subsequent refinement steps.*

*Proof.* We prove the result by induction on $j$. For $j = 1$ there cannot be any $P_i$ satisfying property $(ii)$ of Lemma 22. To prove the lemma for $j = 2$ we observe that, by the proof of Lemma 22, during the first iteration a set satisfying property $(ii)$ is generated only by a subset $L_h$ containing only nodes with minmax pair $(0, 0)$ (that is, nodes only reachable from the source in one step). In any further refinement step the minmax pair of each node will still be $(0, 0)$ so the set will not be further modified.

For $j > 2$ we observe again that, by the proof of Lemma 22, in any subsequent iteration a new partition element satisfying property $(ii)$ is generated only when all nodes in a subset $L_h$ have the same minmax pair $(\ell, \ell)$ and $P_\ell$ is a set already satisfying property $(ii)$. By inductive hypothesis the set $P_\ell$ will not be further split in subsequent iterations; hence the nodes in $L_h$ will still have identical minmax pairs $(\ell', \ell')$ in subsequent iterations and the subset $L_h$ will not be further split. □ □

We use Lemmas 22 and 23 to bound the number of refinement steps in terms of the maximum distance (18):

**Lemma 24.** *Let $\delta$ be as defined in (18). After at most $\delta + 2$ refinement iterations either the algorithm has reported that a Wheeler C-order does not exist, or it has computed a W-consistent partition that the algorithm could not refine.*

*Proof.* After $\delta + 1$ refinement iterations there cannot be a partition element $P_i$ satisfying property $(i)$ of Lemma 22 since $\delta$ is the maximum distance of any node from $s$. Hence, after at most $\delta + 1$ iterations all partition elements $P_i$ are either singletons or satisfy property $(ii)$ of Lemma 22; by Lemma 23 none of them will be refined in subsequent iterations. □ □

**Construction of the Wheeler Automaton for the union language.** When the partition $P_0, P_1, \ldots, P_k$ cannot be further refined we proceed building the output automaton. One can see that if all sets $P_i$ are singleton, then the partition ordering is a Wheeler C-order for $\mathcal{U}$. In the general, case in which some $P_i$ is not a singleton, we use the partition to build a (smaller) Wheeler automaton that also recognizes the union language.

**Definition 25.** Let $\mathcal{U} = (V, E, F, s, \Sigma)$ be the union automaton and $\{P_0, P_1, \ldots, P_k\}$ a W-consistent partition of its nodes that cannot be further refined. We define the automaton $\mathcal{U}' = (V', E', F', s, \Sigma)$ with $V' = \{P_0, P_1, \ldots, P_k\}$, $s = P_0$, $(P_i, P_j, a) \in E'$ iff $(v, v', a) \in E$ for some $v \in P_i$ and $v' \in P_j$, and $P_i \in F'$ iff $P_i \cap F \neq \emptyset$. □

The automaton is well defined, since edges incoming in each $P_i$ have the same label in $\mathcal{U}$ and therefore there is no ambiguity in the definition of edge labels in $\mathcal{U}'$.

**Lemma 26.** *$\mathcal{U}'$ recognizes the same language as $\mathcal{U}$.*

*Proof.* To prove the lemma we preliminary observe that if $v, v' \in P_i$ are both nodes from the same input automaton, say $\mathcal{A}_0$, then they are equivalent in the sense that if there is in $\mathcal{A}_0$ a path with label $\alpha$ from $s$ to $v$, there is in $\mathcal{A}_0$ also a path with the same label from $s$ to $v'$, and vice versa. Therefore we can safely assume that each $P_i$ is either singleton or contains a node from $V_0$ and a node from $V_1$; for simplicity here we call these nodes *special nodes*.

Since the partition $\{P_0, P_1, \ldots, P_k\}$ cannot be further refined we get that in $\mathcal{U}$ all edges entering into a $P_h$ which is not a singleton originate from a single set $P_{h'}$, which therefore either contains nodes from both $V_0$ and $V_1$ or contains the node $s$ (when $h' = 0$). In $\mathcal{U}'$ this implies that each special node is reachable by a single node which is either $s$ or a special node itself. Hence, $\mathcal{U}'$'s subgraph containing the special nodes and the source $s$ is a tree rooted at $s$.

Consider now the word $\alpha$ corresponding to a path in $\mathcal{U}'$ going from $s$ to a final node $f \in F'$. If $f$ is a special node all nodes in the path are special. By construction, the partition element $P_f$ in $\mathcal{U}$ corresponding to $f$ in $\mathcal{U}'$ must contain a node from either $F_0$ or $F_1$ (or both), hence a path labeled $\alpha$ is contained in either $\mathcal{A}_0$ or $\mathcal{A}_1$ and is therefore contained in the union language. If $f$ is not a special node, then a proper prefix of the path from $s$ to $f$ consists of special nodes, while $P_f \cap F_0 \neq \emptyset$ or $P_f \cap F_1 \neq \emptyset$, but not both. Assuming for example $P_f \cap F_0 \neq \emptyset$, then we can find a path in $\mathcal{A}_0$ labeled $\alpha$ which will therefore belong to the union language.

Finally, if the word $\beta$ corresponds to a path in $\mathcal{U}$ from $s$ to a final node, by possibly mapping some nodes in $\mathcal{U}$ to the corresponding special nodes in $\mathcal{U}'$ we get a path labeled $\beta$ in $\mathcal{U}'$. □ □

**Theorem 27.** *$\mathcal{U}'$ is a Wheeler automaton recognizing the union language $\mathcal{L}_0 \cup \mathcal{L}_1$.*

*Proof.* By Lemma 26, we only need to prove that there is an ordering of the nodes of $\mathcal{U}'$ that satisfies the Wheeler conditions.

Consider the ordering $P_0 < P_1 < \cdots < P_k$ and assume by contradiction that this ordering does not satisfy the Wheeler conditions. Condition (3a) is verified by construction by the initial partition (the one from Lemma 18), and all subsequent refinements never change the relative order of existing partition elements, since they are just split in smaller subsets. If condition (3b) is not verified the graph has edges $(P_i, P_h, a)$ and $(P_j, P_\ell, a)$ such that $P_j > P_i$ and $P_h > P_\ell$. However, we notice that at the beginning of the refinement algorithm the nodes in $P_h$ and $P_\ell$ were in the same partition element $\{v | \lambda(v) = a\}$. At the iteration $\tau$ in which these nodes ended in different sets, the nodes in $P_\ell$ had a minmax pair smaller, according to (17), than the nodes in $P_h$. This implies that no edge reaching $P_\ell$ originated from a partition element *following* the partition elements originating the edges reaching $P_h$. This was true at iteration $\tau$ but since the refinement process splits partition elements but never changes their relative order it is impossible that $P_j > P_i$. □ □

Consider again the example of Fig. 8. The partition $P_0 = \{s\}$, $P_1 = \{x^*, x\}$, $P_2 = \{z\}$, $P_3 = \{z^*\}$, $P_4 = \{v\}$, $P_5 = \{w^*\}$ $P_6 = \{w\}$ $P_7 = \{y^*, y\}$ (right side of Fig. 8 and left side of Fig. 9) cannot be further refined. Applying our procedure we obtain the automaton at the right of Fig. 9 which is a Wheeler automaton with the ordering $P_0 < P_1 < \ldots < P_7$.

Note that the union automaton shown in the left of Fig. 9 *is not* a Wheeler automaton: condition (3b) applied to edges $(y^*, z^*)$ and $(y, z)$ together with $z < z^*$ implies $y < y^*$ which in turns implies $x < x^*$ which clashes with $w^* < w$ and the edges $(x^*, w^*)$ and $(x, w)$. This is therefore an instance of a problem where $\mathcal{U}$ does not have a Wheeler C-order but our algorithm still returns a Wheeler automaton.

Unfortunately, our algorithm can return a reduced automaton $\mathcal{U}' \neq \mathcal{U}$ even when $\mathcal{U}$ does have a Wheeler C-order, as shown in the example of Fig. 10 (top). Although a smaller automaton is

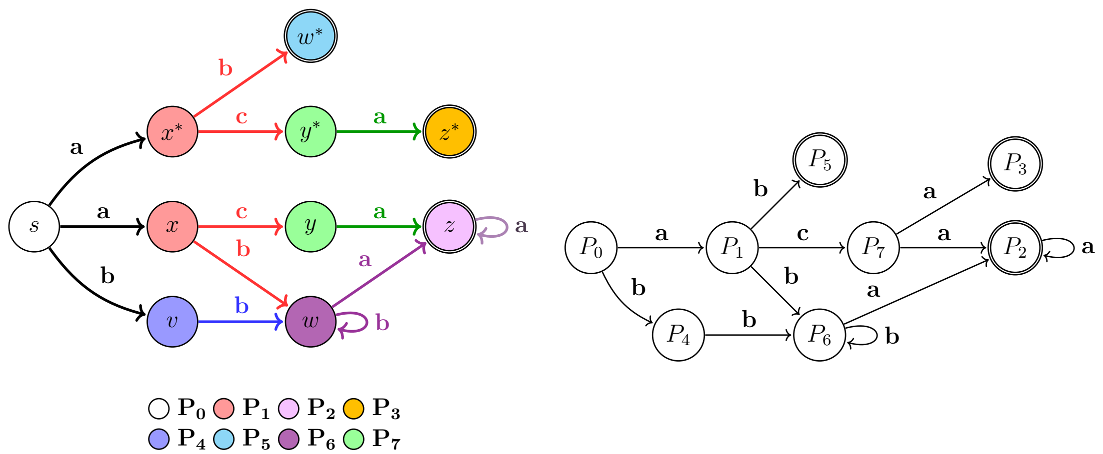



Figure 9: The Wheeler automaton $\mathcal{U}'$ obtained from the union graph of Fig. 8. To the left the final partition that cannot be further refined and to the right the corresponding Wheeler automaton defined according to Definition 25.

preferable for the construction of succinct data structures, this example implies that our algorithm does not provide a complete solution to Problem 13 since, in case $\mathcal{U}' \neq \mathcal{U}$, we cannot say whether a Wheeler C-order for $\mathcal{U}$ exists.

With regard to Problem 11, although our algorithm is correct, in that it returns a Wheeler automaton for the union language, it is not complete since it fails to do so in some cases in which a Wheeler automaton for the union language exists. This is shown in the example of Fig. 10 (bottom) which uses the same automata as in Fig. 6. After computing the partition $P_0$, $P_1$, $P_2$ for the nodes of the union automaton, our algorithm reports that it has no Wheeler C-order, since the minmax pairs for $v$ and $v^*$ in partition $P_1$ are both $(0,1)$ and thus, by Definition 21, are not compatible. However, as noticed in Fig. 6, a Wheeler automaton for the union language exists and can be obtained by the automaton on the lower left and making $v^*$ a final state. Noticing that in Fig. 10 **(b2)** $v$ and $v^*$ belong to the same partition $P_1$, this example suggests that a possible strategy for improving our algorithm could be to analyze the union automaton and eliminate redundant nodes before starting the refinement steps.

## 5.2 Implementation details and analysis

Let $n_0 = |V_0|$, $m_0 = |E_0|$, $n_1 = |V_1|$, $m_1 = |E_1|$. We assume the iterative refining algorithm described in this section takes as input the compact representation of the graphs $G_0$ and $G_1$ supporting constant time navigation operations (see Section 2.2).

The main body of the algorithm is the refinement phase in which the initial W-consistent partition given by Lemma 18 is progressively refined. Let $P_0, P_1, \ldots, P_k$ denote the current partition. We know that $P_0 = \{s\}$ and does not change during the algorithm; for simplicity in our implementation we represent $P_0$ as $\{s_0, s_1\}$. We represent $P_0, P_1, \ldots, P_k$ with two binary arrays $B$ and $Z$ both of length $n_0 + n_1$. The array $B[1, n_0 + n_1]$ encodes the size of the sets: it has exactly $k+1$ **1**s in positions: $|P_0|$, $|P_0| + |P_1|$, ..., $|P_0| + |P_1| + \cdots + |P_k|$. With this encoding for example the size of $P_i$ is given by $\mathsf{select}_1(B, i+1) - \mathsf{select}_1(B, i)$, where we assume $\mathsf{select}_1(B, 0) = 0$. The array $Z$ encodes the content of each set $P_i$: we logically partition it into $k+1$ subarrays, the $i$-th subarray has length $|P_i|$ and consists of $|P_i \cap V_0|$ **0**'s followed by $|P_i \cap V_1|$ **1**'s. For example, for the initial partition mentioned in the caption of Fig. 8, recalling that we represent $P_0$ with $\{s_0, s_1\}$ it is

$$B = \mathbf{01\,0001\,001\,01}$$
$$Z = \mathbf{01\,0011\,011\,01}$$

(spaces have been added for readability). The above arrays highlight the similarities between the

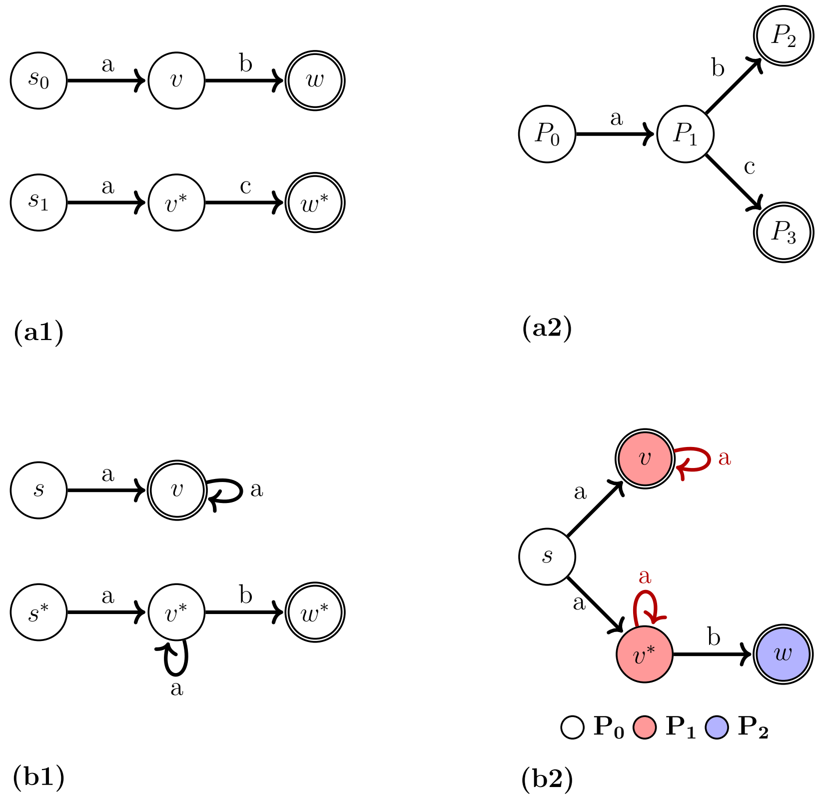



Figure 10: **Top:** Given the automata on the left **(a1)** the algorithm of Fig. 7 returns the automaton on the right **(a2)**, having collapsed the nodes $v$ and $v^*$ to a single node $P_1$. However, the union automaton itself has a Wheeler C-order, for example $s < v < v^* < w < w^*$. **Bottom:** The union of the two Wheeler automata on the left **(b1)** is the automaton on the right **(b2)**. After computing the partition $P_0$, $P_1$, $P_2$, our algorithm correctly reports that the union automaton does not have a Wheeler C-order and stops. However, a Wheeler automaton for the union language could be obtained for example making $v^*$ a final state in the automaton on the lower left.

refining algorithm and the merging algorithm for de Bruijn graphs. The array $Z$ corresponds to the arrays $Z^{(h)}$ used in Section 3.1 to denote status of the merging of the de Bruijn graphs nodes, and the array $B$ is analogous to the integer array, also called $B$, used in the algorithm in Fig. 5 to mark block boundaries. Both algorithms compute the merging of the input nodes by iteratively partitioning the very same nodes they are merging.

Assuming we enrich $B$ and $Z$ with auxiliary data structures to support constant time $\mathsf{rank}$ and $\mathsf{select}$ operations, a single refinement operation (Lines 3–7 in Fig. 7) is implemented as follows. Setting $b_i = \mathsf{select}_1(B, i-1)$, we compute the starting position in $Z$ of the subarray corresponding to $P_i$ and its length $|P_i| = \mathsf{select}_1(B, i) - b_i$. Then we compute $|S_0| = \mathsf{rank}_0(Z, b_i + |P_i|) - \mathsf{rank}_0(Z, b_i)$ and $|S_1| = |P_i| - |S_0|$ (recall that $S_0 = P_i \cap V_0$ and $S_1 = P_i \cap V_1$).

For the merging operation at Line 5 in Fig. 7 we need to compute the minmax pair for each node in $S_0$ and $S_1$. Consider for example the $j$-th node in $S_0$ (for $S_1$ it is analogous). Setting $c_j = j + \mathsf{rank}_0(Z, b_i)$ we compute the rank of such node in $G_0$; using $G_0$'s succinct representation we compute the largest and smallest nodes, say $\alpha_j$ and $\beta_j$, such that the edges $(\alpha_j, c_j)$ and $(\beta_j, c_j)$ belong to $E_0$ (the computation of $\alpha_j$ is the one outlined just before Lemma 2). The minmax pair $(\ell, m)$ coincides with the ids of the blocks containing $\alpha_j$ and $\beta_j$, which are given respectively by

$$\ell = \mathsf{rank}_1(B, \mathsf{select}_0(Z, \alpha_j)), \qquad m = \mathsf{rank}_1(B, \mathsf{select}_0(Z, \beta_j)).$$

Note that the minmax pairs are computed on the spot for the elements that are compared by the merging algorithm so they require only constant storage. The output of the merging is stored into another binary array $Z'[1, n_0 + n_1]$. The output relative to $P_i$ is written to $Z'$ from position $b_i + 1$ up to position $b_i + |P_i|$: if the $k$-th element computed by the merging algorithm comes from $S_0$ we set $Z'[b_j + k] = \mathbf{0}$ otherwise we set $Z'[b_j + k] = \mathbf{1}$.

Finally, the splitting of $P_i$ (Line 6) and the update of the current partition (Line 7) is done using another array $B'[1, n_0 + n_1]$. For $k = 1, \ldots, |Bx_i| - 1$ if the minmax pair corresponding to $Z'[b_j + k]$ is different from the one corresponding to $Z'[b_j + k + 1]$ we set $B'[b_j + k] = \mathbf{1}$, otherwise (the minmax pairs are equal) we set $B'[b_j + k] = \mathbf{0}$. We conclude the processing of $P_i$ setting $B'[b_j + |P_i|] = \mathbf{1}$. It is immediate to see that at the end of the algorithm of Fig. 7 the binary arrays $B'$ and $Z'$ provide the encoding of the new partition. If they are equal to $B$ and $Z$ then no further refinements are possible and we proceed with the construction of $\mathcal{U}'$, otherwise we replace $B$ and $Z$ with $B'$ and $Z'$, build the auxiliary data structures supporting rank and select, and proceed with the next iteration. Summing up, the algorithm in Fig. 7 takes $\mathcal{O}(n_0 + n_1)$ time and uses $4(n_0 + n_1) + o(n_0 + n_1)$ bits of working space. Since the number of refinement steps is at most $\delta + 2 = \mathcal{O}(|V|)$ the total time is $\mathcal{O}(|V|^2)$ and the working space is $4|V| + o(|V|)$ bits.

Given the arrays $B$ and $Z$ from the last refinement steps, the succinct representation of the automata $\mathcal{U}'$ (i.e. the arrays $L$, $I$, and $O$) can be easily computed in $\mathcal{O}(|V| + |E|)$ time without using additional working space. We can therefore summarize our result as follows.

**Theorem 28.** *Given the succinct representation of the Wheeler automata $\mathcal{A}_0$ and $\mathcal{A}_1$, our algorithm either reports that the union automaton $\mathcal{U}$ has no Wheeler C-order or returns a Wheeler automaton $\mathcal{U}'$ recognizing the same language as $\mathcal{U}$. Our algorithm takes $\mathcal{O}(|V|^2)$ time and uses $4|V| + o(|V|)$ bits of working space.* □

## Acknowledgments

The authors thank some anonymous reviewers for their comments that improved the quality of manuscript, and for pointing out some inconsistencies in some passages of the original draft.

**Funding.** L.E. and G.M. were partially supported by PRIN grant 2017WR7SHH and by the INdAM-GNCS Project 2020 *MFAIS-IoT*. L.E. was partially supported by the University of Eastern Piedmont project *HySecEn*. F.A.L. was supported by the grants #2017/09105-0 and #2018/21509-2 from the São Paulo Research Foundation (FAPESP).

## References

[1] Alanko, J., D'Agostino, G., Policriti, A., Prezza, N.: Regular languages meet prefix sorting. In: SODA. pp. 911–930. SIAM (2020)

[2] Alanko, J., D'Agostino, G., Policriti, A., Prezza, N.: Wheeler languages. CoRR **2002.10303** (2020), `https://arxiv.org/abs/2002.10303`

[3] Alanko, J.N., Gagie, T., Navarro, G., Seelbach Benkner, L.: Tunneling on wheeler graphs. In: DCC. pp. 122–131. IEEE (2019)

[4] Alipanahi, B., Kuhnle, A., Boucher, C.: Recoloring the colored de Bruijn graph. In: SPIRE. LNCS, vol. 11147, pp. 1–11. Springer (2018)

[5] 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)

[6] Aspvall, B., Plass, M.F., Tarjan, R.E.: A linear-time algorithm for testing the truth of certain quantified boolean formulas. Inf. Process. Lett. **8**(3), 121–123 (1979)

[7] Baier, U., Dede, K.: BWT tunnel planning is hard but manageable. In: DCC. pp. 142–151. IEEE (2019)

[8] Belazzougui, D., Navarro, G.: Optimal lower and upper bounds for representing sequences. ACM T. Algorithms **11**(4), 31:1–31:21 (2015)

[9] Belazzougui, D., Cunial, F.: Fully-functional bidirectional Burrows-Wheeler indexes and infinite-order de Bruijn graphs. In: CPM. LIPIcs, vol. 128, pp. 10:1–10:15. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2019)

[10] 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)

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

[12] Bowe, A., Onodera, T., Sadakane, K., Shibuya, T.: Succinct de Bruijn graphs. In: WABI. LNCS, vol. 7534, pp. 225–235. Springer (2012)

[13] Burrows, M., Wheeler, D.J.: A block-sorting lossless data compression algorithm. Tech. rep., Digital SRC Research Report (1994)

[14] Chikhi, R., Rizk, G.: Space-efficient and exact de Bruijn graph representation based on a bloom filter. Algorithms for Molecular Biology **8**, 22 (2013)

[15] Cotumaccio, N., Prezza, N.: On indexing and compressing finite automata. In: SODA. SIAM (2021)

[16] Durbin, R.: Efficient haplotype matching and storage using the positional Burrows-Wheeler transform (PBWT). Bioinformatics **30**(9), 1266–1272 (2014)

[17] Egidi, L., Louza, F.A., Manzini, G.: Space-efficient merging of succinct de Bruijn graphs. In: SPIRE. LNCS, vol. 11811, pp. 337–351. Springer (2019)

[18] 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)

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

[20] Egidi, L., Manzini, G.: Lightweight BWT and LCP merging via the Gap algorithm. In: SPIRE. LNCS, vol. 10508, pp. 176–190. Springer (2017)

[21] Egidi, L., Manzini, G.: Lightweight merging of compressed indices based on BWT variants. Theor. Comput. Sci. **812**, 214–229 (2020)

[22] Ferragina, P., Gagie, T., Manzini, G.: Lightweight data indexing and compression in external memory. Algorithmica (2011)

[23] Ferragina, P., Manzini, G.: Indexing compressed text. Journal of the ACM **52**(4), 552–581 (2005)

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

[25] Ferragina, P., Luccio, F., Manzini, G., Muthukrishnan, S.: Compressing and indexing labeled trees, with applications. J. ACM **57** (2009)

[26] Ferragina, P., Venturini, R.: The compressed permuterm index. ACM Trans. Algorithms **7**(1), 10:1–10:21 (2010)

[27] Gagie, T., Gourdel, G., Manzini, G.: Compressing and indexing aligned readsets. In: WABI. LIPIcs, vol. 201. Schloss Dagstuhl — Leibniz-Zentrum fuer Informatik (2021)

[28] Gagie, T., Manzini, G., Sirén, J.: Wheeler graphs: A framework for BWT-based data structures. Theor. Comput. Sci. **698**, 67–78 (2017)

[29] Gibney, D., Thankachan, S.V.: On the Hardness and Inapproximability of Recognizing Wheeler Graphs. In: ESA. LIPIcs, vol. 144, pp. 51:1–51:16. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2019)

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

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

[32] Iqbal, Z., Caccamo, M., Turner, I., Flicek, P., McVean, G.: De novo assembly and genotyping of variants using colored de Bruijn graphs. Nature Genetics **44**(2), 226–232 (2012)

[33] Kärkkäinen, J., Kempa, D.: Engineering a lightweight external memory suffix array construction algorithm. Mathematics in Computer Science **11**(2), 137–149 (2017)

[34] Mantaci, S., Restivo, A., Rosone, G., Sciortino, M.: An extension of the Burrows-Wheeler transform. Theor. Comput. Sci. **387**(3), 298–312 (2007)

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

[36] Muggli, M.D., Alipanahi, B., Boucher, C.: Building large updatable colored de Bruijn graphs via merging. Bioinformatics **35**(14), i51–i60 (2019)

[37] 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, `v1`

[38] Muggli, M.D., Bowe, A., Noyes, N.R., Morley, P.S., Belk, K.E., Raymond, R., Gagie, T., Puglisi, S.J., Boucher, C.: Succinct colored de Bruijn graphs. Bioinformatics **33**(20), 3181–3187 (2017)

[39] Na, J.C., Kim, H., Min, S., Park, H., Lecroq, T., Léonard, M., Mouchard, L., Park, K.: FM-index of alignment with gaps. Theor. Comput. Sci. **710**, 148–157 (2018)

[40] 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)

[41] 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)

[42] Sirén, J.: Burrows-Wheeler transform for Terabases. In: DCC. pp. 211–220. IEEE (2016)

[43] Sirén, J.: Indexing variation graphs. In: ALENEX. pp. 13–27. SIAM (2017)