Bioinformatics
doi.10.1093/bioinformatics/xxxxxx
Advance Access Publication Date: Day Month Year
Manuscript Category

Subject Section

Building Large Updatable Colored de Bruijn Graphs via Merging

Martin D. Muggli [1,*], Bahar Alipanahi [2] and Christina Boucher [2] *

[1]Department of Computer Science, Colorado State University, Fort Collins, CO, 80526
[2]Department of Computer and Information Science and Engineering, University of Florida, Gainesville, FL 32607

*To whom correspondence should be addressed.

Associate Editor: XXXXXXX

Received on XXXXX; revised on XXXXX; accepted on XXXXX

Abstract

Motivation: There exist several large genomic and metagenomic data collection efforts, including GenomeTrakr and MetaSub, which are routinely updated with new data. To analyze such datasets, memory-efficient methods to construct and store the colored de Bruijn graph were developed. Yet, a problem that has not been considered is constructing the colored de Bruijn graph in a scalable manner that allows new data to be added without reconstruction. This problem is important for large public datasets as scalability is needed but also the ability to update the construction is also needed.
Results: We create a method for constructing the colored de Bruijn graph for large datasets that is based on partitioning the data into smaller datasets, building the colored de Bruijn graph using a FM-index based representation, and succinctly merging these representations to build a single graph. The last step, merging succinctly, is the algorithmic challenge which we solve in this paper. We refer to the resulting method as VariMerge. This construction method also allows the graph to be updated with new data. We validate our approach and show it produces a three-fold reduction in working space when constructing a colored de Bruijn graph for 8,000 strains. Lastly, we compare VariMerge to other competing methods — including Vari (Muggli *et al.*, 2017), Rainbowfish (Almodaresi *et al.*, 2017), Mantis (Pandey *et al.*, 2018), Bloom Filter Trie (Holley *et al.*, 2016), the method by Almodaresi *et al.* (2019) and Multi-BRWT (Karasikov *et al.*, 2019) — and illustrate that VariMerge is the only method that is capable of building the colored de Bruijn graph for 16,000 strains in a manner that allows it to be updated. Competing methods either did not scale to this large of a dataset or cannot allow for additions without reconstruction.
Availability: VariMerge is at `https://github.com/cosmo-team/cosmo/tree/VARI-merge` under GPLv3 license.
Contact: Martin D. Muggli martin.muggli@colostate.edu

1 Introduction

The money and time needed to sequence a genome have decreased remarkably in the past decade. With this decrease has come an increase in the number and rate at which sequence data is collected for public sequencing projects. This led to the existence of GenomeTrakr, which is a large public effort to use genome sequencing for surveillance and detection of outbreaks of foodborne illnesses. This effort includes over 50,000 samples, spanning several species available through this initiative — a number that continues to rise as datasets are continually added (Stevens *et al.*, 2017). Another example is illustrated by the sequencing of the human genome. The 1000 Genomes Project (The 1000 Genomes Project Consortium, 2015) was announced in 2008 and completed in 2015, and now the 100,000 Genomes Project is well underway (Turnbull *et al.*, 2018). Unfortunately, methods to analyze these and other large public datasets are limited due to their size.

Iqbal *et al.* (2012) presented one method for analysis of sequence data from large populations (such as the ones described above) which focuses on the construction of the *colored de Bruijn graph*. To define the colored de Bruijn graph, we first define the traditional de Bruijn graph and then

*to whom correspondence should be addressed show how it can be extended. Formally, a de Bruijn graph constructed for a set of strings (e.g., sequence reads) has a distinct vertex v for every unique $(k-1)$-mer (substring of length $k-1$) present in the strings, and a directed edge (u, v) for every observed k-mer in the strings with $(k-1)$-mer prefix u and $(k-1)$-mer suffix v. In the colored de Bruijn graph, the edge structure is the same as the classic structure, but now to each node ($(k-1)$-mer) and edge (k-mer) is associated a list of colors corresponding to the samples in which the node or edge label exists. More specifically, given a set of n samples, there exists a set $\mathcal{C}$ of n colors $c_1, c_2, .., c_n$ where c_i corresponds to sample i and all k-mers and $(k-1)$-mers that are contained in sample i are colored with c_i. A *bubble* in this graph corresponds to an undirected cycle and is shown to be indicative of biological variation. Iqbal *et al.* (2012) show after constructing a colored de Bruijn graph, they can recover all genetic variants in the underlying population by traversing the graph, and finding paths with bubbles ("bubble finding").

A bottleneck in applying Cortex (Iqbal *et al.*, 2012) to large datasets lies in the amount of memory and CPU time required to build and store the colored de Bruijn graph. Thus, Vari (Muggli *et al.*, 2017), Rainbowfish (Almodaresi *et al.*, 2017), Mantis (Almodaresi *et al.*, 2019), the method of Karasikov *et al.* (2019), as well as others, sought to overcome this limitation by building and/or storing the colored de Bruijn graphs in a space- and time- efficient manner. Vari was one of the first methods to build the colored de Bruijn graph in a memory efficient manner. It extends the de Bruijn graph construction of Bowe *et al.* (2012). Rainbowfish (Almodaresi *et al.*, 2017) was later developed. It uses Vari to build the colored de Bruijn graph, which it further compresses by storing identical rows in the Vari color matrix as a single row. Cortex, Vari and Rainbowfish have bubble-calling methods that allow genetic variation between the datasets to be detected.

Mantis (Pandey *et al.*, 2018) improves on Vari and Rainbowfish by constructing a different data structure which is not reliant on Vari. Most recently, the method of Almodaresi *et al.* (2019) was presented, which is referred to as an MST-based color-class representation. This method first uses Mantis to build the colored de Bruijn graph, which it then compresses by building minimum spanning trees of the underlying graph and storing only deltas between similar vectors. Both Mantis and the method of Almodaresi *et al.* (2019) do not have bubble-calling procedures. We note there exists more colored de Bruijn graph construction algorithms, and give a comprehensive review in the next section.

Unfortunately, there does not exist a method to build the colored de Bruijn graph in a manner that is scalable to large datasets and allows for the addition of new data. This is important since one of the original purposes of the colored de Bruijn graph was to analyze population-level datasets — such as GenomeTrakr and 100,000 Genomes Project — which continually grow in size due to the addition of new data. Existing colored de Bruijn graph methods either cannot scale well enough to construct their data structure for large datasets or cannot update the data structure without complete reconstruction. For example, Vari (Muggli *et al.*, 2017), Rainbowfish (Almodaresi *et al.*, 2017), Mantis (Pandey *et al.*, 2018) and the method Almodaresi *et al.* (2019) are efficient with respect to memory and time but cannot be updated without reconstructing the entire colored de Bruijn graph. Bloom Filter Trie (Holley *et al.*, 2016) and the method of Karasikov *et al.* (2019) allow for the addition of new data but cannot scale to large datasets, as we show in this paper. Crawford *et al.* (2019) build a de Bruijn graph that can be updated but restrict interest to the traditional (non-colored) de Bruijn graph.

Here, we focus on scalable construction of the colored de Bruijn graph in a manner that allows new data to be added without reconstruction. One way to achieve this construction is to devise a divide-and-conquer approach which will divide the dataset into smaller sets, construct a succinct colored de Bruijn graph for each smaller set, and merge these succinct colored de Bruijn graphs into progressively larger graphs until a single one remains. The problem of merging succinct colored de Bruijn graphs efficiently is a challenging problem but necessary to solve since it avoids memory, disk and time overhead. This divide-and-conquer approach also allows the graph to be updated. Given an additional dataset, a compressed colored de Bruijn graph can be constructed for it, and then merged into the existing (larger) compressed colored de Bruijn graph.

Our contributions. We present VariMerge which constructs a colored de Bruijn graph for large datasets in a manner that allows new data to be added efficiently. As suggested earlier, VariMerge builds through a process of dividing the data into smaller sets, building the colored de Bruijn graph in a RAM-efficient manner for each smaller set, and merging the resulting colored de Bruijn graphs. Each of the colored de Bruijn graphs is stored using the FM-index in the same manner as Vari (Muggli *et al.*, 2017). Thus, the algorithmic challenge that we tackle is merging the graphs in a manner that keeps them in their compressed format throughout the merging process — rather than decompressing, merging and compressing, which would be impractical with respect to disk and memory usage.

We verify the correctness of our approach and the accompanied bubble-calling algorithm by showing the colored de Bruijn graph built via merging is bit-for-bit identical to standard construction, and successfully identifies all bubbles in the merged graph. Next, we demonstrate that VariMerge improves construction scalability over Vari, reducing the running time by a third and the working space three fold by comparing the peak disk and memory required to build the colored de Bruijn graph for 8,000 Salmonella strains using Vari and VariMerge.

Next, we use VariMerge to build a colored de Bruijn graph for 16,000 strains of Salmonella. This construction required 254 GB of RAM, 2.34 TB of external memory, and approximately 140 hours of CPU time. To contextualize these results, we compare the construction of VariMerge with those of state-of-the-art methods, including Vari (Muggli *et al.*, 2017) / Rainbowfish (Almodaresi *et al.*, 2017), Bloom Filter Trie (Holley *et al.*, 2016), Mantis (Pandey *et al.*, 2018) / the method of Almodaresi *et al.* (2019), and the method of Karasikov *et al.* (2019), on datasets consisting of 4,000, 8,000 and 16,000 strains. Further, we demonstrate the ability of the graph to be updated efficiently through VariMerge by showing a Salmonella strain can be added to the graph in an order of magnitude faster than its initial construction. These results demonstrate that VariMerge is the only method that is capable of building the colored de Bruijn graph for 16,000 strains in a manner where new data can be added. Mantis was the only other method that was capable of building the colored de Bruijn graph for 16,000 strains, yet it is unable to update the graph without reconstruction. Although the authors suggest a dynamic update strategy in future work, no implementation is available. In addition, we note that VariMerge has several practical advantages over Mantis — it has a bubble calling implementation which allows variants to be detected from the graph and is capable of constructing the graph for any value of k up to 64. All other methods were not scalable to 16,000 and thus, were unable to complete construction in 150 hours and using at most 4 TB of disk space and 750 GB of memory.

2 Related Work

Efficient de Bruijn graph. Space-efficient representations of de Bruijn graphs have been heavily researched in recent years. One of the first approaches was introduced with the creation of the ABySS assembler, which stores the graph as a distributed hash table (Simpson *et al.*, 2009). Conway and Bromage (2011) reduced these space requirements by using a sparse bit vector representation, which is due Okanohara and Sadakane (2007), to represent the edges in the graph, and using rank and select operations (to be described shortly) to traverse the edges. Minia (Chikhi and Rizk, 2013) use a Bloom filter to store the edges, which requires the graph to be traversed by generating all possible outgoing edges at each node and testing their membership in the Bloom filter. Bowe *et al.* (2012) develop a succinct data structure based on the Burrows-Wheeler transform (BWT). This data structure is combined with ideas from IDBA-UD (Peng *et al.*, 2012) in order to create MEGAHIT (Li *et al.*, 2015). Chikhi *et al.* (2014) describe a space-efficient data structure that combines the use of the FM-index and minimizers.

Efficient colored de Bruijn graphs. As previously mentioned, Vari (Muggli *et al.*, 2017) and Rainbowfish (Almodaresi *et al.*, 2017) are both space-efficient data structures for storing the colored de Bruijn graph which each use the structure of (Bowe *et al.*, 2012). Muggli *et al.* (2017) build the compressed color matrix by compressing each row using Elias-Fano encoding. Rainbowfish (Almodaresi *et al.*, 2017) takes as input the color matrix of Vari and compress it by decomposing the matrix into "color sets" based on an equivalence relation and compresses each color set individually. Thus, this method relies on the construction of Vari prior to building the sets of compatible colors. Holley *et al.* (2016) introduced the Bloom Filter Trie, which is another succinct data structure for the colored de Bruijn graph. It encodes frequently occurring sets of colors separate from the graph and stores a reference to the set if the reference takes fewer bits than the set itself. This data structure allows incremental updates of the underlying graph. More recently, Mantis (Pandey *et al.*, 2018) and its extension (Almodaresi *et al.*, 2019) improve upon Vari and Rainbowfish. They build a compressed colored de Bruijn graph by building sets of compatible colors (similar to Rainbowfish) but construct the compressed graph directly rather than constructing it from Vari. Lastly, the most recent method of Almodaresi *et al.* (2019) constructs the colored de Bruijn graph using Mantis, which it further compresses by careful reconstruction and compression of the color matrix.

Lastly, there are a couple of methods that construct the color matrix in a manner that is both compressed and dynamic, which include the method of Mustafa *et al.* (2019) and Multi-BRWT (Karasikov *et al.*, 2019). These methods use a simple representation of a de Bruijn graph where each edge (the k-mer) is stored in a hash table. This graph representation allows it to be updated along with the color matrix — thus, the main contribution is not the data structure used to store the graph but that used to store the color matrix.

Other Related Compressed Data Structures. Some related compressed data structures are SeqOthello (Yu *et al.*, 2018), SBT (Solomon and Kingsford, 2016), Split-SBT (Solomon and Kingsford, 2018), and Allsome-SBT (Sun *et al.*, 2017). These methods index all k-mers or variants of k-mer indexes but do not provide graph information. For this reason, they are frequently applied to querying large collections of RNA-seq data but not genome assembly. Another indexing method is BIGSI (Bradley *et al.*, 2017), which provides graph features and can be viewed as a probabilistic colored de Bruijn graph.

Two other approaches are worthy of note because they merge the BWT of a set of strings. BWT-Merge (Sirén, 2016) is related to our work since the data structure we construct and store is similar to BWT. BWT-Merge merges two strings stored using BWT by using a reverse trie of one BWT to generate queries that are then located in the other BWT using FM-Index backward search. The reverse trie allows the common suffixes across multiple merge elements to share the results of a single backward search step. Thus, BWT-Merge finds the final rank of each full suffix completely, one suffix at a time. Finally, MSBWT (Holt and McMillan, 2014) is a method which merges the BWTs of multiple strings in a method similar to our own except applied to strings instead of graphs. Lastly, Egidi *et al.* (2019) recently improved upon the algorithm in this paper. Their algorithm has the same asymptotic cost as the method presented here but is more space-efficient.

In Section 5, we compared the performance of VariMerge against Vari (Muggli *et al.*, 2017) / Rainbowfish (Almodaresi *et al.*, 2017), Bloom Filter Trie (Holley *et al.*, 2016), Multi-BRWT (Karasikov *et al.*, 2019) and Mantis (Pandey *et al.*, 2018) / the method Almodaresi *et al.* (2019). Among all mentioned methods in this section we chose these tools based on the following criteria: the method should be both graph-based and non-probabilistic (exact) for a fair comparison.

3 Preliminaries

As previously mentioned, Vari (Muggli *et al.*, 2017) represents the colored de Bruijn graph using BWT and VariMerge efficiently merges de Bruijn graphs that are represented in this manner. Here, we first define some basic notation and definitions concerning BWT, then show how the colored de Bruijn graph can be stored using BWT.

3.1 Basic Definitions and Terminology

Here, we begin with some basic definitions related to our representation. Throughout we consider a string $\mathsf{X} = \mathsf{X}[1..n] = \mathsf{X}[1]\mathsf{X}[2]\dots\mathsf{X}[n]$ of $|\mathsf{X}| = n$ symbols drawn from the alphabet $[0..\sigma-1]$. For $i = 1, \dots, n$ we write $\mathsf{X}[i..n]$ to denote the *suffix* of X of length $n-i+1$, that is $\mathsf{X}[i..n] = \mathsf{X}[i]\mathsf{X}[i+1]\dots\mathsf{X}[n]$. Similarly, we write $\mathsf{X}[1..i]$ to denote the *prefix* of X of length i. $\mathsf{X}[i..j]$ is the *substring* $\mathsf{X}[i]\mathsf{X}[i+1]\dots\mathsf{X}[j]$ of X that starts at position i and ends at j.

The suffix array SA_X of input text X is an array $\mathsf{SA}[1..n]$ of length n that contains a permutation of the integers $[1..n]$, where $\mathsf{X}[\mathsf{SA}[1]..n] \prec \mathsf{X}[\mathsf{SA}[2]..n] \prec \cdots \prec \mathsf{X}[\mathsf{SA}[n]..n]$ and $\prec$ denotes lexicographic precedence. Next, for a string Y, we refer to the Y-interval in the suffix array SA_X as the interval $\mathsf{SA}[s..e]$ that contains all suffixes having Y as a prefix. For a character c and a string Y, the computation of $c\mathsf{Y}$-interval from Y-interval is called the *left extension*.

BWT and FM-index. For a string Y, we denote A as the list of Y's characters sorted lexicographically by the suffixes starting at those characters. Further, we denote B be the list of Y's characters sorted lexicographically by the suffixes starting immediately after those characters. Thus, if $\mathsf{Y}[i] = \mathsf{Y}[j]$ then $\mathsf{Y}[i]$ and $\mathsf{Y}[j]$ have the same relative order in both lists. This implies that if $\mathsf{Y}[i]$ is in position p in B then in A it is in position

$$|\{h : \mathsf{Y}[h] \prec \mathsf{Y}[i]\}| + |\{h : \mathsf{L}[h] = \mathsf{Y}[i],\ h \le p\}| - 1\,.$$

We note that the last character in Y always appears first in B. It follows that we can recover Y from B, which is the principle of BWT (Burrows and Wheeler, 1994).

Ferragina and Manzini (2005) showed BWT can be used for indexing as follows. Here, we denote $\mathsf{BWT}(\mathsf{Y})$ as the BWT array computed for Y. Hence, if we know the range $\mathsf{BWT}(\mathsf{Y})[i..j]$ occupied by characters immediately preceding occurrences of a pattern P in Y, then we can compute the range $\mathsf{BWT}(\mathsf{Y})[i'..j']$ occupied by characters immediately preceding occurrences of cP in Y, for any character c, since

$$\begin{aligned} i' &= |\{h : \mathsf{A}[h] \prec c\}| + |\{h : \mathsf{B}[h] = c, h < i\}| \\ j' &= |\{h : \mathsf{A}[h] \prec c\}| + |\{h : \mathsf{B}[h] = c, h \le j\}| - 1\,. \end{aligned}$$

As can be seen above: $j' - i' + 1$ is the number of occurrences of cP. To recap, the FM-index for Y requires: (1) an array that stores $|\{h : \mathsf{Y}[h] \prec c\}|$ for each character c and, (2) a (rank) data structure for $\mathsf{BWT}(\mathsf{Y})$ that returns how many times a given character occurs up to a specific position. The latter data structure is used in backward search in order to compute the left extension of a given string.

3.2 Storage of de Bruijn Graph using BWT

We now give a brief explanation of the data structure behind Vari. An example of this representation is shown in Figure 2 in Section 7 in the Supplement. We refer the reader to Muggli *et al.* (2017) for a full explanation.

Given a de Bruijn graph $G = (V, E)$, we assume each edge $e \in E$ has a k-mer corresponding to it. We define the co-lexicographic (colex) ordering of V as the lexicographic order of their reversed $(k-1)$-mers. We let F be the edges in E in colex order by their ending nodes, where ties are broken by their starting nodes, and let L be the edges in E sorted colex by their starting nodes, with ties broken by their ending nodes. We define $\mathsf{label}(e)$ as a function that takes in an edge $e \in E$ and returns the final symbol of the k-mer corresponding to it.

If we are given two edges e and e' that have the same label then they have the same relative order in both F and L; otherwise, their relative order in F is the same as their labels' lexicographic order. This means that if e is in position p in L, then in F it is in position

$$|\{d : d \in E,\ \mathsf{label}(d) \prec \mathsf{label}(e)\}| + |\{h : \mathsf{label}(\mathsf{L}[h]) \\ = \mathsf{label}(e),\ h \leq p\}| - 1.$$

We let $\mathsf{Edge\text{-}BWT}(G)$ be the sequence of edge labels given the ordering of the edges in L. Thus, we let $\mathsf{label}(\mathsf{L}[h])$ be equal to $\mathsf{Edge\text{-}BWT}(G)[h]$ for all h. We let B_F be the bit vector with a 1 marking the position in F of the last incoming edge of each node, and let B_L be the bit vector with a 1 marking the position in L of the last outgoing edge of each node. Therefore, given a character c and the index of a node v[1], we can use B_L to find the interval in L containing v's outgoing edges and search in $\mathsf{Edge\text{-}BWT}(G)$ to find the position of edge e labeled c. Similarly, we can find all incoming edges of a node v using their position in F and B_F.

We annotate each bit of B_F to the corresponding symbols in $\mathsf{Edge\text{-}BWT}(G)$. We let flags be a bit array of such annotation bits. In practice, we store $\mathsf{Edge\text{-}BWT}(G)$, B_L, and flags (rather than B_F) to store the de Bruijn graph but limit our discussion to the construction that uses B_F for ease of explanation. We refer the reader to Bowe *et al.* (2012) for a full explanation of flags.

We note that an important aspect of this succinct representation of the graph is that the $(k-1)$-mers (nodes) and k-mers (edges) of the de Bruijn graph G are not explicitly stored in the above representation — rather they than can be *computed* from this representation. We can efficiently traverse the graph in a forward or reverse manner and recover incoming and outgoing edges of a given node v. Therefore, given a node v identified by its index, we can recover the $(k-1)$-mer corresponding to v by traversing the graph in a backward direction $(k-1)$ times starting from v. See Section 8 in the Supplement for more details. This also illustrates the necessity of adding extra nodes and edges to the graph, which we refer to as *dummy* nodes and edges. In order to ensure there is a directed path of length at least $k-1$ to each original node, we augment the graph with additional nodes (and edges) so that each new node has a $(k-1)$-mer that is prefixed by one or more copies of a special symbol \$ not in the alphabet and lexicographically strictly less than all others. When new nodes are added, we are assured that the node corresponding to $\$^{k-1}$ is always first in colex order and has no incoming edges. Lastly, we augment the graph in a similar manner by adding extra outgoing edges, whose k-mer ends in \$, doing so for each original node with no outgoing edge.

[1] We note that there is no explicit stored set of nodes. Hence, we refer to the "node index" as the position among all nodes if sorted in colex order.

Storage of colors. Given a multiset $\mathcal{G} = \{G_1, \ldots, G_t\}$ of individual de Bruijn graphs, we set G to be the union of those individual graphs and build the previously-described representation for G. We also build and store a two-dimensional binary array C in which $\mathsf{C}[i, j]$ indicates whether the ith edge in G is present in the jth individual de Bruijn graph.

4 Method

In this section, we begin by describing a naive merge algorithm, which will motivate the necessity of the succinct algorithm. In the description of both algorithms, we describe how to merge two colored de Bruijn graphs but note that it generalizes to an arbitrary number of graphs. Hence, we assume that we have two de Bruijn graphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ as input, which are stored as $\mathsf{Edge\text{-}BWT}(G_1)$, B_{L1}, B_{F1}, and C_1 as well as $\mathsf{Edge\text{-}BWT}(G_2)$, B_{L2}, B_{F2}, and C_2, respectively. And we output the merged graph $G_M = (V_M, E_M)$ as $\mathsf{Edge\text{-}BWT}(G)_M$, B_{LM}, B_{FM}, and C_M. An illustration of G_1, G_2, G_M, and the corresponding data structures is given in Figure 1.

4.1 A Naive Merge Algorithm

We recall from Section 3 that the edges (k-mers) of G_1 and G_2 can be computed from the succinct representation. We denote these k-mers for G_1, G_2 as L_1 and L_2, respectively. For example, if we want to reconstruct the k-mer `AGAGAGTTA` contained in G_1 which is stored as `A` in $\mathsf{Edge\text{-}BWT}(G_1)$, we need to backward navigate in G_1 from the edge labeled `A` through $k-1$ predecessor edges (`T`, `T`, `G`,...), and concatenate the abbreviated characters encountered during this backward navigation in reverse order. Thus, a naive merging algorithm would reconstruct L_1 and L_2 then merge them, which can be trivially given they are in sorted (colex) order. This algorithm requires explicitly building L_1, L_2, and the merged list. Thus, it has a significant memory footprint. See Algorithm 3 in Section 9 for the details of this naive merge algorithm.

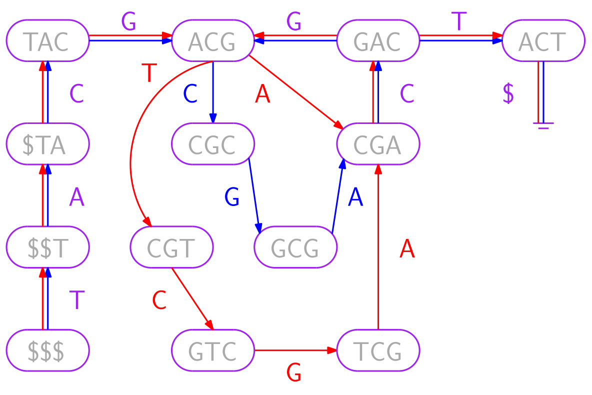
4.2 The Succinct Merge Algorithm

Before we give a detailed explanation of our succinct merge algorithm, we consider the problem of merging two sorted lists of strings with the constraint that we can only examine a single character from each string at a time. We can solve this problem with a divide and conquer approach. First, we group all the strings in each list by their first character. This partially solves the problem, as we know all the strings in the first group from each list must occur in the output before all the strings in the second group in each list and so on. Thus, the problem is now reduced to merging the strings in the first group, followed by merging the strings in the second group and so on. Each of these merges can be addressed by again grouping the elements (i.e. subgroups of the initial groups) by examining the second character of each string. We can apply this step recursively until all characters of each string have been examined. We now draw the reader's attention to the fact that our succinct colored de Bruijn graph representation is a space-efficient representation of the list of sorted k-mers.

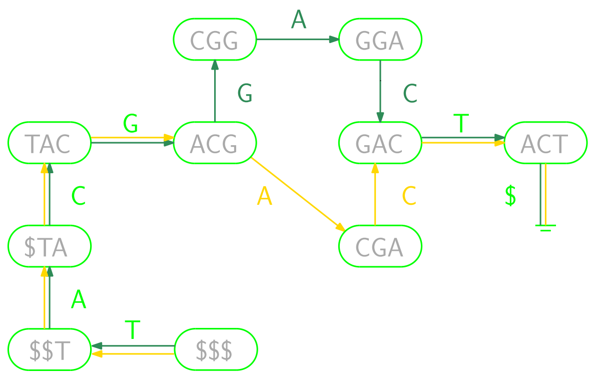
4.2.1 Overview of the Algorithm.

We now return to the problem of merging succinct colored de Bruijn graphs. The algorithm consists of two steps: (1) a planning step which plans the merge, and (2) an executing step which executes the merge. Thus, the planning step outputs a *merge plan*, which consists of lists of non-overlapping intervals for L_1 and L_2. These lists detail how to construct the data structure for the merged graph from the succinct data structures for G_1 and G_2.

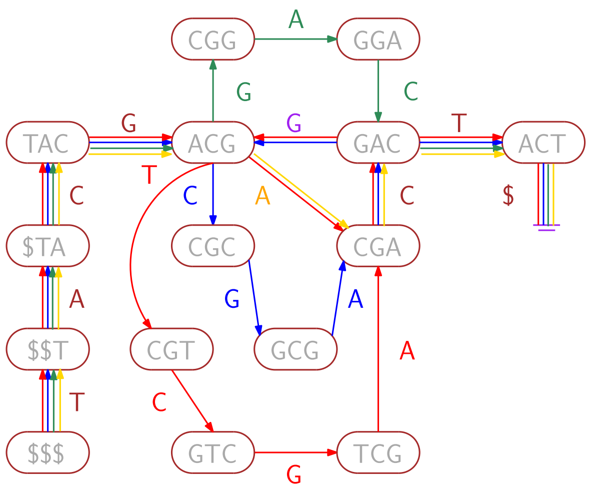
We refer to $\mathsf{Edge\text{-}BWT}$ and C as the *primary components* of the data structure and B_F and B_L as *secondary components*. We describe how to



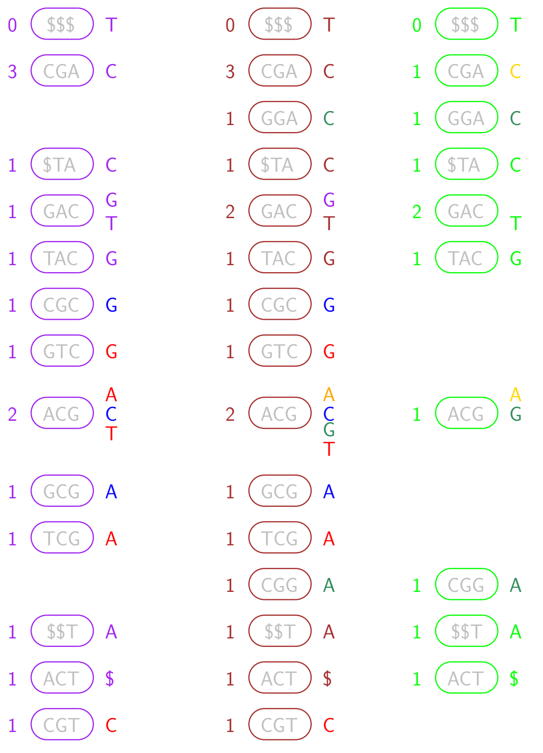
(a) A colored de Bruijn graph G_1.



(b) A second colored de Bruijn graph G_2.



(c) The merged colored de Bruijn graph G_M.



(d) The representation of the merged colored de Bruijn graph.

Fig. 1: **(a):** A colored de Bruijn graph consisting of two individual graphs, whose edges are shown in red and blue. The nodes are shown in purple because they can occur in either graph. **(b):** A second colored de Bruijn graph, whose edges are green and yellow. Again, the nodes are shown in lime because they can occur in either graph. **(c) :** A colored de Bruijn graph merged from the two colored de Bruijn graphs. **(d):** The nodes for all three graphs arranged in columns (red and blue, merged, green and yellow). Each column is sorted into co-lexicographic order, with each node's number of incoming edges shown on its left and the labels of its outgoing edges shown on its right. Vertical alignment illustrates how the merged components (center) are copied from either the left, the right, or both.

merge the primary components, and leave the details of how to merge the secondary components to Section 12 in the Supplement.

4.2.2 The Planning Step.

Formally, we denote the merge plan as $P_1 = \{[0, p_1^1], \ldots, [p_i^1, |\mathsf{L}_1| - 1]\}$, where each $p_1^1, \ldots, p_i^1$ is an index in L_1, and $P_2 = \{[0, p_1^2], \ldots, [p_i^2, |\mathsf{L}_2| - 1]\}$, where each $p_1^2, \ldots, p_i^2$ is an index in L_2. An overview of the planning step is given in Algorithm 1. We first initialize P_1 and P_2 to be single intervals covering L_1 and L_2, respectively (e.g. $P_1 = \{[0, |\mathsf{L}_1| - 1]\}$ and $P_2 = \{[0, |\mathsf{L}_2| - 1]\}$). Next, we revise P_1 and P_2 in an iterative manner k times (where k corresponds to the k-mer value). At each iteration of the algorithm, a single character of the strings in L_1 and L_2 is processed, and the merge plan is revised. Thus, in order to fully describe the planning stage, we define (1) how the characters of L_1 and L_2 are computed, and (2) how P_1 and P_2 are revised based on these characters. An illustration of the merge plans is given in Figure 3 in Section 10 of the Supplement.

Computing the next character of L_1 *and* L_2. Let i denote the current iteration of our revision of P_1 and P_2, where $1 \leq i < k$. We compute the next character of L_1 and L_2 using two temporary character vectors

Algorithm 1 The planning step to merge G_1 and G_2.

```
$P_1 \leftarrow ([1, |EBWT(G)_1|])$
$P_2 \leftarrow ([1, |EBWT(G)_2|])$
$Col_1 \leftarrow []$
$Col_2 \leftarrow []$
% Iterate through "edge label matrix" columns in sort precedence order
for all $i \in \{1..k\}$ do
    $Col_1 \leftarrow GetCol(i, Col_1, G_1)$
    $Col_2 \leftarrow GetCol(i, Col_2, G_2)$
    % RefinePlan is given in Algorithm 2.
    $(P_1', P_2') \leftarrow RefinePlan(P_1, P_2, Col_1, Col_2, i)$
    $(P_1, P_2) \leftarrow (P_1', P_2')$
end for
```

Col_1^i and Col_2^i, which are of length $|\mathsf{L}_1|$ and $|\mathsf{L}_2|$, respectively. We note that the "next" character is the preceding character in the k-mer since $\mathsf{Edge\text{-}BWT}(G_1)$ and $\mathsf{Edge\text{-}BWT}(G_2)$ only store the last character of L_1 and L_2. Thus, we process the characters of L_1 and L_2 from right to left. Conceptually, we define these vectors as follows: $Col_1^i[j] = \mathsf{L}_1[j][k-i]$ if $i < k$ and otherwise $Col_1^i[j] = \mathsf{L}_1[j][k]$, and $Col_2^i[j] = \mathsf{L}_2[j][k-i]$ if $j < k$; and otherwise $Col_2^i[j] = \mathsf{L}_2[j][k]$. Yet, since we do not explicitly build or store L_1 and L_2, we must compute Col_1^i and Col_2^i.

We define the computation of Col_1^i by describing the following three cases. When $1 < i < k$, we compute Col_1^i by traversing G_1 in a forward direction from the first incoming edge of every node and copying the character found at the $(q+1)$-th position of that incoming edge (again, stored in Col_1^{i-1}) into q-th position of all outgoing edges of that node. When $i = 1$, the $(q+1)$-th position corresponds to $\mathsf{Edge\text{-}BWT}(G_1)$, so $\mathsf{Edge\text{-}BWT}(G_1)$ is used in place of Col_1^{i-1} but is otherwise identical to the previous case. Lastly, when $i = k$, we let Col_1^i equal $\mathsf{Edge\text{-}BWT}(G_1)$. We compute Col_2^i in an analogous manner. An illustration of how to compute the next column of L_1 and L_2 is given in Section 11 of the Supplement. Moreover, the pseudocode for *GetCol* is shown Algorithm 11 in this same section.

Revising P_1 and P_2. We revise P_1 and P_2 based on Col_1^i and Col_2^i at iteration i by considering each pair of intervals in P_1 and P_2, i.e., $P_1[n]$ and $P_2[n]$ for $n = 1, \ldots, |P_1|$, and partitioning each interval into at most five sub-intervals. We store the list of sub-intervals of P_1 and P_2 as $SubP_1$ and $SubP_2$. Intuitively, we create $SubP_1$ in order to divide $P_1[n]$ based on continuous ranges in Col_1^i that have the same character, e.g., each continuous range of `A`'s, `C`'s, `G`'s, `T`'s or `$`'s. Similarly, $SubP_2$ is used to divide $P_2[n]$. We divide P_1 by first computing the subvector of Col_1^i that is covered by $P_1[n]$, which we denote as $Col_1^i(P_1[n])$, and computing the subvector of Col_2^i that is covered by $P_2[n]$, which we denote as $Col_2^i(P_2[n])$. Next, given character c, we populate $SubP_1[c]$ and $SubP_2[c]$ based on $Col_1^i(P_1[n])$ *and* $Col_2^i(P_2[n])$ as follows: (1) we check whether c exists in either $Col_1^i(P_1[n])$ or $Col_2^i(P_1[n])$; (2) if so, we add an interval to $SubP_1[c]$ covering the contiguous range of c in $Col_1^i(P_1[n])$ (or add an empty interval if $Col_1^i(P_1[n])$ lacks any instances of c), and add an interval to $SubP_2[c]$ covering the contiguous range of c in $Col_2^i(P_1[n])$ (or, likewise, add an empty interval if $Col_2^i(P_1[n])$ lacks any instances of c)[2]. Finally, we concatenate all the lists in $SubP_1$ and $SubP_2$ to form the revised plan P_1' and P_2'. This revised plan P_1' and P_2' becomes the input P_1 and P_2 for the next refinement step. Figure 3 in the Supplement shows three merged plans, including two refined ones (green and blue). The pseudocode for this step is given in Algorithm 2.

[2] We are guaranteed by the definition of our data structure that any instances of c in $Col_1^i(P_1[n])$ will be in a contiguous range, and likewise, any instances of c in $Col_2^i(P_1[n])$ will also be in a contiguous range

We crafted the method above to maintain the property described in the following observation.

Observation 1. *Let P_1 be a (partial) merge plan, and P_1' its refinement by our merge algorithm, where $\ell_1, .., \ell_n$ are the elements in L_1 that are covered by interval $p_i \in P_1$ and $m_1, ..., m_o$ are the elements of L_2 covered by interval $q_j \in P_2$. The following conditions hold: (1) $|P_1| = |P_2|$ and $|P_1'| = |P_2'|$; (2) given any pair of elements where $\ell_a \in p_i$, $\ell_b \in p_j$ and $p_i \cap p_j = \emptyset$ there exists intervals p_i' and p_j' in P_1' such that $p_i' \cap p_j' = \emptyset$ and $\ell_a \in p_i'$, $\ell_b \in p_j'$; and lastly, (3) given an interval p_i in P_1 and the subsets of the alphabet used $\sigma_1 \in \ell_1, .., \ell_n$ and $\sigma_2 \in m_1, ..., m_o$, then p_i will be partitioned into $|SubP_1| = |\sigma_1 \cup \sigma_2|$ subintervals in P_1'.*

We defined this observation for P_1 but note that an analogous observation exists for P_2.

Algorithm 2 Revising P_1 and P_2

```
procedure Partition($W_1$,$W_2$)
    $\Sigma' \leftarrow AlphabetUsed(W_1, W_2)$
    $SubP_1 \leftarrow ()$
    $SubP_2 \leftarrow ()$
    for all $c \in \Sigma'$ do
        $SubP_1.Append(IntervalOccupied(c, W_1))$
        $SubP_2.Append(IntervalOccupied(c, W_2))$
    end for
    return $(SubP_1, SubP_2)$
end procedure

procedure RefinePlan($P_1$, $P_2$, $Col_1$, $Col_2$, $i$)
    $P_1' \leftarrow ()$
    $P_2' \leftarrow ()$
    % For each interval in $P_1$ (and $P_2$)
    for all $j \in \{1..|P_1|\}$ do
        %...extract a window from each column covered by the interval...
        $W_1 \leftarrow CoveredSymbols(Col_1, P_1[j])$
        $W_2 \leftarrow CoveredSymbols(Col_2, P_2[j])$
        %...and partitioning that window on its character runs, forming sub-intervals.
        $(SubP_1, SubP_2) \leftarrow Partition(W_1, W_2)$
        $P_1'.Concatenate(SubP_1)$
        $P_2'.Concatenate(SubP_2)$
    end for
    if $i \in \{1, k-1, k-2\}$ then
        $S_i \leftarrow (P_1', P_2')$
    end if
    return $(P_1', P_2')$
end procedure
```

4.2.3 The Execution Step.

We execute the merge plan by combining the elements of $\mathsf{Edge\text{-}BWT}(G_1)$ that are covered by an interval in P_1 with the elements of $\mathsf{Edge\text{-}BWT}(G_2)$ that are covered by the same interval in P_2 into a single element in $\mathsf{Edge\text{-}BWT}(G_M)$. We note that when all characters of each k-mer in L_1 and L_2 have been computed and accounted for, each interval in P_1 (and P_2) will cover either 0 or 1 element of L_1 (and L_2) and the number of intervals in P_1 (equivalently P_2) will be equal to $|\mathsf{Edge\text{-}BWT}(G_M)|$. Thus, we consider and merge each pair of intervals of P_1 and P_2 in an iterative manner. We let (p_i^1, p_i^2) as the i-th pair of intervals. We concatenate the next character of $\mathsf{Edge\text{-}BWT}(G_1)$ onto the end of $\mathsf{Edge\text{-}BWT}(G_M)$

if $|p_i^1| = 1$. If $|p_i^2| = 1$ then we dismiss the next character of Edge-BWT(G_2) since it corresponds to the edge that was just added. Lastly, if $|p_i^1| = 0$ and $|p_i^2| = 1$, we copy the next character from Edge-BWT(G_2) onto the end of Edge-BWT(G_M).

We merge the color matrices in an identical manner by copying elements of C_1 and C_2 to C_M. Again, we iterate through the plan by considering each pair of intervals. If $|p_i^1| = 1$ and $|p_i^2| = 1$ then we concatenate the corresponding rows of C_1 and C_2 to form a new row that is added to C_M. If only one of p_i^1 or p_i^2 is non-zero then the corresponding row of C_1 or C_2 is copied to C_M with the other elements of the new row set to 0. Lastly, we note that we discussed the planning and execution steps of merging the primary components. As previously mentioned, details about merging the secondary components are given in Section 12 in the Supplement.

The following theorem demonstrates the efficiency of our approach. The proof of the following theorem is found in Section 13 in the Supplement.

Theorem 1. *Given two colored de Bruijn graphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ constructed for k, where $|E_1| \geq |E_2|$. It follows that our merge algorithm constructs the merged colored de Bruijn graph G_M in $O(|E_1| \cdot \max(k, t))$-time, where t is the number of colors in G_M.*

5 Results

In this section, we show the correctness of our method, the reduction in the resources used to build a colored de Bruijn graph by merging, and the comparison between state-of-the-art methods for building colored de Bruijn graphs. We ran all performance experiments on a machine with two Xeon E5-2640 v4 chips, each having 10 2.4 GHz cores. The system contains 755 GB of RAM and two ZFS RAID pools of 9 disk each for storage. We report wall clock time and maximum resident set size from Linux.

5.1 Validation on E coli.

In order to validate the correctness of our approach, we generated two succinct colored de Bruijn graphs with two sets of three *E. coli* incomplete assemblies each, merged them to a single six-color graph, and verified its equivalence to a six-color graph built from scratch using Vari. We obtained the six sub-strains of *E. coli* K-12 from NCBI. Each of the genomes contained approximately 4.6 million base pairs and had a median GC content of 49.9%. This experiment tests that the merged colored de Bruijn graph built by VariMerge is equivalent to that produced by building the graph without merging (i.e., with Vari alone) even in the presence of any redundant dummy edges. We tested equivalence by running a bubble calling algorithm on both six-color graphs and found the identical set of bubbles between them. Further, we found VariMerge produced files on disk that were bit-for-bit identical to those generated by Vari when consuming complete assemblies, demonstrating the merge algorithm perfectly merges the succinct source graphs. We leave additional details of this validation to the supplement (see Table 4 in the supplement).

5.2 Demonstration of Large-scale Construction and Incremental Updates

We downloaded the sequence data from for 16,000 Salmonella strains (NCBI BioProject PRJNA18384), assembled them individually with IDBA, then divided them into four sets of 4,000 strains, which we label 4A, 4B, 4C, and 4D. From these datasets we construct a colored de Bruijn graph for 8,000 strains, to demonstrate the efficiency of the merge process, and 16,000 strains, to demonstrate scalability. The exact accessions for each dataset are available in our repository.

In order to measure the effectiveness of VariMerge for the proposed divide-and-conquer method of building large graphs, we constructed the colored de Bruijn graph using Vari for a set of 4,000 salmonella assemblies (4A). This took 8 hours 46 minutes, 1 TB of external memory, and 136 GB of RAM to build the graph for 4,000 strains. We then built a graph for a second set of 4,000 assemblies (4B) using 10 hours 40 minutes, 1.5 TB of external memory, and 137 GB of RAM. We merged these two 4,000 sample graphs (i.e. 8AB) using our proposed algorithm in 2 hours 1 minutes, no external memory, and 10 GB of RAM. "'0" is shown in the external memory column of Table 1 for merge since no external memory is ever used for merging. Thus the VariMerge method required a combined 137 GB of RAM, 26 hours 30 minutes of runtime to produce the graph for 8,000 strains. We denote this graph with 8,000 strains as 8AB. In contrast, running Vari on the same 8,000 strains required 37 hours 27 minutes, 4.6 TB of external memory and 271 GB of RAM. Thus VariMerge reduced runtime by 11 hours, reducing RAM requirements to 134 GB, and reducing external memory requirements by 3.1 TB.

We further used this facility to merge two more 4,000 color graphs (i.e. 4C + 4D) (Table 2). We denote the resulting graph as 8CD. We then merged this 8,000 sample graph with the aforementioned 8,000 color graph to produce a succinct colored de Bruijn graph of 16,000 samples (i.e. 8AB + 8CD).

In order to measure the effectiveness of VariMerge for incremental additions to a graph that holds a growing population of genomes, we started with the colored de Bruijn graph of 16,000 salmonella assemblies. We then constructed a second graph for a singleton set of just one additional assembly. Next, we ran our proposed merge algorithm on these two graphs. VariMerge took 69 hours 8 minutes, 2.34 TB of external memory, and 254 GB of RAM to build the graph for 16,000 strains. To build a single colored de Bruijn graph for an additional strain, Vari took 7 seconds, 460 MB of external memory, and 2.3 GB of RAM. Our proposed merge algorithm took 7 hours 9 minutes, no external memory, and 254 GB of RAM to merge the 16,000 color graph with the 1 color graph. This is an order of magnitude faster than the almost 70 hours it would take to build the same 16,001 color graph from scratch.

This experiment also reveals that increasing the number of divisions (below a reasonable threshold) will lower the time, memory, and external memory – with the trade-off being the practical challenge of dividing and merging the data. Nonetheless, this is one of the benefits of VariMerge, as smaller and smaller memory and external memory could be used by making further divisions of the data.

5.3 Comparison to Existing Methods

We compare our method to the existing space- and memory- efficient colored de Bruijn graphs. To accomplish this, we ran Bloom Filter Trie (Holley *et al.*, 2016), Vari (Muggli *et al.*, 2017) / Rainbowfish (Almodaresi *et al.*, 2017), Mantis (Pandey *et al.*, 2018) / the method of Almodaresi *et al.* (2019), and Multi-BRWT (Karasikov *et al.*, 2019). The peak RAM use and running time required by the methods to construct a colored de Bruijn graph for 4,000, 8,000, and 16,000 strains is shown in Table 3. In addition, Table 3 illustrates the output size of all the methods. We note that all methods were run with default parameters and $k = 32$, except from Bloom Filter Trie which can only work with k values which are multiples of 9, hence we ran it with $k = 27$. We ran Mantis with their "log-slot" parameter set to 33 and 36 for the experiments using 4,000 and 8,000 strains, and 16,000 strains, respectively. All methods are exact, colored de Bruijn graph construction methods. Since Rainbowfish first runs Vari and then compresses, the peak RAM, peak disk usage, and time will be at least that of Vari for construction; this is why they are shown together in Table 3. Similarly, we report peak RAM, peak disk usage and time of Mantis together with that of the method of Almodaresi *et al.* (2019)

	Input Stats		de Bruijn Graph			Color Matrix			Combined Requirements			
Program and Dataset	**k-mers**	**Colors**	**RAM**	**Time**	**Size**	**RAM**	**Time**	**Size**	**RAM**	**Ext. Mem.**	**Time**	**Size**
Vari(4A)	1.1 B	4,000	136 GB	8 h 46 m	0.31 GB	52 GB	1 h 39 m	51.2 GB	136 GB	1 TB	10 h 25 m	51 GB
Vari(4B)	1.5 B	4,000	137 GB	10 h 40 m	0.52 GB	54 GB	2 h 22 m	52.5 GB	137 GB	1.5 TB	13 h 2 m	53 GB
Merge(4A, 4B)	2.4 B	8,000	10 GB	2 h 1 m	0.63 GB	117 GB	1 h 2 m	106 GB	117 GB	0 TB	3 h 3 m	106 GB
VariMerge	2.4	8,000	137 GB	21 h 27 m	0.63 GB	117 GB	5 h 3 m	117 GB	137 GB	1.5 TB	26 h 30 m	106 GB

Table 1. Breakdown of the memory, disk and time usage of VariMerge to build the colored de Bruijn graph for 8,000 strains. The VariMerge method consists of running Vari on subsets of the population (4A and 4B) and then merging the results with our proposed merge algorithm (denoted Merge here). We list the resources used for both individual runs of Vari, the Merge required, and the combined resources. The combined resources consist of the total time and maximum space used across all three components of VariMerge used in this dataset. No external memory is needed for merging itself so "0" is in the external memory column for Merge

	Input Stats		de Bruijn Graph			Color Matrix			Combined Requirements			
Program and Dataset	**k-mers**	**Colors**	**RAM**	**Time**	**Size**	**RAM**	**Time**	**Size**	**RAM**	**Ext. Mem.**	**Time**	**Size**
Vari(4C)	1.7 B	4,000	135 GB	10 h 53 m	0.46 GB	53 GB	2 h 34 m	51.8 GB	135 GB	1.6 TB	13 h 27 m	52 GB
Vari(4D)	2.4 B	4,000	137 GB	14 h 35 m	0.67 GB	59 GB	3 h 37 m	57.9 GB	137 GB	2.34 TB	18 h 12 m	59 GB
Merge(4C, 4D)	3.8 B	8,000	17 GB	2 h 59 m	1.00 GB	118 GB	57 m	107 GB	118 GB	0 TB	3 h 56 m	108 GB
Merge(8AB, 8CD)	5.8 B	16,000	25 GB	4 h 53 m	1.60 GB	254 GB	2 h 10 m	232 GB	254 GB	0 TB	7 h 3 m	233 GB
VariMerge	5.8 B	16,000	137 GB	54 h 47 m	1.60 GB	254 GB	14 h 21 m	232 GB	254 GB	2.34 TB	69 h 8 m	233 GB

Table 2. Breakdown of the peak memory, peak disk and time required by VariMerge to build the colored de Bruijn graph for 16,000 strains of Salmonella. We note VariMerge includes the resources required of the two 4,000 runs of Vari (i.e. Vari(4A) and Vari(4B)) and the merge run (i.e. Merge(4A, 4B)) from Table 1. No extra external memory is needed for merging so "0" is in the external memory column for Merge

Dataset	**No. of k-mers**	**Program**	**Output Size**	**Time**	**RAM**
4,000	1.1 Billion	Vari / Rainbowfish	51 GB	10 h 25 m	136 GB
		Bloom Filter Trie	99 GB	51 h 42 m	120 GB
		Multi-BRWT	1.3 TB	42 h 23 m	156 GB
		Mantis / Method of Almodaresi et al.	36 GB	5 h 58 m	313 GB
		VariMerge	51 GB	10 h 25 m	136 GB
8,000	2.4 Billion	Vari / Rainbowfish	114 GB	37 h 27 m	271 GB
		Bloom Filter Trie	N/A	N/A	N/A
		Multi-BRWT	N/A	N/A	N/A
		Mantis / Method of Almodaresi et al.	38 GB	13 h 37 m	370 GB
		VariMerge	106 GB	26 h 30 m	137 GB
16,000	5.8 Billion	Vari / Rainbowfish	N/A	N/A	N/A
		Bloom Filter Trie	N/A	N/A	N/A
		Multi-BRWT	N/A	N/A	N/A
		Mantis / Method of Almodaresi et al.	256 GB	36 h 12 m	316 GB
		VariMerge	233 GB	69 h 8 m	254 GB

Table 3. Comparison between space-efficient colored de Bruijn graph construction methods for 4,000, 8,000, and 16,000 Salmonella strains using VariMerge versus competing methods. We report N/A for any method that exceeded 140 CPU hours, 4 TB of disk space, and 750 GB of memory. We anticipate add-on methods to compress better but will still consume the resources shown for their base method because they reuse base the method's output. We measured RAM as Max Resident Set Size. Mantis authors noted their use of memory mapped I/O means this reveals opportunistic consumption and not necessarily requirement for their program. To the best of our knowledge, no extra external memory is needed for Bloom Filter Trie, Multi-BRWT, Mantis, and the method of Almodaresi et al. so it is omitted from the table.

because the latter method first runs Mantis and then compresses the output of Mantis. These points are also discussed in the related work. Again, because we are interested in construction (not exclusively compression) we are interested in peak RAM and external memory usage.

We report N/A in Table 3 for any method that exceeded 140 hours, and/or our RAM and external memory quota. We had 4 TB of disk space for each method (20 TB in total) and 750 GB of RAM. All methods performed reasonably well on 4,000 strains. Bloom Filter Trie required the most time, and Multi-BRWT required the most space. This was unsurprising as Multi-BRWT focuses on compression of the color matrix and not the graph (see Related Work). Mantis (and it's successor (Almodaresi *et al.*, 2019)) required the largest peak memory (313 GB). With 8,000 strains Bloom Filter Trie and Multi-BRWT required more than 140 hours, Vari and Rainbowfish were able to run within the given constraints but had the longest running time (over 37 hours). Mantis was more efficient with respect to final output (38 GB vs 106 GB) and VariMerge was more efficient with respect to RAM consumed (137 GB vs 370 GB). Lastly, only Mantis and VariMerge were capable of building the colored de Bruijn graph with 16,000 strains. For 16,000 strains our method had improved output size and RAM usage and Mantis had improved running time. Yet, as shown in the previous section, VariMerge allows the graph to be updated with new data via merging. Mantis (as well as Vari and Rainbowfish) provide no option to update the graph without reconstruction.

Lastly, VariMerge has several practical advantages over competing methods. It allows large values of k, i.e., $k \leq 64$, and allows for efficient queries of the form: given a color c, return all k-mers that have that color. Competing methods cannot perform such queries without the input files or cannot scale to large datasets. We discuss this more in the conclusions.

6 Conclusion

In this paper, we develop a method to build the colored de Bruijn graph by merging smaller graphs in a resource-efficient manner. This allows the colored de Bruijn graph to be constructed for large datasets and provides an efficient means to update it. As previously mentioned, resource use can be optimized by partitioning the data into smaller, carefully-selected datasets. We leave how to optimize the construction via data partitioning size as future work.

Lastly, we mention that the underlying graph data structure of VariMerge has some advantages over competing methods. First, VariMerge allows arbitrary k up to 64 while other tools are more restricted; BFT requires k to be multiples of 9 and Mantis only supports values of k up to 32. Second, the node labels can be recovered from the index alone, allowing the following queries to be performed: given a particular color c, what k-mers have color c. These queries can be accomplished by VariMerge by scanning a particular column of the color matrix and can be used for comparing samples, i.e., which k-mers are shared and which differ between sample x and sample y. While the Mantis authors have suggested an approach to supporting this query, it is unimplemented and requires the input files, increasing the size of the data structure.

Funding

This work National Science Foundation (NSF) IIS (Grant No. 1618814 to CB) and by the by National Institute of Allergy and Infectious Diseases Institute of the National Institutes of Health (Grant No. R01AI141810-01 to CB).

References

Almodaresi, F., Pandey, P., and Patro, R. (2017). Rainbowfish: A succinct colored de Bruijn graph representation. In *Proc. of WABI*, pages 251–265.

Almodaresi, F., Pandey, P., Ferdman, M., Johnson, R., and Patro, R. (2019). An efficient, scalable and exact representation of high-dimensional color information enabled via de Bruijn graph search. *Accepted to RECOMB 2019, Preliminary version posted on bioRxiv*.

Bowe, A., Onodera, T., Sadakane, K., and Shibuya, T. (2012). Succinct de Bruijn graphs. In *Proc. of WABI*, pages 225–235.

Bradley, P., Bakker, H., Rocha, E., McVean, G., and Iqbal, Z. (2017). Real-time search of all bacterial and viral genomic data. *BioRxiv*.

Burrows, M. and Wheeler, D. (1994). A block sorting lossless data compression algorithm. Technical Report 124, Digital Equipment Corporation, Palo Alto, California.

Chikhi, R. and Rizk, G. (2013). Space-efficient and exact de Bruijn graph representation based on a Bloom filter. *Algorith Mol Biol*, **8**(1).

Chikhi, R., Limasset, A., Jackman, S., Simpson, J. T., and Medvedev, P. (2014). On the representation of de Bruijn graphs. In *Proc. of RECOMB*, pages 35–55.

Conway, T. and Bromage, A. (2011). Succinct data structures for assembling large genomes. *Bioinformatics*, **27**(4), 479–486.

Crawford, V., Kuhnle, A., Boucher, C., Chikhi, R., and Gagie, T. (2019). Practical dynamic de bruijn graphs. *Bioinformatics*, page to appear.

Deorowicz, S., Kokot, M., Grabowski, S., and Debudaj-Grabysz, A. (2015). KMC 2: Fast and resource-frugal k-mer counting. *Bioinformatics*, **31**(10), 1569–1576.

Egidi, L., Louza, F., and Manzini, G. (2019). Space-efficient merging of succinct de bruijn graphs. arXiv:1902.02889.

Ferragina, P. and Manzini, G. (2005). Indexing compressed text. *JACM*, **52**(4), 552–581.

Holley, G., Wittler, R., and Stoye, J. (2016). Bloom filter trie–a data structure for pan-genome storage. *Algorithm Mol Biol*, **11**(3), 217–230.

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

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

Karasikov, M., Mustafa, H., Joudaki, A., Javadzadeh-No, S., Rätsch, G., and Kahles, A. (2019). Sparse binary relation representations for genome graph annotation. *Accepted to RECOMB 2019, Preliminary version posted on bioRxiv*.

Li, D., Liu, C.-M., Luo, R., Sadakane, K., and Lam, T.-W. (2015). MEGAHIT: An ultra-fast single-node solution for large and complex metagenomics assembly via succinct de Bruijn graph. *Bioinformatics*, **31**(10), 1674–1676.

Muggli, M., Bowe, A., Noyes, N., Morley, P., Belk, K., Raymond, R., Gagie, T., Puglisi, S., and Boucher, C. (2017). Succinct colored de bruijn graphs. *Bioinformatics*, **33**(20), 3181–3187.

Mustafa, H., Schilken, I., Karasikov, M., Eickhoff, C., Rätsch, G., and Kahles, A. (2019). Dynamic compression schemes for graph coloring. *Bioinformatics*, pages 407–414.

Okanohara, D. and Sadakane, K. (2007). Practical entropy-compressed rank/select dictionary. In *Proc. of ALENEX*, pages 60–70.

Pandey, P., Almodaresi, F., Bender, M., Ferdman, M., Johnson, R., and Patro, R. (2018). Mantis: A fast, small, and exact large-scale sequence-search index. *Cell*, **7**(2), 201–207.

Peng, Y. *et al.* (2012). IDBA-UD: A *de novo* assembler for single-cell and metagenomic sequencing data with highly uneven depth. *Bioinformatics*, **28**(11).

Simpson, J. *et al.* (2009). ABySS: A parallel assembler for short read sequence data. *Genome Research*, **19**(6), 1117–1123.

Sirén, J. (2016). Burrows-Wheeler transform for terabases. In *Proc. of DCC*, pages 211–220.

Solomon, B. and Kingsford, C. (2016). Fast search of thousands of short-read sequencing experiments. *Nature Biotechnology*.

Solomon, B. and Kingsford, C. (2018). Improved search of large transcriptomic sequencing databases using split sequence bloom trees. *Journal of Computational Biology*, **25**(7).

Stevens, E., Timme, R., Brown, E., Allard, M., Strain, E., Bunning, K., and Musser, S. (2017). The public health impact of a publically available, environmental database of microbial genomes. *Frontiers in Microbiology*, **8**, 808.

Sun, C., Harris, R., Chikhi, R., and Medvedev, P. (2017). AllSome Sequence Bloom Trees. In *Proc. of RECOMB*, pages 272–286.

The 1000 Genomes Project Consortium (2015). A global reference for human genetic variation. *Nature*, **526**, 68–74.

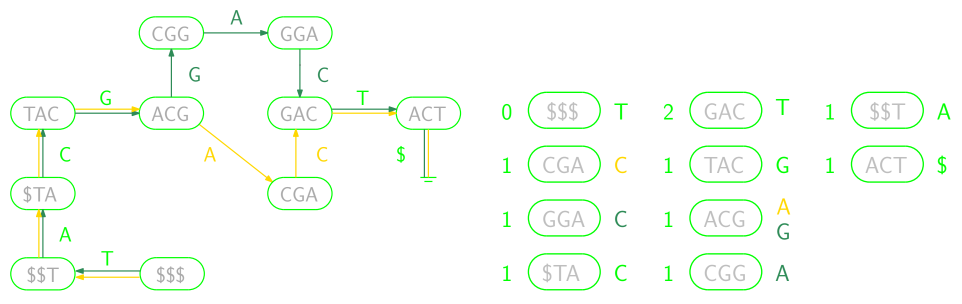
Turnbull, C. *et al.* (2018). The 100,000 genomes project: bringing whole genome sequencing to the nhs. *British Medical Journal*, **361**, k1687.

Yu, Y., Liu, J., Liu, X., Zhang, Y., Magner, E., Lehnert, E., Qian, C., and Liu, J. (2018). SeqOthello: querying RNA-seq experiments at scale. *Genome Biology*, **19**, 167.

Supplement

7 Illustration of Succinct Colored de Bruijn Graph

Below we illustrate the succinct colored de Bruijn graph data structure that was presented by Muggli *et al.* (2017) and is used in this paper.



$$\text{Edge-BWT}(G) = \texttt{TCCCTGAGAA\$}$$
$$\mathsf{B}_F = \texttt{11110111111}$$
$$\mathsf{B}_L = \texttt{11111101111}$$
$$C^{\mathrm{T}} = \begin{array}{l}\texttt{11011110011}\\ \texttt{10111101111}\end{array}$$

Fig. 2: **Left:** A colored de Bruijn graph consisting of two individual graphs, whose edges are shown in yellow and green. The nodes are present in either the yellow graph, the green graph, or both graphs and thus, are shown in lime. **Center:** The co-lexicographically sorted nodes, with the number of incoming edges shown on their left and the labels of the outgoing edges shown on their right. The edge labels are shown in yellow or green if the edges occur only in the respective graph, or lime if they occur in both. **Right:** The Vari representation of the colored de Bruijn graph: the edge-BWT is the list of edge labels. B_F and B_L are bit vectors to track the incoming and outgoing node degree, respectively. Finally, the binary array C (shown transposed) indicates which edges are present in which individual graphs.

8 k-mer Recovery

If we know the range $\mathsf{B}_L[i..j]$ of k-mers whose starting nodes end with a pattern P of length less than $(k-1)$, then we can compute the range $\mathsf{B}_F[i'..j']$ of k-mers whose ending nodes end with Pc, for any character c, as follows:

$$\begin{aligned} i' &= |\{d : d \in E,\ \mathsf{label}(d) \prec c\}| \\ &\quad +|\{h : \text{Edge-BWT}[h] = c,\ h < i\}| \\ j' &= |\{d : d \in E,\ \mathsf{label}(d) \prec c\}| \\ &\quad +|\{h : \text{Edge-BWT}[h] = c,\ h \leq j\}| - 1 . \end{aligned}$$

Thus, we can find the interval in B_L containing v's outgoing edges in $O(k \log\log \sigma)$-time, provided there is a directed path to v of length at least $k-1$. Thus, we add extra nodes and edges to the graph to ensure there is a directed path of length at least $k-1$ to each original node. More formally, we augment the graph so that each new node has a $(k-1)$-mer that is prefixed by one or more copies of a special symbol \$ not in the alphabet and lexicographically strictly less than all others. When new nodes are added, we are assured that the node with k-mer $\$^{k-1}$ is always first in colex order and has no incoming edges. Lastly, we augment the graph in a similar manner by adding an extra outgoing edge, labeled \$, to each node with no outgoing edge.

9 Pseudocode for the Naive Merge Algorithm

Below is a detailed sketch of the naive merge algorithm.

Algorithm 3 Naive Merge Algorithm. Because L_1 and L_2 are explicitly constructed, a large amount of memory is needed.

```
$\mathsf{L}_1 \leftarrow \emptyset$
$\mathsf{L}_2 \leftarrow \emptyset$
Populate $\mathsf{L}_1$ and $\mathsf{L}_2$ (See “$k$-mer Recovery” of Subsection 3)
Merge $\mathsf{L}_1$ and $\mathsf{L}_2$ into $\mathsf{L}_M$.
Create $\text{Edge-BWT}(G)_M$, $\mathsf{B}_{LM}$, $\mathsf{B}_{FM}$ from $\mathsf{L}_M$
Create $\mathsf{C}_M$ from $\mathsf{C}_1$ and $\mathsf{C}_2$
```

10 Illustration of merge plans

We provide an example of merge plans in two conceptual edge lists L_1 and L_2 from graphs G_1 and G_2 in Figure 3, where $k = 4$. As mentioned before the merge plan is defined as $P_1 = \{[0, p^1_1], \dots, [p^1_i, |\mathsf{L}_1| - 1]\}$ where each $p^1_1, \dots, p^1_i$ is an index in L_1, and $P_2 = \{[0, p^2_1], \dots, [p^2_i, |\mathsf{L}_2| - 1]\}$, where each $p^2_1, \dots, p^2_i$ is an index in L_2. Next, all revisions of P_1 and P_2 are computed iteratively. In this example, we start with the red merge plans P_1 and P_2 covering the whole L_1 and L_2 respectively. In the next iteration, based on the next letter (the order is right to left), we revise every interval into at most five subintervals (five being the number of alphabets: $\{\$, \mathtt{A}, \mathtt{C}, \mathtt{G}, \mathtt{T}\}$). In this example see how P_1 and P_2 in red are revised to P'_1 and P'_2 in green each with five intervals. Similarly, every interval in P'_1 and P'_2 are revised to at most five subintervals making P''_1 and P''_2.

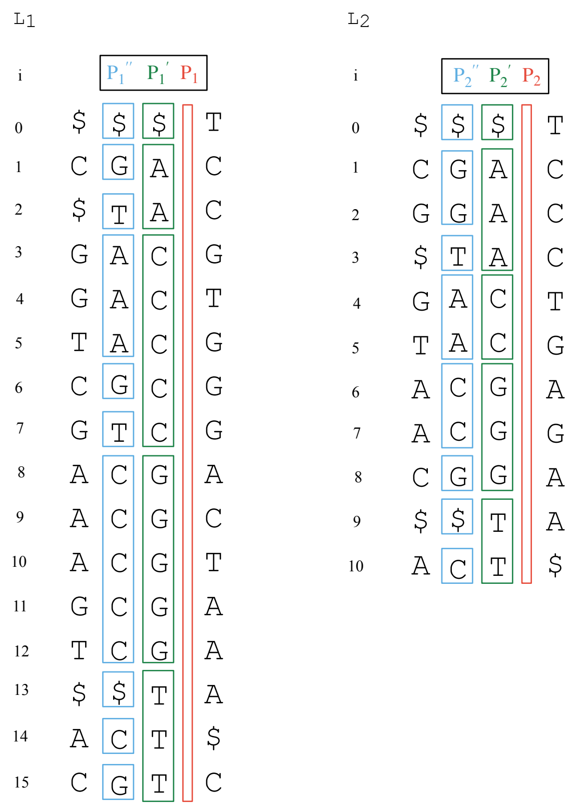


Fig. 3: Three merge plans. The initial merge plans are shown in red, where P_1 is initialized $\{[0, 15]\}$ and P_2 is initialized to $\{[0, 10]\}$. Green is the first refinement of P_1 and P_2. Thus, P'_1 and P'_2 consist of the continuous range of elements that have $, A, C, G, and T in the next character position, e.g. $P'_1 = \{[0, 0], [1, 2], [3, 7], [8, 12], [13, 15]\}$ and $P'_1 = \{[0, 0], [1, 3], [4, 5], [6, 8], [9, 10]\}$. Blue is the third merge plan which is based on the refinement of P'_1 and P'_2. P''_1 breaks each continuous range in P'_1 into five (possibly empty) continuous ranges based on the next character position. For example, the first continuous range of P'_1 is $\{[0, 0]\}$ broken into five ranges in P''_1, e.g., one for $, A, C, G, and T. Yet, four of these five ranges are empty since there exists only a single element with $ in the next character position.

11 Illustration of computing the next character of L

We provide an example of how we can navigate the set of edge k-mers without explicitly storing them. We compute the next character at each iteration with only three columns present in memory (shown with colors blue, red and orange).

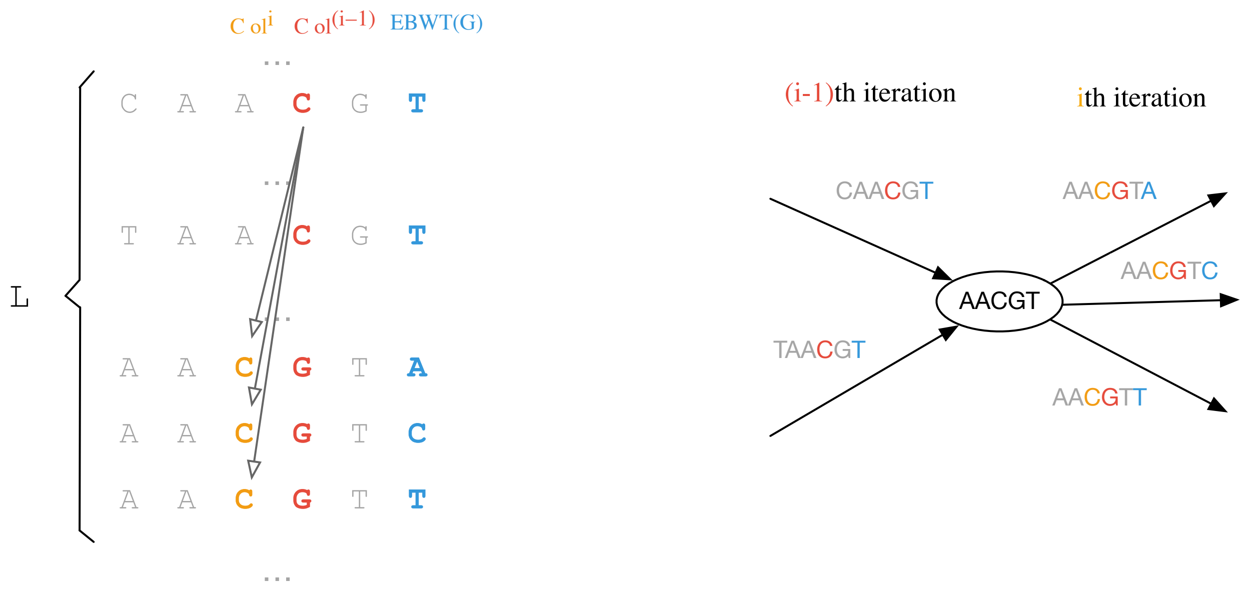


Fig. 4: Method for populating Col^i based on Col^{i-1} and graph navigation. Colored (blue, red and orange) nucleotides represent data that is in memory and valid. Grey represents data that is stored in external memory in Vari but is computed as needed and only exists ephemerally in VariMerge. Thus, only three columns are ever present in memory, which is a significant memory savings relative to the full set of edge k-mers. The three resident vectors are 1.) Edge-BWT(G) (which is always present and used for navigation), 2.) Col^{i-1} which is already completely populated when a new column to the left, 3.) (Col^i) is being generated. As one can see in the associated graph representation on right, the successor edges have almost the same k-mer but shifted by one position. e.g. the red colored C in $i-1$-th iteration in position 4 is shifted to position 3 and turned into orange in i-th iteration.

Algorithm 4 GetCol(i, $PrevCol$, G)

```
if $i = k$ then
    return $EBWT(G)$
else
    if $i = 1$ then
        $PrevCol \leftarrow EBWT(G)$
    end if
    $ResultCol \leftarrow []$
    for all $e \in \{1..|E|\}$ do
        $a \leftarrow \text{select}(\text{rank}(|\{d : d \in E, \text{label}(d) \prec \text{label}(e)\}| + 1, \mathsf{B}_{L_1}) + r - 1, \mathsf{B}_{L_1})$
        $b \leftarrow \text{select}(\text{rank}(|\{d : d \in E, \text{label}(d) \prec \text{label}(e)\}| + 1, \mathsf{B}_{L_1}) + r, \mathsf{B}_{L_1})]$
        for all $j \in \{a..b\}$ do
            $ResultCol[j] \leftarrow PrevCol[e]$
        end for
    end for
    return ResultCol
end if
```

12 Merging the secondary components

Delimiting common origin with B_{LM}. We prepare to produce B_{LM} in the planning step by preserving a copy of the merge plan after $k-1$ refinement iterations as S_{k-1}. After $k-1$ refinement steps, our plan will demarcate a pair of edge sets where their k-mers have identical $k-1$ prefixes. Thus, whichever merged elements in $\mathsf{Edge\text{-}BWT}(G)_M$ result from those demarcated edges will also share the same $k-1$ prefix. Therefore, while executing the primary merge plan, we also consider the elements covered by S_{k-1} concurrently, advancing a pointer into $\mathsf{Edge\text{-}BWT}(G)_1$ or $\mathsf{Edge\text{-}BWT}(G)_2$ every time we merge elements from them. We form B_{LM} by appending a delimiting 1 to B_{LM} (again, indicating the final edge originating at a node) whenever both pointers reach the end of an equal rank pair of intervals in S_{k-1}'s lists.

Delimiting common destination with flags_M. We produce flags in a similar fashion to B_{LM} but create a temporary copy of S_{k-2} in the planning stage after $k-2$ refinement iterations instead of $k-1$. In this case, the demarcated edges are not strictly those that share the same destination; only those edges that are demarcated and share the same final symbol. Thus, in addition to keeping pointers into $\mathsf{Edge\text{-}BWT}(G)_1$ or $\mathsf{Edge\text{-}BWT}(G)_2$, we also maintain a vector of counters which contain the number of characters for each (final) symbol that have been emitted in the output. We reset all counters to 0 when a pair of delimiters in S_{k-2} is encountered. Then, when we append a symbol onto $\mathsf{Edge\text{-}BWT}(G)_M$, we consult the counters to determine if it is the first edge in the demarcated range to end in that symbol. If so, we will not output a flag for the output symbol; otherwise, we will.

13 Proof of Theorem 1

Theorem 1. Given two colored de Bruijn graphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ constructed for k such that $|E_1| \geq |E_2|$. It follows that our merge algorithm constructs the merged colored de Bruijn graph G_M in $O(|E_1| \cdot \max(k,t))$-time, where t is the number of colors in G_M.

Proof. In our merge algorithm, we will perform k refinements of P_1 and P_2 after they are initialized. We know by definition and Observation 1 that $|P_1| \leq |\mathsf{L}_1|$, $P_2 \leq |\mathsf{L}_2|$, $Col_1^i = |\mathsf{L}_1| = |E_1|$ and $Col_2^i = |\mathsf{L}_2| \leq |E_1|$ at each iteration i of the algorithm. Further, it follows from Observation 1 that a constant number of operations are performed to P_1, P_2, Col_1^i and Col_2^i. Thus, the planning step will require $O(|E_1|k)$-time since there will k refines of P_1 and P_2, each of which has size at most $|E_1|$.

Executing the merge plan requires constructions of $\mathsf{Edge\text{-}BWT}$, B_F, B_L, and C_M from the merge plan. Construction of $\mathsf{Edge\text{-}BWT}$, B_F and B_L requires $O(|E_1|)$-time since a constant number of operations are needed for each item of the merge plan and there are at most $|E_1|$ items in P_1 and P_2. Next, in order to construct C_M, we perform a constant number of operations for each element of C_M. Since C_M is a binary matrix of size $2|E_1| \times t$ this will require $O(|E_1|t)$-time. Thus, the total merge algorithm requires $O(|E_1|k + |E_1|t)$-time which is equal to $O(|E_1| \cdot \max(k,t))$.

14 Details Concerning Strains of E. coli K-12

Table 4 describes the accession number, sub-strain and genome length of the E.coli genomes used for validation of our construction and bubble-calling.

Accession Number	Sub-strain	Genome Size
AP009048	W3110	4,646,332 bp
CP009789	ER3413	4,558,660 bp
CP010441	ER3445	4,607,634 bp
CP010442	ER3466	4,660,432 bp
CP010445	ER3435	4,682,086 bp
U00096	MG1655	4,641,652 bp

Table 4. Characteristics of the substrains of E. coli K-12 used to test the performance and accuracy of VariMerge

.

We validate VariMerge by generating two succinct colored de Bruijn graphs with three *E. coli* assemblies each, merge them, and verify the correctness of the merged graph. First, we generated all k-mers for each reference genome, counted all unique k-mers with KMC2 Deorowicz *et al.* (2015), constructed two de Bruijn graphs of three assemblies each using Vari, and merged them into a six-color graph using VariMerge. Independently, we constructed a second colored de Bruijn graph using Vari on all six assemblies in one run and then compared these two graphs. We found VariMerge produced files on disk that were bit-for-bit identical to those generated by Vari, demonstrating they construct equivalent graphs and data structures.