Succinct De Bruijn Graph Construction for Massive Populations Through Space-Efficient Merging

Martin D. Muggli[1] and Christina Boucher[2]

[1] Department of Computer Science, Colorado State University, Fort Collins, CO
martin.muggli@colostate.edu

[2] Department of Computer and Information Science and Engineering, University of Florida, Gainesville, FL
christinaboucher@ufl.edu

Abstract. Recently, there has been significant amount of effort in developing space-efficient and succinct data structures for storing and building the traditional de Bruijn graph and its variants, including the colored de Bruijn graph. However, a problem not yet considered is developing a means to merge succinct representations of the de Bruijn graph—a challenge is necessary for constructing the de Bruijn graph on very-large datasets. We create VariMerge, for building the colored de Bruijn graph on a very-large dataset through partitioning the data into smaller subsets, building the colored de Bruijn graph using a FM-index based representation, and merging these representations in an iterative format. This last step is an algorithmic challenge for which we present an algorithm in this paper. Lastly, we demonstrate the utility of VariMerge by demonstrating: a four-fold reduction in working space when constructing an 8,000 color dataset, and the construction of population graph two orders of magnitude larger than previous reported methods.

1 Introduction

Approximately one million people every year suffer from foodborne illness attributed to Salmonella according to the US Center for Disease Control (CDC). When an outbreak occurs there is a desire to identify the source in order to intervene in the outbreak. A lab method known as Pulse Flow Gel Electrophoresis (PFGE) has been traditionally used to capture identifying characteristics of the pathogen which are then compared to a database known as PulseNet to narrow down its possible source in outbreak investigation [24, 4]. In recent years, there has been an initiative to move toward using whole genome sequencing (WGS) to augment or replace PFGE in the hopes of delivering more accurate identification [6]. This has led to the existence and ongoing development of GenomeTrakr, which is a large public effort to use genome sequencing for surveillance and detection of outbreaks of foodborne illnesses. Currently, there are over 50,000 samples spanning several species available through this initiative—a number that continues to rise as datasets are continually added [28]. Unfortunately, methods to analyze the complete dataset given its current size are limited. Existing methods for using this WGS data frequently focus on Multi Locus Sequence Typing (MLST) [23], a method that aligns reads to genes from a reference genome. These alignments identify sets of alleles and thus, are limited to capturing genetic variations that are shorter than the length of reads and only those variations that align to a reference genome or gene set [16].

Given the limitations of existing methods, we would like to apply advanced methods for finding variants—such as Cortex [14] and Vari [20]—which are able to detect complex variants without a reference. These methods use a variant of the de Bruijn graph referred to as the colored de Bruijn graph. We define the *de Bruijn graph* constructively as follows: an edge is created for every unique k-length subsequence (*k-mer*) in the data and is labeled with the prefix and suffix of that k-mer, and after all edges have been created and labeled, the nodes that have the same label are merged to a single vertex. Thus, we create a *colored de Bruijn graph* by adding a set of colors (or labels) to each vertex (and each edge) indicating which samples contain the respective $(k-1)$-mer (k-mer). Next, we can traverse these colored graphs to find genetic variation between the samples. Unfortunately, Vari [20] and Rainbowfish [1], which construct the colored de Bruijn graph in a memory-efficient manner, are unable to scale in a manner that is necessary for large datasets, such as GenomeTrakr. Principally, both tools manipulate uncompressed data in external memory, which while cheaper than RAM can still be limited and is much slower than RAM. Thus, one way to accomplish the construction of the colored de Bruijn graph on a massive dataset is to use a divide-and-conquer approach: divide the data into smaller partitions, construct the colored de Bruijn graph for each partition, and merge the (smaller) colored de Bruijn graph until a single graph remains.

In this paper, we present VariMerge, which builds a colored de Bruijn graph by partitioning and merging smaller colored de Bruijn graphs stored using the FM-index [3]. This succinct representation stores an abbreviated form of traditional edge labels (e.g. only the single final base instead of k bases for each edge) in a sorted order. Our method exploits this fact. Thus, briefly, VariMerge works as follows: (1) we reconstruct all edge label positions ephemerally (one base position at a time) in decreasing precedence order, and while doing so, we incrementally refine a data structure that captures the inferable rank and equivalence of full labels based on the portion of the labels our algorithm has seen. After we have integrated the information from all positions of the labels, we use the rank and equivalence data to merge the abbreviated edge labels as well as to merge the companion color annotation matrices. Hence, we demonstrate that VariMerge takes $O(m \max(k,t))$-time, where m is the edges, k is the k-mer value, and t is the number of colors.

Our contributions. We develop an efficient algorithm for succinctly merging colored de Bruijn graphs, and demonstrate its efficiency in producing a massive colored de Bruijn graph from a

collection of smaller ones. In particular, we build a de Bruijn graph using 16,000 strains of salmonella which were collected and housed at NCBI as part of the GenomeTrakr database. Hence, we presents the very first large-scale assembly based modeling of the GenomeTrakr data since previous analysis of this data were limited to only [23] analyzing reads in isolation. Further, to the best of our knowledge, this is the largest dataset for which the (colored) de Bruijn graph has been constructed. The most recent large-scale construction is due to Holley et al. [13], which presents a de Bruijn graph construction for 473 clinical isolates of *Pseudomonas aeruginosa* (NCBI BioProject PRJEB5438). Lastly, we compare VariMerge with the Bloom Filter Trie on over 2,000 Salmonella Isolates and find the size of the VARI representation grows more slowly with this population. Our method is available at `https://github.com/cosmo-team/cosmo/tree/VARI-merge`.

2 Related Work

Space-efficient representations of de Bruijn graphs have thus been heavily researched in recent years. One of the first approaches, introduced with the creation of the ABySS assembler, stores the graph as a distributed hash table [25]. In 2011, Conway and Bromage [9] reduced these space requirements by using a sparse bitvector (by Okanohara and Sadakane [21]) to represent the k-mers, and used rank and select operations (to be described shortly) to traverse it. Minia [8] uses a Bloom filter to store 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, Onodera, Sadakane and Shibuya [3] developed a succinct data structure based on the Burrows-Wheeler transform (BWT) [5]. This data structure is discussed in more detail in the next section. This data structure of Bowe et al. [3] is combined with ideas from IDBA-UD [22] in a metagenomics assembler called MEGAHIT [15]. Chikhi et al. [7] implemented the de Bruijn graph using an FM-index and *minimizers*.

More recently, methods have been developed to store de Bruijn graphs for a population which entails an additional space burden in tracking which samples contribute to graph elements. Holley et. al. [13] released the Bloom Filter Trie, which is another succinct data structure for the colored de Bruijn graph. SplitMEM [18] is a related algorithm to create a colored de Bruijn graph from a set of suffix trees representing the other genomes. Lastly, Vari [20] and Rainbowfish [1] are both memory-efficient data structures for storing the colored de Bruijn graph. Both are discussed later in this paper.

The closest related work to that proposed here concerns other reduced memory colored de Bruijn graphs with efficient construction. SplitMEM [18] uses suffix trees to directly construct the compacted de Bruijn graph, where non-branching paths become single nodes. Here, we use the term *compacted* to distinguish this approach from data compression techniques underlying succinct data structures. Baier et al. [2] improved on this method with two alternative construction methods, using the compressed suffix tree and using BWT. TwoPaCo [19] uses a bloom filter to represent the ordinary de Bruijn graph and then constructs the compacted de Bruijn graph from the bloom filter encoded one. Bloom filter tries, proposed by Holley et al. [13] encode 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. Rainbowfish [1] also stores distinct sets of colors in a table and uses Huffman-like variable length bit patterns to reference color sets from each edge in the succinct de Bruijn graph. Both the Bloom filter trie method and Rainbowfish are able to collapse redundant color sets across the entire graph to a single instance instead of just along non-branching paths in the compacted graph methods.

Although Rainbowfish can store the colored de Bruijn graph in less memory, it still depends on VARI, so is limited to VARI's construction capacity.

Lastly, BWT-Merge by Sirén [26] is also 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 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.

3 Preliminaries

As previously mentioned, in 2017 Muggli et al. [20] presented Vari which is a representation of the colored de Bruijn graph using BWT. VariMerge efficiently merges de Bruijn graphs that are represented in this manner. Thus, we briefly give an overview of this data structure but we refer the reader to the full paper for a more detailed discussion. Further, we refer the reader to Section 7 in the supplement for a summary of BWT.

3.1 Storage of de Bruijn Graph using BWT

Given a de Bruijn graph $G = (V, E)$, we refer the edge label of an edge $e \in E$ as the k-mer corresponding to it, and denote it as $\mathsf{label}(e)$. Further, given V, we define the co-lexicographic (colex) ordering of V as the lexicographic order of their reversed labels ($(k-1)$-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 refer to the ordering of L as *Vari-sorted*. If we are given two edges e and e' have the same label then we are guaranteed that 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

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

where $\prec$ denotes lexicographic precedence. We define the edge-BWT (EBWT) of G to be the sequence of edge labels sorted according to the ordering of the edges in L, so $\mathsf{label}(\mathsf{L}[h]) = \mathsf{EBWT}(G)[h]$ for all h. Therefore, if we have an array storing $|\{d : d \in E,\ \mathsf{label}(d) \prec c\}|$ for each character c and a fast rank data structure on $\mathsf{EBWT}(G)$ then given an edge's position in L, we can quickly compute its position in F.

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. In addition, if B_G is the bit vector with a 0 marking the position in F of the first incoming edge of each node, we define flags to be the vector that stores the permutation from F to L applied to B_G. Next, we see how we use flags. If we are given a character c and the colex rank of a node v, we can use B_L to find the interval in L containing v's outgoing edges, then we can search in $\mathsf{EBWT}(G)$ to find the position of the one e labeled c. We need flags in order to obtain the colex rank of v. Thus, we can find e's position in F, as described above. Finally, we can use B_F to find the co-lexicographic rank of e's ending node.

Therefore, briefly we explained how we can construct and represent a de Bruijn graph $G = (V, E)$ with EBWT, B_L, and flags in a manner that allows for efficient navigation of the graph. Next, we demonstrate how the labels (k-mers) can be recovered using this data structure.

3.2 Label Recovery

We note that an important aspect of this succinct representation of the graph is that the labels of the nodes and edges are not explicitly stored but can still be computed. Here, we briefly describe the procedure for recovering these labels. This method could be used for a trivial merge algorithm that we discuss later.

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

$$i' = |\{d : d \in E,\ \mathsf{label}(d) \prec c\}| + |\{h : \mathsf{EBWT}(G)[h] = c,\ h < i\}|$$
$$j' = |\{d : d \in E,\ \mathsf{label}(d) \prec c\}| + |\{h : \mathsf{EBWT}(G)[h] = c,\ h \leq j\}| - 1\,.$$

Thus, we can find the interval in L containing v's outgoing edges in $Ok \log\log\sigma$-time, provided there is a directed path to v of length at least $k-1$ but note that we cannot use $\mathsf{EBWT}(G)$, B_F and B_L alone to recover the labels of nodes with no incoming edges. 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's label is 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 labeled $\$^{k-1}$ 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.

3.3 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 (i.e., whether that edge has the jth color). Hence, we store a given de Bruijn graph using EBWT, the described bit vectors, and a compressed color matrix.

4 Methods

We describe this procedure for merging colored de Bruijn graphs that are stored succinctly using the Vari data structure. We describe how to merge two colored de Bruijn graph 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 $\mathsf{EBWT}(G)_1, \mathsf{B}_{L1}, \mathsf{flags}_1, \mathsf{D}_1, \mathsf{C}_1$ and $\mathsf{EBWT}(G)_2, \mathsf{B}_{L2}, \mathsf{flags}_2, \mathsf{D}_2, \mathsf{C}_2$, respectively. We will output the merged graph $G_M = (V_M, E_M)$ stored in the same format as the input, more descriptively: a set of abbreviated edge labels $\mathsf{EBWT}(G)_M$, a bit vector that delimits their common origins B_{LM}, a bit vector that encodes their common destinations flags_M, the array D_M storing $|\{h : S[h] \prec c\}|$ for each character c, and the color matrix C_M.

4.1 A Naive Merge Algorithm

We begin by describing a naive merge procedure to motivate the use of the succinct merge algorithm, and define the operations that need to be accomplished. We recall from Section 3 that Vari stores the edge labels (k-mers) L by storing the last nucleotide in EBWT, and that L is not explicitly stored. For example, if we want to reconstruct the k-mer `AGAGAGTTA` contained in G_1 which stored

as A in $\mathsf{EBWT}(G)_1$, we need to backward navigate in G_1 from the edge labeled A through $k-1$ predecessor edges (T, T, G,...). We concatenate the abbreviated edge labels encountered during this backward navigation in reverse order to construct the label AGAGAGTTA. Thus, we could naively merge two G_1 and G_2 by reconstructing L_1 and L_2, merging them into L_M and abbreviating L_M to produce $\mathsf{EBWT}(G)_M$. We note that this algorithm requires explicitly building L_1, L_2 and L_M and thus, has a significant memory footprint. Therefore, the challenge we face is producing $\mathsf{EBWT}(G)_M$ in a manner that avoids constructing these full edge labels.

4.2 Overview of Succinct Merge Algorithm

Next, we describe the merge algorithm used by VariMerge. We refer to $\mathsf{EBWT}(G)_M$ and C_M as the *primary components* of the data structure—as this is where the nodes and edges of the graph are stored—and remaining elements as *secondary elements*. Briefly, we create primary components by copying elements in $\mathsf{EBWT}(G)_1$, $\mathsf{EBWT}(G)_2$, C_1, and C_2 based on their corresponding (non-abbreviated) edge labels in L_1 and L_2. We perform this copying step in a careful manner that ensures identical elements between both L_1 and L_2 are collapsed, and thus, G_M is the union of G_1 and G_2. We construct the secondary elements using intermediate data structures arising in the aforementioned task of generating $\mathsf{EBWT}(G)_M$. In the remainder of this section, we describe the details of merging the primary components and the secondary components.

4.3 Merging Primary Components

We merge the primary components by performing two steps: planning and executing. After performing the planning step, we output a list of non-overlapping intervals for L_1 and a list for L_2 which corresponds to a *merge plan* that we execute in the second step. Hence, we denote the merge plan as $P_1 = \{[0, p_1^1], ..., [p_i^1, |\mathsf{L}_1|]\}$ where each $p_1^1, .., p_i^1$ is an index in L_1, and $P_2 = \{[0, p_1^2], ..., [p_i^2, |\mathsf{L}_2|]\}$, where each $p_1^2, .., p_i^2$ is an index in L_2.

The Planning Step. We first initialize P_1 and P_2 to be single intervals covering L_1 and L_2, respectively (e.g. $P_1 = \{[0, |\mathsf{L}_1|]\}$ and $P_2 = \{[0, |\mathsf{L}_2|]\}$). Next, we revise P_1 and P_2 in an iterative manner. In particular, we perform k consecutive revisions of P_1 and P_2, where k is the k-mer value used to construct G_1 and G_2—each revision of P_1 and P_2 is based on a small fraction of L_1 and L_2, i.e., a single character of each edge (k-mer). Thus, in order to fully describe the planning stage, we define 1.) how the fractions of L_1 and L_2 are computed, and 2.) how P_1 and P_2 are revised based on these fractions. We also refer to the reader the pseudocode found in Algorithm 1 in the supplement.

Computing the fraction of L_1 *and* L_2. We let i denote the current iteration of our revision of P_1 and P_2, where $1 \leq i < k$. We compute the fraction of L_1 and L_2 using two temporary character vectors Col_1^i and Col_2^i, which are of length $|\mathsf{L}_1|$ and $|\mathsf{L}_2|$, respectively. Conceptually, we can define these vectors as follows: $Col_1^i[j] = \mathsf{L}_1[j][k-i]$ if $j < k$ 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 $Col_2^i[j] = \mathsf{L}_2[j][k]$. However, we remind the reader that this is a conceptual view because we do not explicitly store L_1 and L_2 (the edge labels) and thus, we must compute Col_1^i (Col_2^i).

We recall a couple artifacts about the Vari data structure prior to describing how we compute Col_1^i (and Col_2^i). We first note that Col_1^{i-1} contains the $(q+1)$-th position of every edge label, and after computation, Col_1^i will contain the q-th position of every edge label. Hence, we consider the characters in the label from right to left (i.e. decreasing sort precedence). Fortunately, we have the final character of each edge label stored in $\mathsf{EBWT}(G)_1$ to begin—and thus, we start by computing

the second to final character ($(k-1)$-th position) and consider the characters in the decremented position at each iteration. Second, we note that given any edge $e_{pred} = (s_{pred}, t_{pred})$ in G_1 and the $(q+1)$-th character c of the label of e_{pred}, all outgoing edges of t_{pred}, say $e^1_{succ}, .., e^n_{succ}$, have c in the q-th position of their edge labels. This follows from the fact that G_1 is an de Bruijn graph. Thus, we can compute $e^1_{succ}, .., e^n_{succ}$ by first performing a query of rank of e_{pred} ($r = \mathsf{rank}(e_{pred}, \mathsf{EBWT}(G)_1)$) in order to identify t_{pred}, and then determining the appropriate range in $\mathsf{EBWT}(G)_1$ in order to find all outgoing edges of t_{pred}. Given that the edges are in colex order of their $k-1$ prefix, we know all outgoing edges of t_{pred} will be in a contiguous range in $\mathsf{EBWT}(G)_1$ and in the same relative order as their immediate predecessor edges. We find this range in $\mathsf{EBWT}(G)_1$ by computing $\mathsf{select}(\mathsf{rank}(\mathsf{D}_1[e_{pred})] + 1, \mathsf{B}_{L_1}) + r - 1, \mathsf{B}_{L_1}), \mathsf{select}(\mathsf{rank}(\mathsf{D}_1[e_{pred})] + 1, \mathsf{B}_{L_1}) + r, \mathsf{B}_{L_1})]$. We use both these facts in our computation of Col^i_1 (See Figure 2 in the supplement).

We define the computation of Col^i_1 by describing the following three cases. When $1 < i < k$, we compute Col^i_1 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^{i-1}_1) into q-th position of all outgoing edges of that node. When $i = 1$, the $(q+1)$-th position corresponds to $\mathsf{EBWT}(G)_1$, so $\mathsf{EBWT}(G)_1$ is used in place of Col^{i-1}_1 but is otherwise identical to the previous case. Lastly, when $i = k$, we let Col^i_1 equal $\mathsf{EBWT}(G)_1$.

We compute Col^i_2 in an analogous manner.

Revising P_1 and P_2. We revise P_1 and P_2 based on Col^i_1 and Col^i_2 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 $n = 1, .., |P_1|$, 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 $SubP_1$ and $SubP_2$ in order to divide $P_1[n]$ and $P_2[n]$ based on the runs of covered characters in Col^i_1 and Col^i_2—e.g., for each run of `A`, `C`, `G`, `T` or `$` (See Figure 1 in the supplement). Next, we formally define this computation.

Thus, we partition P_1 by first computing the subvector of Col^i_1 that is covered by $P_1[n]$, which we denote this as $Col^i_1(P_1[n])$, and computing the subvector of Col^i_2 that is covered by $P_2[n]$, which we denote this as $Col^i_2(P_2[n])$. Next, given a character c in $\{\$, \mathtt{A}, \mathtt{C}, \mathtt{G}, \mathtt{T}\}$, we populate $SubP_1[c]$ and $SubP_2[c]$ based on $Col^i_1(P_1[n])$ *and* $Col^i_2(P_2[n])$ as follows: (1) we check whether c exists in either $Col^i_1(P_1[n])$ or $Col^i_2(P_1[n])$; (2) if so, we add an interval to $SubP_1[c]$ covering the contiguous range of c in $Col^i_1(P_1[n])$ (or add an empty interval if $Col^i_1(P_1[n])$ lacks any instances of c), and add an interval to $SubP_2[c]$ covering the contiguous range of c in $Col^i_2(P_1[n])$ (or add an empty interval if $Col^i_2(P_1[n])$ lacks any instances of c)[3]. 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. We refer the reader to Algorithm 1 in the supplement for the pseudocode.

We crafted the method above to maintain the property described in the following observation. The first condition states that the size of the two lists of intervals remains the same. The second condition implies that if two elements are in separate intervals in P_1 or P_2 then they remain in separate intervals after they are revised. Lastly, the third condition implies that the revision of P_1 and P_2 will divide any interval into at most five subintervals. Hence, this observation implies that at each iteration of the algorithm will further partition the elements of L_1 and L_2 relative to the previously processed $(q+1)$ position.

Observation 1 *Let P_1 be a (partial) merge plan, and P'_1 its refinement by* VariMerge*, 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 $|P'_1| = |P'_2|$;*

[3] We are guaranteed by the definition of our data structure that any instances of c in $Col^i_1(P_1[n])$ will be in a contiguous range, and likewise, any instances of c in $Col^i_2(P_1[n])$ will also be in a contiguous range

(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.

The Executing Step. We execute the merge plan by combining the elements of $\mathsf{EBWT}(G)_1$ that are covered by an interval in P_1 with the elements of $\mathsf{EBWT}(G)_2$ that are covered by the equal position interval in P_2 into a single element in $\mathsf{EBWT}(G)_M$. We note that when all characters of each label 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 $|\mathsf{EBWT}(G)_M|$. 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 $\mathsf{EBWT}(G)_1$ onto the end of $\mathsf{EBWT}(G)_M$ if $|p_i^1| = 1$. If $|p_i^2| = 1$ then we dismiss the next character of $\mathsf{EBWT}(G)_2$ since it is an abreviated form of an identical edge to that just added. Next, if $|p_i^1| = 0$ and $|p_i^2| = 1$, we copy the next character from $\mathsf{EBWT}(G)_2$ onto the end of $\mathsf{EBWT}(G)_M$. We refer the reader to Algorithm 2 for the pseudocode in the supplement.

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 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.

4.4 Merging Secondary Components

Next, we present our method for generating the secondary components of the succinct data structure for G_M.

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 labels have identical $k-1$ prefixes. Thus, whichever merged elements in $\mathsf{EBWT}(G)_M$ 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 $\mathsf{EBWT}(G)_1$ or $\mathsf{EBWT}(G)_2$ 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. Otherwise, we append a 0 (indicating more edges with a common origin follow in $\mathsf{EBWT}(G_M)$) for each element added to $\mathsf{EBWT}(G)_M$.

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 cases, 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 $\mathsf{EBWT}(G)_1$ or $\mathsf{EBWT}(G)_2$, 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 $\mathsf{EBWT}(G)_M$, 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.

Enabling navigation with D_M. We produce D_M using the merge plan after the first refinement iteration. The intervals at this point are identical to that encoded in D_1 and D_2 so we use the latter rather than consume more space with another copy of an intermediate merge plan. Then, like for B_L we increment a variable tracking position within the intervals in parallel with consuming elements from $\mathsf{EBWT}(G)_1$ or $\mathsf{EBWT}(G)_2$. We count the number of emitted characters while consuming elements from each of the $\leq \sigma$ intervals and emit a prefix sum of these counts as D_M.

4.5 Computational Complexity

Theorem 2. *Given two de Bruijn graphs $G_1 = (V_1, E_2)$ and $G_2 = (V_2, E_2)$ constructed with integral value k such that, without loss of generality, $|E_1| \geq |E_2|$, it follows that* VariMerge *constructs the merged de Bruijn graph G_M in $O(m \cdot \max(k, t))$-time, where t is the number of colors (columns) in C_M and $m = |E_1|$.*

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 $|P_1| \leq |\mathsf{L}_1|$, $P_2 \leq |\mathsf{L}_2|$, $Col_1^i \leq |\mathsf{L}_1|$ and $Col_2^i \leq |\mathsf{L}_2|$ at each iteration i of the algorithm. Further, it follows from Observation 1 a constant number of operations are performed to P_1, P_2, Col_1^i and Col_2^i. We populate C_M in the last step of merging the primary components of the data structure. Since the C_M is a bit matrix of size k by t, it follows that this step will take time $O(m \max(k, t)$-time. Hence, if $k \leq t$ the merge algorithm will take $O(mk)$-time; otherwise it will take $O(mt)$-time (since populating C_M will dominate in this case).

5 Results

In this section, we present our experimental results on *E. coli* and GenomeTrakr data. We show the scalability of VariMerge by demonstrating the time and computational resources needed to build the colored de Bruijn graph for 16,000 strains of salmonella. Next, in order to validate the correctness of our approach, we generated two succinct colored de Bruijn graphs with sets of three *E. coli* assemblies each, merged them, and verified its equivalence to a six color graph built from scratch. This experiment demonstrates that the merged colored de Bruijn graph is equivalent to that produced by building the graph without merging. 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 Large-scale Construction using GenomeTrakr

We demonstrate the scalability of VariMerge by constructing the succinct de Bruijn graph for 16,000 salmonella strains in NCBI BioProject PRJNA183844. We downloaded the sequence data from NCBI and preprocessed the data by assembling each individual sample with IDBA-UD, and counting k-mers (k=32) using KMC. We used these k-mers as input to VariMerge. We modified IDBA by setting kMaxShortSequence to 1,024 per public advice from the author to accommodate longer paired ends reads that modern sequencers produce. We sorted the full set of samples by the size of their k-mer counts and selected 16,000 samples about the media. This avoids exceptionally short assemblies, which may be due to low read coverage, and exceptionally long assemblies which may be due to contamination. We divide these 16,000 samples into four sets of 4,000 which we label 4k-1, 4k-2, 4k-3, and 4k-4. The exact accessions for each dataset is available in our repository. Merged graphs are numbered in the order of their constituents, that is the merged 8k-1 comprises the graphs from 4k-1 and 4k-2. We summarize our results in Table 1.

In order to measure the effectiveness of VariMerge for incremental additions to a graph that holds a growing population of genomes, we constructed the colored de Bruijn graph using Vari for a set of 4,000 salmonella assemblies as well as for a set of just one assembly. Next, we ran VariMerge on these two graphs. Vari took 8 hours 46 minutes, 1 TB of external memory, and 136 GB of RAM to build the graph for 4,000 strains. To build a single colored de Bruijn graph for an additional strain, Vari took 27 seconds, 10 GB of external memory, and 3 GB of RAM. To merge these graphs, VariMerge took 49 minutes, no external memory, and 5 GB of RAM.

In order to measure the effectiveness of VariMerge for the proposed divide-and-conquer method of building large graphs, we built a graph for a second set of 4,000 assemblies using 10 hours 40 minutes, 1.5 TB of external memory, and 137 GB of RAM. We merged these two 4,000 sample graphs using VariMerge in 2 hours 1 minutes, no external memory, and 10 GB of RAM. Even considering the cost of producing the two 4,000 color graphs, the total compares favorably with building the equivalent 8,000 color graph. Doing so required 30 hours 49 minutes, 4.6 TB of external memory and 271 GB of RAM when we used VARI to build the 8,000 color graph from scratch.

We further used this facility to merge two more 4,000 color graphs (i.e. 4k-3 + 4k-4 = 8k-2) and then merged this 8,000 sample graph with the aformentioned 8,000 graph to produce a succinct colored de bruijn graph of 16,000 samples (i.e. 8k-1 + 8k-2 = 16k-1).

Both VARI and VariMerge have their color matrix production implementation as separate programs which we denote with a -C suffix. VariMerge-C also compares favorably with Vari-C on runtime for producing compressed color matrices since Vari took 1 hour 39 minutes and 53 GB of RAM to compress the color matrix on the first 4,000 sample set (4k-1), and took 2 hours 22 minutes and 54 GB of RAM to compress the color matrix for the second dataset(4k-2). VariMerge-C took 1 hour 3 minute to merge the two compressed color matrices. We note that our implementation benefits from the SDSL-Lite library by Gog *et al.* [12], but unfortunately it currently requires us to load the two source color matrices into memory while producing the merged matrix in memory. Algorithmically, these are accessed sequentially and could be streamed to/from disk, only requiring constant RAM. Merging these two color matrices thus required 224 GB of RAM. Merging two 8,000 color matrices required 2 hours 8 minutes and 489 GB of RAM.

Program and Dataset	**Total Unique k-mers**	**Total Colors**	**RAM**	**Ext. Mem.**	**Time**	**Output Size**
VARI(4k-1)	1.1 B	4,000	136 GB	1 TB	8 h 46 m	0.308 GB
VARI(4k-2)	1.5 B	4,000	137 GB	1.5 TB	10 h 40 m	0.516 GB
VARI(8k-1)	2.4 B	8,000	271 GB	4.6 TB	30 h 49 m	0.627 GB
VariMerge(4k-1, 4k-2)	2.4 B	8,000	10 GB	N/A	2 h 1 m	0.628 GB
VariMerge(4k-2, 4k-3)	3.8 B	8,000	17 GB	N/A	2 h 59 m	1.00 GB
VariMerge(8k-1, 8k-2)	5.8 B	16,000	25 GB	N/A	4 h 53 m	1.60 GB
VARI-C(4k-1)	1.1 B	4,000	52 GB	N/A	1 h 39 m	51.2 GB
VARI-C(4k-2)	1.5 B	4,000	54 GB	N/A	2 h 22 m	52.5 GB
VARI-C(8k-1)	2.4 B	8,000	117 GB	N/A	6 h 28 m	106 GB
VariMerge-C(4k-1, 4k-2)	2.4 B	8,000	224 GB	N/A	1 h 3 m	106 GB
VariMerge-C(4k-3, 4k-4)	3.8 B	8,000	231 GB	N/A	1 h 6 m	107 GB
VariMerge-C(8k-1, 8k-2)	5.8 B	16,000	489 GB	N/A	2 h 8 m	232 GB

Table 1: Performance statistics for VARI and VariMerge. We separately list the succinct de Bruijn graph producing programs (labeled VARI and VariMerge) from the color annotation matrix producing programs (labeled VARI-C and VariMerge-C). The total size of a succinct colored de Bruijn graph is comprised of the space for the succinct de Bruijn graph plus the space for the color matrix.

5.2 Comparison to Bloom Filter Trie

In addition to demonstrating scalability, we used the Samonella strains from GenomeTrakr to directly compare data structure space of VariMerge with Bloom Filter Trie [13]. We found super-linear growth in space consumption (see Figure 3 in the supplement), producing a graph 40 GB in size after 2,000 samples were inserted. This is only slightly less space than the 50 GB the VARI data structure requires to represent twice as many samples. Holley et al. [13] report sub-linear growth up through 471 samples; however, we posit these differing observations may be a result of both differing dataset and preprocessing methods. More specifically, these isolates were extracted from a single species (humans) in contrast to GenomeTrakr, and thus may result in data that is more more homogenous and has slower growth in the diversity with population size; GenomeTrakr Salmonella samples are culled from diverse food production environments. Furthermore, they filter k-mers that have low multiplicity as a means to clean the data. This may reduce the growth as parts of the so called core genome may be missing in some samples, and the set of population k-mers could converge asymptotically toward the core genome. We do not compare to Sequence Bloom Trees because they are designed for transcript querying rather than variant detection.

5.3 Validation using E. coli

We validate VariMerge by generating two succinct colored de Bruijn graphs with three *E. coli* assemblies each, merge them, and verify equivalence. In particular, we verified equivalence in two ways. First, we generated all k-mers for each reference genome, counted all unique k-mers with KMC2, constructed two de Bruijn graphs using Vari, and merged them using VariMerge. In addition, we constructed a second colored de Bruijn graph using Vari with this same dataset without merging, and 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.

Second, we partition each reference genome, leading to a de Bruin graph in which some nodes have no successors and others no predecessors. We recall Vari inserts extra edges for these conditions; however, some extra edges may become obsolete when graphs are merged. In this situation, we expect the merged graph to have these additional obsolete extra edges relative to a graph built by Vari from the complete set. We verify the equivalence of these graphs by printing the list of edges for the graph merged from two colored de Bruijn graphs (where the number of colors is equal to 3) as well as the colored de Bruijn graph (where the number of colors is equal to 6) constructed using Vari. We compared the full edge labels and found that the only difference was 126 additional extra edges in the merged graph. Furthermore, applying VARI's bubble calling method to both graphs returned the same set of bubbles.

6 Conclusion

We find VariMerge provides a more efficient means to update a succinct colored de Bruijn graph as more data becomes available. We also find it enables building larger graphs in a given total working space (RAM and external).

This radix based merge method may be of benefit to other prefix-only compressed suffix arrays such as GCSA by Sirén*et al.* [27] and XBW by Ferragina *et al.*[11], which may be an avenue for future work.

References

1. F. Almodaresi, P. Pandey, and R. Patro. Rainbowfish: A succinct colored de bruijn graph representation. In *Proc. of WABI*, pages 251–265, 2017.
2. U. Baier, T. Beller, and E. Ohlebusch. Graphical pan-genome analysis with compressed suffix trees and the Burrows–Wheeler transform. *Bioinformatics*, 32(4):497–504, 2015.
3. A. Bowe, T. Onodera, K. Sadakane, and T. Shibuya. Succinct de Bruijn graphs. In *In Proc. of WABI*, pages 225–235, 2012.
4. D. Boxrud, T. Monson, T. Stiles, and J. Besser. The role, challenges, and support of pulsenet laboratories in detecting foodborne disease outbreaks. *Public Health Reports*, 125(2_suppl):57–62, 2010.
5. M. Burrows and D.J. Wheeler. A block sorting lossless data compression algorithm. Technical Report 124, Digital Equipment Corporation, Palo Alto, California.
6. H.A. Carleton and P. Gerner-Smidt. Whole-genome sequencing is taking over foodborne disease surveillance. *Microbe*, 11:311–317, 2016.
7. R. Chikhi, A. Limasset, S. Jackman, J. T. Simpson, and P. Medvedev. On the representation of de Bruijn graphs. In *Proc. of RECOMB*, pages 35–55, 2014.
8. R. Chikhi and G. Rizk. Space-efficient and exact de Bruijn graph representation based on a Bloom filter. *Algorithms for Molecular Biology*, 8(1), 2013.
9. T.C. Conway and A. Bromage. Succinct data structures for assembling large genomes. *Bioinformatics*, 27(4):479–486.
10. P. Ferragina and G. Manzini. Indexing compressed text. *Journal of the ACM*, 52(4):552–581.
11. Paolo Ferragina, Fabrizio Luccio, Giovanni Manzini, and S Muthukrishnan. Compressing and indexing labeled trees, with applications. *Journal of the ACM (JACM)*, 57(1):4, 2009.
12. Simon Gog, Timo Beller, Alistair Moffat, and Matthias Petri. From theory to practice: Plug and play with succinct data structures. In *13th International Symposium on Experimental Algorithms, (SEA 2014)*, pages 326–337, 2014.
13. G. Holley, R. Wittler, and J. Stoye. Bloom filter trie–a data structure for pan-genome storage. In *Proc. of WABI*, pages 217–230, 2015.
14. Z. Iqbal, M. Caccamo, I. Turner, P. Flicek, and G. McVean. De novo assembly and genotyping of variants using colored de Bruijn graphs. *Nature Genetics*, 44:226–232, 2012.
15. D. Li, C.-M. Liu, R. Luo, K. Sadakane, and T.-W. Lam. MEGAHIT: An ultra-fast single-node solution for large and complex metagenomics assembly via succinct de Bruijn graph. *Bioinformatics*, 31(10):1674–1676, 2015.
16. M.C.J. Maiden, M.J. Van Rensburg, J.E. Bray, S.G. Earle, S.A. Ford, K.A. Jolley, and N.D. McCarthy. MLST revisited: the gene-by-gene approach to bacterial genomics. *Nature Reviews*, 11(10):728, 2013.
17. U. Manber and G. W. Myers. Suffix arrays: A new method for on-line string searches. *SIAM Journal of Scientific Computing*, 22(5), 1993.
18. S. Marcus, H. Lee, and M.C. Schatz. SplitMEM: A graphical algorithm for pan-genome analysis with suffix skips. *Bioinformatics*, 30(24):3476–3483, 2014.
19. I. Minkin, S. Pham, and P. Medvedev. TwoPaCo: An efficient algorithm to build the compacted de Bruijn graph from many complete genomes. *Bioinformatics*, page btw609, 2016.
20. M.D. Muggli, A. Bowe, N.R. Noyes, P.S. Morley, K.E. Belk, R. Raymond, T. Gagie, S.J. Puglisi, and C. Boucher. Succinct colored de bruijn graphs. *Bioinformatics*, page btx067, 2017.
21. D. Okanohara and K. Sadakane. Practical entropy-compressed rank/select dictionary. In *Proc. of ALENEX*, pages 60–70. SIAM, 2007.
22. Y. Peng et al. IDBA-UD: A *de novo* assembler for single-cell and metagenomic sequencing data with highly uneven depth. *Bioinformatics*, 28(11), 2012.
23. J.B. Pettengill, A.W. Pightling, J.D. Baugher, H. Rand, and E. Strain. Real-time pathogen detection in the era of whole-genome sequencing and big data: Comparison of k-mer and site-based methods for inferring the genetic distances among tens of thousands of salmonella samples. *PloS ONE*, 11(11):e0166162, 2016.
24. C.H. Sandt, D.A. Krouse, C.R. Cook, A.L. Hackman, W.A. Chmielecki, and N.G. Warren. The key role of pulsed-field gel electrophoresis in investigation of a large multiserotype and multistate food-borne outbreak of salmonella infections centered in pennsylvania. *Journal of Clinical Microbiology*, 44(9):3208–3212, 2006.
25. J.T. Simpson et al. ABySS: A parallel assembler for short read sequence data. *Genome Research*, 19(6):1117–1123, 2009.
26. J. Sirén. Burrows-Wheeler transform for terabases. In *Proc. of DCC*, pages 211–220, 2016.
27. Jouni Sirén, Niko Valimaki, and Veli Makinen. Indexing graphs for path queries with applications in genome research. *Computational Biology and Bioinformatics, IEEE/ACM Transactions on*, 11(2):375–388, 2014.
28. E.L. Stevens, R. Timme, E.W. Brown, M.W. Allard, E. Strain, K. Bunning, and S. Musser. The public health impact of a publically available, environmental database of microbial genomes. *Frontiers in Microbiology*, 8:808, 2017.

7 Supplementary Material

7.1 Basic Definitions and Terminology

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

Suffix arrays. The suffix array [17] SA_X (we drop subscripts when they are clear from the context) of a string X is an array $\mathsf{SA}[1..n]$ which contains a permutation of the integers $[1..n]$ such that $\mathsf{X}[\mathsf{SA}[1]..n] < \mathsf{X}[\mathsf{SA}[2]..n] < \cdots < \mathsf{X}[\mathsf{SA}[n]..n]$. In other words, $\mathsf{SA}[j] = i$ iff $\mathsf{X}[i..n]$ is the jth suffix of X in lexicographical order.

Suffix array intervals. For a string Y, the Y-interval in the suffix array SA_X is the interval $\mathsf{SA}[s..e]$ that contains all suffixes having Y as a prefix. The Y-interval is a representation of the occurrences of Y in X. For a character c and a string Y, the computation of $c\mathsf{Y}$-interval from Y-interval is called a *left extension*.

BWT and backward search. The Burrows-Wheeler Transform [5] $\mathsf{BWT}[1..n]$ is a permutation of X such that $\mathsf{BWT}[i] = \mathsf{X}[\mathsf{SA}[i] - 1]$ if $\mathsf{SA}[i] > 1$ and \$ otherwise. We also define $\mathsf{LF}[i] = j$ iff $\mathsf{SA}[j] = \mathsf{SA}[i] - 1$, except when $\mathsf{SA}[i] = 1$, in which case $\mathsf{LF}[i] = I$, where $\mathsf{SA}[I] = n$.

Ferragina and Manzini [10] linked BWT and SA in the following way. Let $\mathsf{C}[c]$, for symbol c, be the number of symbols in X lexicographically smaller than c. The function $\mathsf{rank}(\mathsf{X}, c, i)$, for string X, symbol c, and integer i, returns the number of occurrences of c in $\mathsf{X}[1..i]$. It is well known that $\mathsf{LF}[i] = \mathsf{C}[\mathsf{BWT}[i]] + \mathsf{rank}(\mathsf{BWT}, \mathsf{BWT}[i], i)$. Furthermore, we can compute the left extension using C and rank. If $\mathsf{SA}[s..e]$ is the Y-interval, then $\mathsf{SA}[\mathsf{C}[c] + \mathsf{rank}(\mathsf{BWT}, c, s), \mathsf{C}[c] + \mathsf{rank}(\mathsf{BWT}, c, e)]$ is the $c\mathsf{Y}$-interval. This is called *backward search* [10], and a data structure supporting it is called an *FM-index*.

7.2 Figures Illustrating VariMerge

Here, we give two figures to illustrate the merge procedure of the primary components. We show the planning stage in Figure 1, and how to produce the necessary components in memory in Figure 2.

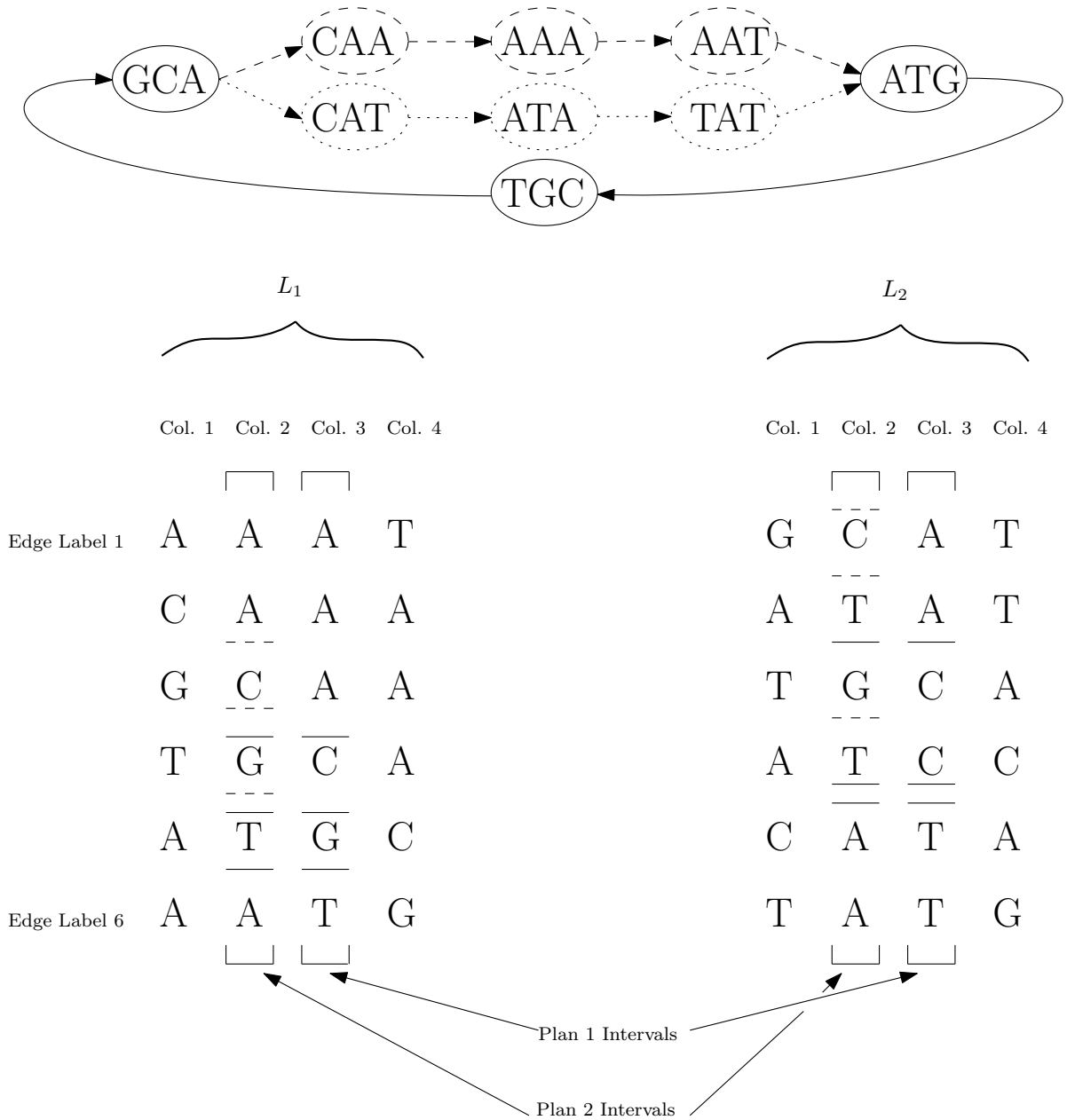

Fig. 1: **Two lists of (conceptual) edge labels and the corresponding de Bruijn graph.** Dotted and dashed lines denote edges exclusive to one graph. Solid lines are common to both. The merged graph will contain all components. Plans are refined in decreasing sort precedence order: (Col. 3, Col. 2,. Col. 1, Col. 4). Plan 1 partitions the full range of edges into four intervals. These intervals are further partition in Plan 2. The number of subintervals an interval is partitioned into depends on the size of the alphabet in use in both sub columns from L_1 and L_2 (e.g. The L_2 half of Plan 1 has four intervals because the entire DNA alphabet is used in L_1, Col. 3). This may introduce empty intervals, denoting that the corresponding edge labels for one graph are absent in the other. Identical edge labels between graphs will always be in equal ranked intervals (e.g. TCGA is in the second interval for both graphs in Plan 1, while it is in the fourth interval in Plan 2.)

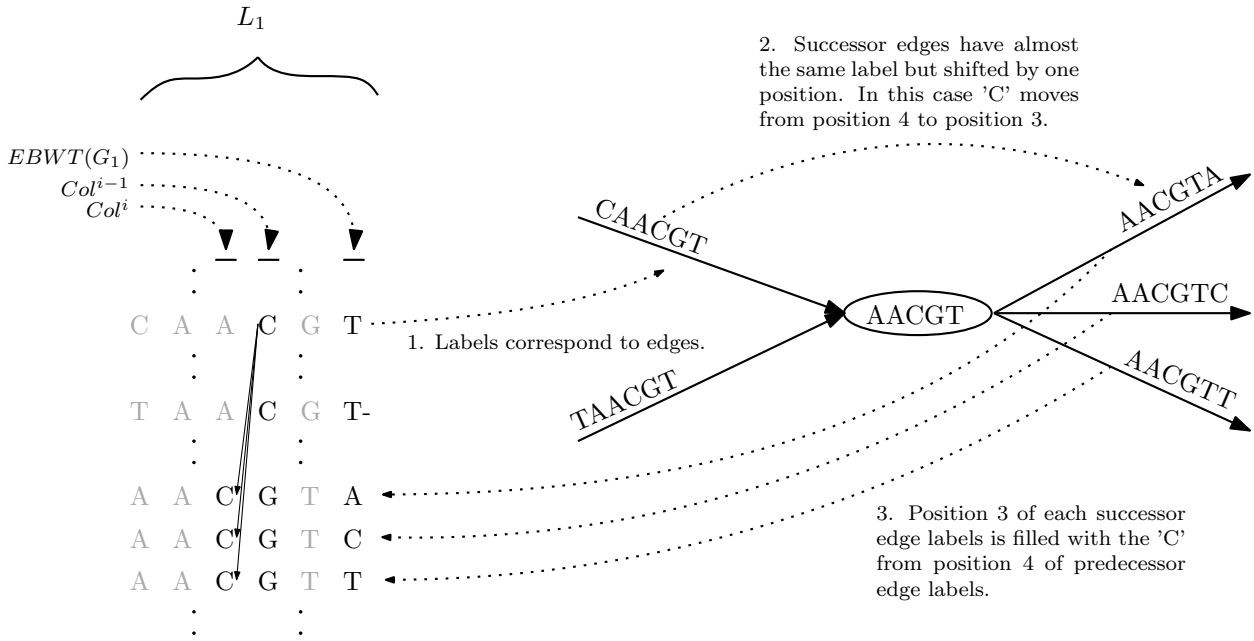

Fig. 2: **Method for populating Col^i based on Col^{i-1} and graph navigation.** Black nucleotides represent data that is in memory and valid. Grey represents data that is stored in external memory in VARI but only exists ephemerally in VariMerge. Thus, only three columns are ever present in memory, which is a significant memory savings relative to the full set of edge labels. The three resident vectors are $EBWT(G_1)$ (which is always present and used for navigation, Col^{i-1} which is already completely populated when a new column to the left (Col^i) is being generated.

7.3 Comparison Between VariMerge and Bloom Filter Trie

We give Figure 3 to illustrate the scalability of the VARI data structure and Bloom Filter Trie on the GenomeTrakr dataset.

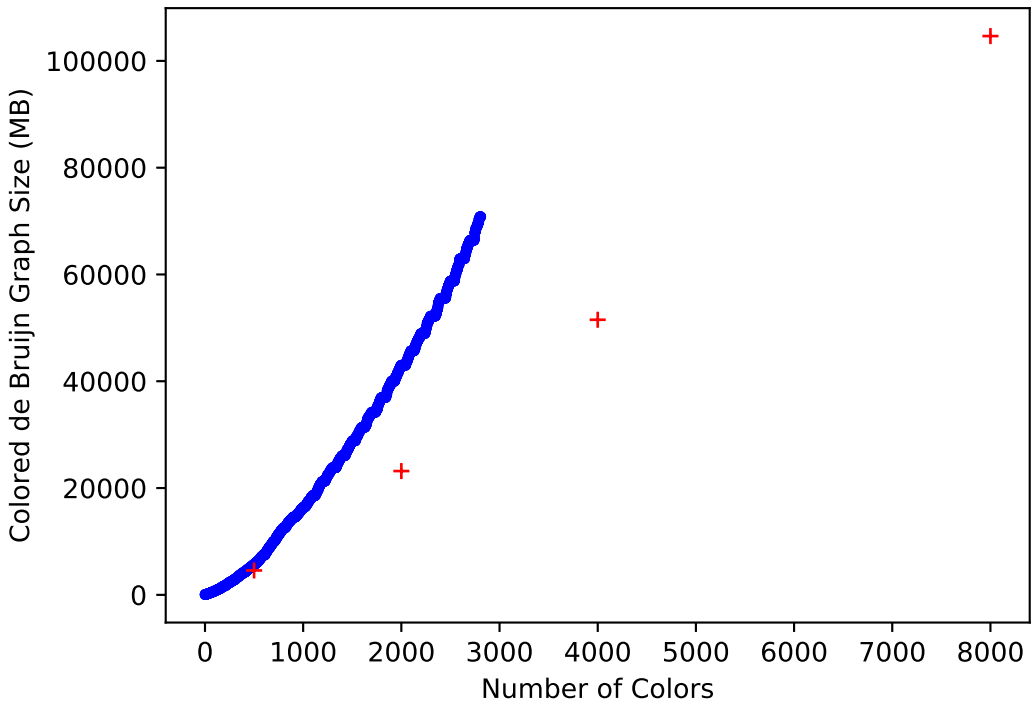

Fig. 3: Bloom Filter Tries (blue dots) grow more quickly than VARI (red pluses) on isolates from GenomeTrakr. Bloom Filter Trie was run on 2,000 isolates and VariMerge was ran on 4,000.

7.4 Pseudocode for VariMerge

Algorithm 1 Algorithm to generate a plan. *AlphabetUsed*() returns the set of symbols used in its arguments. *IntervalOccupied*() returns the c run interval found in its second window argument. Assume window objects (W_1 and W_2) retain their origin such that *IntervalOccupied*() returns intervals with respect to the source positions in Col_1 and Col_2. *CoveredSymbols*() returns the substring (with the aforementioned source interval) which is covered by an argument interval. *IntervalLast*() returns true if the given position is the last in the given interval.

```
procedure PARTITION(W1,W2)
    Σ' ← AlphabetUsed(W1, W2)
    SubP1 ← ()
    SubP2 ← ()
    for all c ∈ Σ' do
        SubP1.Append(IntervalOccupied(c, W1))
        SubP2.Append(IntervalOccupied(c, W2))
    end for
    return (SubP1, SubP2)
end procedure

procedure REFINEPLAN(P1, P2, Col1, Col2, i)
    P'1 ← ()
    P'2 ← ()
    ▷ For each interval in the (equal length) plans...
    for all j ∈ {1..|P1|} do
        ▷ ...extract a window from each column covered by the interval...
        W1 ← CoveredSymbols(Col1, P1[j])
        W2 ← CoveredSymbols(Col2, P2[j])
        ▷ ...and partitioning that window on its character runs, forming sub-intervals.
        (SubP1, SubP2) ← Partition(W1, W2)
        P'1.Concatenate(SubP1)
        P'2.Concatenate(SubP2)
    end for
    ▷ Capture snapshots of important intermediate plan states.
    if i ∈ {1, k − 1, k − 2}  then
        Si ← (P'1, P'2)
    end if
    return (P'1, P'2)
end procedure

procedure VARIMERGEPLAN(G1, G2)
    ▷ Initialize plan to single intervals covering entire EBWT(G)s.
    P1 ← ([1, |EBWT(G)1|])
    P2 ← ([1, |EBWT(G)2|])
    ▷ Iterate through "edge label matrix" columns in sort precedence order
    for all i ∈ {1..k} do
        Col1 ← GetCol(i, G1)
        Col2 ← GetCol(i, G2)
        (P'1, P'2) ← RefinePlan(P1, P2, Col1, Col2, i)
        (P1, P2) ← (P'1, P'2)
    end for
end procedure
```

Algorithm 2 Algorithm to execute the merge plan.

```
procedure VariMergeExecute(G_1, G_2)
    ▷ Phase 2: Execute plan
    ▷ For each interval in the (equal length) plans...
    for all j ∈ {1..|P_1|} do
        NTcounts ← [0, 0, 0, 0, 0]
        flagcounts ← [0, 0, 0, 0, 0]
        EBWT(G)_M ← (), B_LM ← (), flags_M ← ()
        G1ptr ← 1, G2ptr ← 1
        if |P_1[j]| = 1 then
            EBWT(G)_M.Append(EBWT(G)_1[G1ptr])
            B_LM.Append(IntervalLast(G1ptr, S_{k−1}[1]))
            flags_M.Append(flagcounts[EBWT(G)_1[G1ptr]] ≠ 0)
            flagcounts[EBWT(G)_1[G1ptr]] ← +1
            G1ptr ← +1
            if |P_2[j]| = 1 then
                G2ptr ← +1
            end if
        else
            EBWT(G)_M.Append(EBWT(G)_2[G2ptr])
            B_LM.Append(IntervalLast(G2ptr, S_{k−1}[2]))
            flags_M.Append(flagcounts[EBWT(G)_2[G2ptr]] ≠ 0)
            flagcounts[EBWT(G)_2[G1ptr]] ← +1
            G2ptr ← +1
        end if
        ▷ When the last symbol(s) are consumed from equal an rank interval pair in S_{k−2}, reset the flag counter.
        if IntervalLast(G1ptr, S_{k−2}[1]) and IntervalLast(G2ptr, S_{k−2}[2]) then
            flagcounts ← [0, 0, 0, 0, 0]
        end if
    end for
    D_M ← PrefixSum(NTcounts)
end procedure
```