DISSERTATION

ENHANCING SPACE AND TIME EFFICIENCY OF GENOMICS IN PRACTICE THROUGH SOPHISTICATED APPLICATIONS OF THE FM-INDEX

Submitted by

Martin D. Muggli

Department of Computer Science

In partial fulfillment of the requirements

For the Degree of Doctor of Philosophy

Colorado State University

Fort Collins, Colorado

Fall 2018

Doctoral Committee:

Advisor: Ross McConnell

Charles Anderson
Hamid Chitsaz
Paul S Morley

ABSTRACT

ENHANCING SPACE AND TIME EFFICIENCY OF GENOMICS IN PRACTICE THROUGH SOPHISTICATED APPLICATIONS OF THE FM-INDEX

Genomic sequence data has become so easy to get that the computation to process it has become a bottleneck in the advancement of biological science. A data structure known as the FM-Index both compresses data and allows efficient querying, thus can be used to implement more efficient processing methods. In this work we apply advanced formulations of the FM-Index to existing problems and show our methods exceed the performance of competing tools.

ACKNOWLEDGEMENTS

I would like to thank Christina Boucher who filled the roll of advisor for all of this work for her patience, guidance, and insight in my training. I would also like to thank my committee, Ross McConnell, Charles Anderson, Hamid Chitsaz, and Paul S Morley, for their continued help and support. I would like to thank the many people who have supported me emotionally throughout this work, specifically: Alex Bowe, Lindsay Burchfield, Aaron Ciuffo, Katie Davidson, Teri Dillion, Thomas Harrison, Amber Johnson, Anne Mangiardi, Piper Murray, Simon Puglisi, Susan Rainey, Sanjay Rajopadhye, Sara Rector, Stacey Reynolds, Basir Shariat, Mike Storlie, Michelle Strout, John Wagner, Jason Walp, Adrienne Watral and others who I may have forgotten. I would also like to thank the various people and organizations whose outstanding science work provided inspiration: Richard P Feynman, ESA, CERN, LIGO, GHC. In addition, I would like to thank my co-authors for their contribution to our work. And last but not least I would like to thank my family, Jaynie Muggli, Jim Muggli, and Karen Muggli for their support as well.

TABLE OF CONTENTS

Chapter 1

Introduction

With the advent of high throughput sequencing, biologists have gained access to massive amounts of data sampled from their specimens. While this is a boon for biologists, all this data must be processed to yield useful information and many times, state of the art computing hardware and software are a limiting factors for biological inquiry. For example, a set of samples from 3,765 *E. coli* would require over 3 TB of memory to represent in a state of the art tool, far exceeding even the 1 TB of RAM in machines some labs are fortunate to have access to.

There are however opportunities for improvement. Existing methods store their data, which usually has some inherent redundancy, in a direct, one-to-one representation. Thus enhancing these methods with compression techniques can reduce their memory footprint. Additionally, data often contain erroneous values which existing methods accomodate with some form of exhaustive search techniques. Instead, sufficiently powerful indexing techniques can provide error tolerant lookup mechanisms instead and reduce the runtime.

A data structure known as the FM-Index can provide both indexing and compression simultaneously. In fact, the basic string search and compression capabilities of the FM-Index are fairly well known and applied. The search capabilities are used within the nearly ubiquitous BWA [1] software in bioinformatics. And the compression capabilities underlying the FM-Index are even more widespread in tools such as the BZIP2.

The FM-Index, however, is not a drop in replacement for existing techniques. Various problems need more than simple string indexing over small alphabets, but require indexing graphs and large alphabets. Fortunately, there are more advanced developments that make the FM-Index applicable to graphs and queries based on ordinal elements.

We explore extending the reach of the available tool set with new tools which apply sophisticated forms of the FM-Index. This enhances the efficiency on each of the runtime and space

requirements over state of the art tools on existing genomics problems. Specifically to illustrate this, we enable detection of variants among a dataset of approximately 16,000 *Salmonella* samples taken from food production facilities. Additionally, we demonstrate the usefulness of tools which reduce runtime by means of sophisticated indexing on a more recent form of genomic data in the form of optically derived restriction maps.

1.1 Background

Within the scope of genomics, there are two overarching tasks we'll focus on improving: The acquisition of complete genome sequences for organisms and detection of variations of genomes among a population.

Though scientists have been advancing the lab methods for sequencing genomes for almost four decades, no method today can sequence entire chromosomes. Genomic DNA still must be sheared into fragments small enough for available technology to sequence (currently approx 100-10,000 base pairs), and then an assembly process used to reconstruct the original genome sequence by joining related fragments' strings (called *reads*) based on their similar substrings [2, 3].

The question naturally arises for how we can find these similar regions between reads, especially in the presence of noise such as read errors. An alignment is a relationship between two strings which may not exactly match each other. As such, alignments are an effective measure of similarity [4]. While alignments have many applications, we will first focus our discussion on alignments as relationships between strings (e.g., genomic data such as reads, the genome itself, or other sequence data derived from the genome). An alignment can be expressed as a sequence of edits (insertions, deletions, and substitutions) to convert one string into another. When we align genomic data, these edits appear either in the presence of read errors when the two strings originated as reads of a single genome, or in the presence of variation between two genomes otherwise. Scores are often associated with alignments based on the sum of scores of the edit operations which compose alignments. A good scoring alignment between two strings can serve as an indicator that they may share a common genomic origin.

Often the scoring scheme allows penalty free runs of insertions or deletions at one or more ends of a sequence, allowing for the fact that two strings may both cover a shared region of a genome but each string also covering a portion the other did not. If opposing ends of the two sequences are allowed not to match in a penalty free manner, it is called semi-global alignment and is the expected case for two reads that overlap the same locus but have different start and end positions.

Dynamic programming based algorithms can be used to find the optimal alignment between two strings under some scoring scheme; however, this is typically formulated as an $\mathcal{O}(mn)$ problem where m and n are the lengths of the strings. While this running time is not necessarily a problem between a single pair of strings, we're often interested in all high scoring alignments between all pairs of strings. Computing this would entail running dynamic programming $O(|R|^2)$ times where R is the set of all reads, making the composition of these running times far too time consuming for all but the smallest genomes. Thus other methods we'll explore later often replace dynamic programming based alignment techniques.

When the objective is to recover a complete genome sequence given a set of reads, it is necessary to find similar regions between reads such that we can reconstruct the genomic regions where the reads originate. There are two paradigms in active use for this: overlap-layout-consensus based and de Bruijn graph based.

Overlap layout consensus assembly works by finding alignments between pairs of reads, representing those alignments in a graph where reads form nodes and alignments form edges, and then finding Hamiltonian tours [5].

De Bruijn graph based assemblers chop reads up into a series of overlapping substrings of length k called k-mers [6]. k-mers are subdivided into a $(k-1)$-mer prefix and $(k-1)$-mer suffix. Each $(k-1)$-mer becomes a node in a graph, with the original k-mer becoming a directed edge connecting them. All vertices with the same $(k-1)$-mer label are then glued together. When vertices with the same $(k-1)$-mer label originate from different reads, the glued node is how a similar region between those reads is represented. Thus gluing $(k-1)$-mers takes the place of finding alignments between reads. This graph is then traversed, finding Eularian tours.

In the ideal case, either method could reconstruct a genome from reads given enough resources. However, this task is complicated in practice by the fact that genomes contain repeated regions. This means that two reads that appear to share a similar substring may actually have been read from different loci (or locations) in the genome. Thus for either assembly approach, repeats in the genome cause read data originating from disparate loci to have a spurious relation in an assembly graph (either glued together, becoming one node in the de Bruijn graph, or having an alignment edge in the overlap-layout-consensus graph). These coincidental alignment relations introduce cycles in the graph. Such cycles can make it impossible to unambiguously determine how to reconstruct the original genome - while the original genome sequence can be found as one specific walk through either graph, it's typically not possible to determine which of many possible walks in an assembly graph represents the true genome path, so assembly tools emit those non-branching paths which can be inferred with high confidence to be contiguous regions of the genome. These paths spell strings known as *contigs*.

1.1.1 FM-Index

As mentioned previously, finding alignments by means of pairwise dynamic programming can be too computationally expensive for all but the smallest genomes. However, with relatively error-free strings (either because an error correction procedure has been run, or the strings are small enough to often avoid errors, or the sequencing technology is highly accurate) another alternative to dynamic programming based alignment is to use a data structure called a *suffix array* for finding predominantly identical common substrings. Conceptually, this is an array consisting of all the suffixes of a string in sorted order [7]. Associated with each element in the array is the index in the original string where that element's suffix begins. This can be efficiently implemented in practice. For example, in the C programming language, the suffix array can be represented as an array of either pointers or offsets into the original string, avoiding the redundancy of storing each suffix separately. Any string that matches the prefix of some suffix of the original string of length n can then be found by binary search in time $\mathcal{O}(\log n)$.

Formally, 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 $\Sigma = [0..\sigma - 1]$. 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 and suffix array intervals.

The suffix array [8] 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 $\mathsf{X}[\mathsf{SA}[1]..n] \prec \mathsf{X}[\mathsf{SA}[2]..n] \prec \cdots \prec \mathsf{X}[\mathsf{SA}[n]..n]$. In other words, $\mathsf{SA}[j] = i$ if and only if $\mathsf{X}[i..n]$ is the j^{th} suffix of X in lexicographical order. Here, $\prec$ denotes lexicographic precedence.

A clever data structure known as an *FM-index* is often used as a memory efficient alternative to a suffix array. To explain this structure, we will start with a conceptual model. The source string has a special out-of-alphabet symbol (e.g., '$') appended to it. Then all possible rotations of this string are created and stacked vertically as rows in a matrix. The rows of this matrix are then sorted, yielding a matrix similar to the suffix array. The string comprising the last column of this matrix is known as the Burrows-Wheeler transform (BWT) [9] of the source string.

Formally, for a string X, let F be the list of X's characters sorted lexicographically by the suffixes starting at those characters, and L be the list of X's characters sorted lexicographically by the suffixes starting immediately after those characters. (The names F and L are standard for these lists.) If $\mathsf{Y}[i]$ is in position p in F then $\mathsf{Y}[i-1]$ is in position p in L. Moreover, if $\mathsf{Y}[i] = \mathsf{Y}[j]$ then $\mathsf{Y}[i]$ and $\mathsf{Y}[j]$ have the same relative order in both lists; otherwise, their relative order in F is the same as their lexicographic order. This means that if $\mathsf{Y}[i]$ is in position p in L then (assuming arrays are indexed from 0) in F it is in position

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

Finally, notice that the last character in X always appears first in L. It follows that we can recover X from L, and thus L is the famous *Burrows-Wheeler Transform (BWT)* [10] of X. For an example string "catgcat$", the Burrows-Wheeler transform is "tccg$taa".

More succinctly, the Burrows-Wheeler Transform [10] $\mathsf{BWT}[1..n]$ is a permutation of X such that $\mathsf{BWT}[i] = \mathsf{X}[\mathsf{SA}[i] - 1]$ if $\mathsf{SA}[i] > 1$ and \$ otherwise.

The BWT has a number of useful properties. If the source string has repeats, then the sorted rotations will naturally position all the repeated suffixes sharing the same prefix in a contiguous run of rows. All of those same suffixes without their first character will also be in a contiguous run of rows, and since each row is a rotation, all the first characters we considered initially will be found in the last column as a run of the repeated character. Runs of repeated characters can be compressed by various means, such as run length encoding where the repeated character and the length of the run are stored instead of the repeated instances of that character. Thus, the BWT of a string containing repeats can be represented in less memory than the original string.

Note that in practice, the conceptual BWT matrix outlined above does not need to be constructed to get the BWT of the text; one can simply sort all the suffixes and take the character that precedes each suffix. This sequence of characters is then equivalent to the last column of our conceptual model since they are rotations in the matrix.

Additionally, by adding two auxiliary data structures (*Occ*: a rank() capable dictionary for the last matrix column L and S[1]: a trivial select() capable data structure for the equivalent of the first matrix column F) to the BWT, an extended data structure known as the aformentioned FM-index can be constructed. It can allow the BWT to act as a self index into the original string which allows exact matches to a query string to be found in time linear in the length of the query [11]. This works by finding a succession of intervals in the suffix array (whose elements correspond to those of the BWT as seen from the Burrows Wheeler matrix) which match progressively longer suffixes of a query string.

[1]This is traditionally represented as C

Formally, for a string Y, the Y-interval in the suffix array $\mathsf{SA_X}$ 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*.

Ferragina and Manzini [12] first realized BWT can be used for indexing in addition to compression. Hence, if we know the range $\mathsf{BWT}(\mathsf{X})[i..j]$ is occupied by characters immediately preceding occurrences of a pattern Y in X, then we can compute the range $\mathsf{BWT}(\mathsf{X})[i'..j']$ occupied by characters immediately preceding occurrences of cY in X, for any character c, since

$$\begin{aligned} s' &= |\{h : \mathsf{X}[h] \prec c\}| + |\{h : \mathsf{X}[h] = c, h < s\}| \\ e' &= |\{h : \mathsf{X}[h] \prec c\}| + |\{h : \mathsf{X}[h] = c, h \leq e\}| - 1\,. \end{aligned}$$

Notice $e' - s' + 1$ is the number of occurrences of cY in S. The essential components of an FM-index for X are: (1) an array S storing $|\{h : \mathsf{X}[h] \prec c\}|$ for each character c and, (2) a *rank* data structure Occ for $\mathsf{BWT}(\mathsf{X})$ that quickly tells us how often any given character occurs up to any given position. To be able to locate the occurrences of a pattern Y in X (in addition to just counting them), we can use a sampled suffix array of X and a bitvector indicating the positions in $\mathsf{BWT}(\mathsf{X})$ of the characters preceding the sampled suffixes.

Hence, we define the function $\mathsf{rank}_c(\mathsf{X}, i)$, for string X, symbol c, and integer i, as the number of occurrences of c in $\mathsf{X}[1..i]$. Rank is used in *backward search* [12] in order to compute left extension of a given string, i.e., the previous character.

To support rank queries in backward search, a data structure called a *wavelet tree* [13] can be used. It occupies $n \log \sigma + o(n \log \sigma)$ bits of space and supports rank queries in $\mathcal{O}(\log \sigma)$ time. Wavelet trees also support a variety of more complex queries on the underlying string efficiently [13]. One such query we will use in this paper is to return the set Z of distinct symbols occurring in $X[i, j]$, which takes $\mathcal{O}(|Z| \log \sigma)$ time.

In the remainder of this text we will look at two broad problems in genomics. In Chapter 2, we will examine approaches that are useful for validating draft genome assemblies. In Chapter 3, we will examine methods for comparing genomes in large populations. Finally, we'll conclude by considering the relationship between the graph based representation of genomic data that were presented.

Chapter 2

Reducing Runtime by Indexing

In this section, we look at applications of the FM-Index which reduce alignment runtime over competing tools by using the indexing capabilities of sophisticated variations of the FM-Index. In particular, we consider how an emerging form of genomic data, optically derived restriction maps, present special challenges but can still be aligned with the FM-Index.

2.1 TWIN: Efficient Indexed Alignment of Contigs to Optical Maps[2]

In this section, we look at applying the FM-Index to aligning a consensus form of data, where the principle challenge to application is that we are dealing with sequences over a large alphabet and the symbols rarely match exactly. Later, we will build upon the solution explored here to solve alignment on raw, non-consensus data, which has additional complexities.

2.1.1 Introduction

In this section, we begin our more in depth investigation of succinct data structures. Specifically, we examine how the FM-Index and wavelet tree can be used to store a compressed index of a string of integral valued symbols which can be efficiently queried.

Our motivation for this application is as follows. Despite considerable research, *de novo* genome assembly, the process of reconstructing long contiguous sequences (*contigs*) from short sequence reads, still produces a substantial number of errors [14, 15] and is easily misled by repetitive regions [16].

[2]M. Muggli *et al.* Efficient indexed alignment of contigs to optical maps. In *Proceedings WABI*, pages 68-81, 2014.

One way to improve the quality of assembly is to use secondary information (independent of the short sequence reads themselves) about the order and orientation of contigs. Optical mapping, which constructs ordered genome-wide high-resolution restriction maps, can provide such information. Optical mapping is a system that works as follows [17, 18]. An ensemble of DNA molecules adhered to a charged glass plate are elongated by fluid flow. An enzyme is then used to cleave them into fragments at loci where the enzyme's recognition sequence occurs. Next, the remaining fragments are highlighted with fluorescent dye and digitally photographed under a microscope. Finally, these images are analyzed to estimate the fragment sizes, producing a molecular map. Since the fragments stay relatively stationary during the aforementioned process, the images capture their relative order and size [19]. Multiple copies of the genome undergo this process, and a consensus map is formed that consists of an ordered sequence of fragment sizes, each indicating the approximate number of bases between occurrences of the recognition sequence in the genome [20].

The raw optical mapping data identified by the image processing is an ordered sequence of fragment lengths. Hence, an optical map with m fragments can be denoted as $\ell = \{\ell_1, \ell_2, \ldots, \ell_m\}$, where ℓ_i is the length of the ith fragment in base pairs. This data can then be converted into a sequence of locations, each of which determines where a restriction site occurs. We denote the converted data as follows: $L(x) = \{L_0 < L_1 < \cdots < L_n\}$, where $\ell_i = L_i - L_{i-1}$ for $i = 1, \ldots, n$, and L_0 and L_n are defined by the original molecule as a segment of the whole genome by shearing. This latter representation is convenient for algorithmic descriptions. The approximate mean and standard deviation of the fragment size error rate for current consensus data [21] are zero and 150 bp, respectively. See Figure 2.1 for an illustration of the data produced by this technique. Each restriction enzyme recognizes a specific nucleotide sequence so a unique optical map results from each enzyme.Optical maps have recently become commercially available for mammalian-sized genomes[3], allowing them to be used in a variety of applications.

[3]OpGen (http://www.opgen.com) and BioNano (http://www.bionanogenomics.com) are commercial producers of optical mapping data.

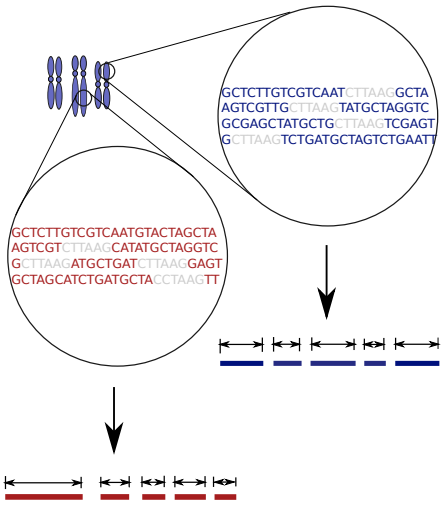


Figure 2.1: An illustration of the data produced by optical mapping. Optical mapping locates and measures the distance between restriction sites. Analogous to sequence data, optical mapping data is produced for multiple copies of the same genome, and overlapping single molecular maps are analyzed to produce a map for each chromosome.

Although optical mapping data has been used for structural variation detection [22], scaffolding and validating contigs for several large sequencing projects — including those for various prokaryote species [23–25], *Oryza sativa* (rice) [26], maize [27], mouse [28], goat [29], *Melopsittacus Undulatus* (budgerigar) [30], and *Amborella trichopoda* [31] — there exist few non-proprietary tools for analyzing this data. Furthermore, the currently available tools are extremely slow because most of them were specifically designed for smaller, prokaryote genomes.

Our Contribution. We present the first index-based method for aligning contigs to an optical map. We call our tool TWIN to illustrate the association between the assembly and optical map as two representations of the genome sequence. The first step of our procedure is to *in silico* digest the contigs with the set of restriction enzymes, computationally mimicking how each restriction enzyme would cleave the short segment of DNA defined by the contig. Thus, *in silico digested contigs* are miniature optical maps that can be aligned to the much longer (sometimes genome-wide) optical maps. The objective is to search and align the *in silico* digested contigs to the correct location in the optical map. By using a suitably-constructed FM-Index data structure [12] built on

the optical map, we show that alignments between contigs and optical maps can be computed in time that is faster than competing methods by more than two orders of magnitude.

TWIN takes as input a set of contigs and an optical map, and produces a set of alignments. The alignments are output in Pattern Space Layout (PSL) format, allowing them to be visualized using any PSL visualization software, such as IGV [32]. TWIN is specifically designed to work on a wide range of genomes, anything from relatively small genomes, to large eukaryote genomes. Thus, we demonstrate the effectiveness of TWIN on *Yersinia kristensenii*, rice, and budgerigar genomes. Rice and budgerigar have genomes of total sizes 430 Mb and 1.2 Gb, respectively. *Yersinia kristensenii*, a bacteria with genome size of 4.6 Mb, is the smallest genome we considered. Short read sequence data was assembled for these genomes, and the resulting contigs were aligned to the respective optical map. We compared the performance of our tool with available competing methods; specifically, the method of Valouev et al. [33] and SOMA [34]. TWIN has superior performance on all datasets, and is demonstrated to be the only current method that is capable of completing the alignment for the budgerigar genome in a reasonable amount of CPU time; SOMA [34] required over 77 days of machine time to solve this problem, whereas, TWIN required just 35 minutes. Lastly, we verify our approach on simulated *E. coli* data by showing our alignment method found correct placements for the *in silico* digested contigs on a simulated optical map. TWIN is available for download at http://www.cs.colostate.edu/twin.

Roadmap. We review related tools for the problem in the remainder of this section. Section 2.1.2 then sets notation and formally lays the data structural tools we make use of. Section 2.1.3 gives details of our approach. We report our experimental results in Section 2.1.4. Finally, Section 2.1.5 offers reflections and some potentially fruitful avenues future work may take.

Related Work. The most recent tools to make use of optical mapping data in the context of assembly are AGORA [35] and SOMA [34]. AGORA [35] uses the optical map information to constrain de Bruijn graph construction with the aim of improving the resulting assembly. SOMA [34] is a scaffolding method that uses an optical map and is specifically designed for

short-read assemblies. SOMA requires an alignment method for scaffolding and implements an $\mathcal{O}(n^2m^2)$-time dynamic programming algorithm. Gentig [20], and software developed by Valouev et al. [33] also use dynamic programming to address the closely related task of finding alignments between optical maps. Gentig is not available for download. BACop [27] also uses a dynamic programming algorithm and corresponding scoring scheme that gives more weight to contigs with higher fragment density. Antoniotti et al. [36] consider the unique problem of validating an optical map by using assembled contigs. This method assumes the contigs are error-free. Optical mapping data was produced for Assemblathon 2 [37].

2.1.2 Background

Optical Mapping. From a computational point of view, restriction mapping (by optical or other means) is a process that takes two strings: a genome $\mathsf{A}[1, n]$ and a restriction sequence $\mathsf{B}[1, b]$, and produces an array (string) of integers $\mathsf{M}[1, m]$, such that $\mathsf{M}[i] = j$ if and only if $\mathsf{A}[j..j + b] = \mathsf{B}$ is the ith occurrence of B in A.

For example, if we let $\mathsf{B} =$ *act* and

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22
$\mathsf{A} =$	a	t	a	c	t	t	a	c	t	g	g	a	c	t	a	c	t	a	a	a	c	t

then we would have

$$\mathsf{M} = 3, 7, 12, 15, 20.$$

It will also be convenient to view M slightly differently, as an array of fragment sizes, or distances between occurrences of B in A (equivalently differences between adjacent values in M). We denote this *fragment size domain* of M, as the array $\mathsf{R}[1, m]$, defined such that $\mathsf{R}[i] = (\mathsf{M}[i] - \mathsf{M}[i - 1])$, with $\mathsf{R}[1] = \mathsf{M}[1] - 1$. In words, R contains the distance between occurrences of B in A. Continuing with the example above, we have

$$\mathsf{R} = 2, 4, 5, 3, 5.$$

Or if we let B be `act` and A = `atacttactggactactaaact` then we would have $\mathsf{M} = 3, 7, 12, 15, 20$ and $\mathsf{R} = 2, 4, 5, 3, 5$.

2.1.3 Methods

We find alignments in four steps. First, we convert contigs from the sequence domain to the optical map domain through the process of *in silico* digestion. Second, an FM-index is built from the sequence of optical map fragment sizes. Third, we execute a modified version of the FM-index backward search algorithm discussed in Subsection 1.1.1 that allows inexact matches. As a result of allowing inexact matches, there may be multiple fragments in an optical map that could each be a reasonable match for an *in silico* digested fragment, and in order to include all of these as candidate matches, backtracking becomes necessary in the backward search. For every backward search path that maintains a non-empty interval for the entire query contig, we emit the alignments denoted by the final interval.

Converting Contigs to the Optical Map Domain

In order to find alignments for contigs relative to the optical map, we must first convert the strings of bases into the domain of optical maps, that is, strings of fragment sizes. We do this by performing an *in silico* digest of each contig, which is performing a linear search over its bases, searching for occurrences of the enzyme recognition sequence and then computing the distances between adjacent restriction sites. These distances are taken to be equivalent to the fragment sizes that would result if the contig's genomic region underwent digestion in a lab. Additionally, the end fragments of the *in silico* digested contig are removed, as the outside ends are most likely not a result of the optical map restriction enzyme digestion, but rather an artifact of the sequencing and assembly process.

Building an FM-index from Optical Mapping Data

We construct the FM-index for ℓ, the string of fragment sizes. The particular FM-index implementation we use is the SDSL-Lite[4] [38] library's *compressed suffix array with integer wavelet tree* data structure[5].

In preparation for finding alignments, we also keep two auxiliary data structures. The first is the suffix array, SA_F, corresponding to our FM-index, which we use to report the positions in ℓ where alignments of a contig occur. While we could decode the relevant entries of SA on demand with the FM-index in $\mathcal{O}(p)$ time, where p is the so-called sample period of the FM-index, storing SA explicitly significantly improves runtime at the cost of a modest increase in memory usage. The second data structure we store is M, which allows us to map from positions in ℓ to positions in the original genome in constant time.

Alignment of Contigs Using the FM-index

After constructing the FM-index of the optical map, we find alignments between the optical map and the *in silico* digested contigs.

Specifically, we try to find substrings of the optical map fragment sequence ℓ that are similar to the string of each *in silico* digested contig's non-end fragments F satisfying an alignment goodness metric suggested by Nagarajan et al. [34] [6]:

$$\left|\sum_{i=s}^{t} F_i - \sum_{j=u}^{v} \ell_j\right| \leq F_\sigma \sqrt{\sum_{j=u}^{v} \sigma_j^2},$$

where a parameter F_σ will affect the precision/recall tradeoff.

This computation is carried out using a modified FM-index backward search. A simplified, recursive version of our algorithm for finding alignments is shown in Figure 2.2. The original

[4]https://github.com/simongog/sdsl-lite.

[5]The exact revision we used was commit ae42592099707bc59cd1e74997e635324b210115.

[6]N.B. Alternative goodness metrics could be substituted. They must satisfy the property that pairs of strings considered to align well are composed of substrings that are also considered to align well would also work.

FM-index backward search proceeds by finding a succession of intervals in the suffix array of the original text that progressively match longer and longer suffixes of the query string, starting from the rightmost symbol of the query. Each additional symbol in the query string is matched in a process taking two arguments: 1) a suffix array interval, the Υ-interval, corresponding to the suffixes in the text, ℓ, whose prefix matches a suffix of the query string, and 2) an extension symbol c. The process returns a new interval, the $c\Upsilon$-interval, where a prefix of each text suffix corresponding to the new interval is a left extension of the previous query suffix. This process is preserved in TWIN, and is represented by the function *BackwardSearchOneSymbol* in the TWIN algorithm, displayed in Figure 2.2.

Since the optical map fragments include error from the measurement process, it cannot be assumed an *in silico* fragment size will exactly match the optical map fragment size from the same locus in the genome. To accommodate these differences, we determine a set of distinct candidate match fragment sizes, D, each similar in size to the next fragment to be matched in our query. These candidates are drawn from the interval of the BWT currently active in our backward search. We do this by a wavelet tree traversal function provided by SDSL-Lite, which implements the algorithm described in [13] and takes $\mathcal{O}(|D| \log(f/\Delta))$ time. This is represented by the function *RestrictedUniqueRangeValues* in Figure 2.2. We emphasise that, due to the large alphabet of ℓ, the wavelet tree's ability to list unique values in a range efficiently is vital to overall performance. Unlike in other applications where the FM-index is used for approximate pattern matching (e.g. read alignment), we cannot afford a bruteforce enumeration of the alphabet at each step in the backward search.

These candidates are chosen to be within a reasonable noise tolerance, t, based on assumptions about the distribution of optical measurement error around the true fragment length. Since there may be multiple match candidates in the BWT interval of the optical map for a query fragment, we extend the backward search with backtracking so each candidate size computed from the wavelet tree is evaluated. That is, for a given *in silico* fragment size (i.e. symbol) c, every possible candidate fragment size, c', that can be found in the optical map in the range $c - t \ldots c + t$ and in the interval

$s \ldots e$ (of the BWT) for some tolerance t is used as a substitute in the backward search. Each of these candidates is then checked to ensure that a left extension would still satify the goodness metric, and then used as the extension symbol in the backward search. So it is actually a set of c'Y-intervals that is computed as the left extension in TWIN. Additionally, small DNA fragments may not adhere sufficiently to the glass surface and can be lost in the optical mapping process, so we also branch the backtracking search both with and without small *in silico* fragments to accommodate the uncertainty.

Each time the backward search algorithm successfully progresses throughout the entire query (i.e. it finds some approximate match in the optical map for each fragment in the contig query), we take the contents of the resulting interval in the SA as representing a set of likely alignments.

Output of Alignments in PSL format

For each *in silico* digested contig that has an approximate match in the optical map, we emit the alignment, converting positions in the fragment string ℓ to positions in the genome using the M table. We provide a script to convert the human readable output into PSL format.

2.1.4 Results

We evaluated the performance of TWIN against the best competing methods on *Yersinia kristensenii*, rice and budgerigar. These three genomes were chosen because they have available sequence and optical mapping data and are diverse in size. For each dataset, we compared the runtime, peak memory usage, and the number of contigs for which at least one alignment was found for TWIN, SOMA [34], and the software of Valouev et al. [33]. Peak memory was measured as the maximum resident set size as reported by the operating system. Runtime is the user process time, also reported by the operating system. SOMA [34] v2.0 was run with example parameters provided with the tool and the software of Valouev et al. [33] was run with its scoring parameters object constructed with arguments (0.2, 2, 1, 5, 17.43, 0.579, 0.005, 0.999, 3, 1). TWIN was run with $D_\sigma = 4$, $t = 1000$, and $[250 \ldots 1000]$ for the range of small fragments. Gentig [20] and BACop [27] were not available for download so we did not test the data using these approaches.

procedure MATCH(s,e,q,h)
  if $h = -1$ **then**
    ▷ Recursion base case. Suffix array indexes $s..e$ denote original query matches.
    Emit(s, e)
  else
    ▷ The next symbol to match, c, is the last symbol in the query string.
    $c \leftarrow q[h]$
    ▷ Find the approximately matching values in $\mathsf{BWT}[s \ldots e]$, within tolerance t.
    $D \leftarrow$ *RestrictedUniqueRangeValues*(s, e, $c + t$, $c - t$)
    ▷ Let c' be one possible substitute for c drawn from D
    for all $c' \in D$ **do**
      ▷ If Equation 1 is still satisified with c' and c, ...
      if $\left|\sum_{i=0}^{|q|-h} \mathsf{SA}[s]_i + c' - \sum_{j=h}^{|q|-1} q_j - c\right| \leq F_\sigma \sqrt{\sum_{j=0}^{|q|-h} \sigma_j^2}$ **then**
        ▷ ... determine the suffix array range of the left extension of c'.
        $s', e' \leftarrow$ *BackwardSearchOneSymbol*(s, e, c')
        ▷ Recurse to attempt to match the currently unmatched prefix.
        MATCH$(s', e', q, h - 1)$

Figure 2.2: MATCH(s, e, q, h) Provided a suffix array start index s and end index e, query string q, and rightmost unmatched query string index h (initially $s = 1$, $e = m$, $h = |q| - 1$), emit alignments of an *in silico* digested contig to an optical map.

The sequence data was assembled for *Yersinia kristensenii*, rice and budgerigar by using various assemblers. The relevant assembly statistics are given in Table 2.1. An important statistic in this table is the number of contigs that have at least two restriction sites, since contigs with fewer than two are unable to be aligned meaningfully by any method, including TWIN. This statistic was computed to reveal cases of ambiguity in placement from lack of information. Indeed, Assemblathon 2 required there to be nine restriction sites present in a contig to align it to the optical mapping data [37]. All experiments were performed on Intel x86-64 workstations with sufficient RAM to avoid paging, running 64-bit Linux.

Table 2.1: Assembly and genome statistics for *Yersinia kristensenii*, rice and budgerigar. The assembly statistics were obtained from Quast. [39].

Genome	N50	Genome Size	No. of Contigs with ≥ 2 restriction sites
Y. kristensenii	30,719	4.6 Mb	92
Rice	5,299	430 Mb	3,103
Budgerigar	77,556	1.2 Gb	10,019

The experiments for *Yersinia kristensenii*, rice and budgerigar illustrate how each of the programs' running time scale as the size of the genome increases. However, due to the possibility of mis-assemblies in these draft genomes, comparing the actual alignments could possibly lead to erroneous conclusions. Therefore, we will verify the alignments using simulated *E. coli* data. See Subsection 2.1.4 for this experiment.

Performance on *Yersinia kristensenii*

The sequence and optical map data for *Yersinia kristensenii* are described by Nagarajan *et al.* [34]. The *Yersinia kristensenii* ATCC 33638 reads were generated using 454 GS 20 sequencing and assembled using SPAdes version 3.0.0 [40] using default parameters. Contigs from this assembly were aligned against an optical map of the bacterial strain generated by OpGen using the AfIII restriction enzyme. There are approximately 1.4 million single-end reads for this dataset, and they were obtained from the NCBI Short Read Archive (accession SRX013205). Of the 92 contigs that could be aligned to the optical map, the software of Valouev et al. aligned 91 contigs, SOMA aligned 54 contigs, and TWIN aligned 61 contigs. Thus, TWIN found more alignments than SOMA, and did so faster. It should be noted that, for this dataset, all three tools had reasonable runtimes. However, while the software of Valouev et al. found more alignments, our validation experiments (below) suggest these results may favor recall over precision, and many of the additional alignments may not be credibled.

Performance on Rice Genome

The second dataset consists of approximately 134 million 76 bp paired-end reads from *Oryza sativa Japonica* rice, generated by Illumina, Inc. on the Genome Analyzer (GA) IIx platform, as described by Kawahara *et al.* [41]. These reads were obtained from the NCBI Short Read Archive (accession SRX032913) and assembled using SPAdes version 3.0.0 [40] using default parameters. The optical map for rice was constructed by Zhou *et al.* [26] using SwaI as the restriction enzyme. This optical map was assembled from single molecule restriction maps into 14 optical map contigs,

labeled as 12 chromosomes, with chromosome labels 6 and 11 both containing two optical map contigs.

Again, TWIN found alignments for more contigs than SOMA on the rice genome. SOMA and TWIN found alignments for 2,434, and 3,098 contigs, respectively, out of 3,103 contigs that could be aligned to the optical map. However, while SOMA required over 29 minutes to run, TWIN required less than one minute. The software of Valouev executed faster than SOMA (taking around 3 minutes), though still several times slower than TWIN on this modest sized genome.

Performance on Budgerigar Genome

The sequence and optical map data for the budgerigar genome were generated for the Assemblathon 2 project of Bradnam *et al.* [37]. Sequence data consists of a combination of Roche 454, Illumina, and Pacific Biosciences reads, providing 16x, 285x, and 10x coverage (respectively) of the genome. All sequence reads are available at the NCBI Short Read Archive (accession ERP002324). For our analysis we consider the assembly generated using Celera [42], which was completed by the CBCB team (Koren and Phillippy) as part of Assemblathon 2 [37]. The optical mapping data was created by Zhou, Goldstein, Place, Schwartz, and Bechner using the SwaI restriction enzyme and consists of 92 separate pieces. As with the two previous data sets, TWIN found alignments for more contigs than SOMA on the budgerigar genome. SOMA and TWIN found alignments for 9,668, and 9,826 contigs, respectively, out of 10,019 contigs that could be aligned to the optical map. However, SOMA required over 77 days of CPU time and TWIN required 35 minutes. The software of Valouev et al. returned 9,814 alignments and required over an order of magnitude (6.5 hours) of CPU time. Hence, TWIN was the only method that efficiently aligned the *in silico* digested budgerigar genome contigs to the optical map. It should be kept in mind that the competing methods were developed for prokaryote genomes and so we are repurposing them at a scale for which they were not designed. Lastly, the amount of memory used by all the methods on all experiments was low enough for them to run on a standard workstation.

We were forced to parallelize SOMA due to the enormous amount of CPU time SOMA required for this dataset. To accomplish this task, the FASTA file containing the contigs was split into 300

different files, and then IPython Parallel library was used to invoke up to two instances of SOMA on each machine from a set of 150 machines. Thus, when using a cluster with up to 300 jobs concurrently, the alignment for the budgerigar genome took about a day of wall clock time. In contrast, we ran the software of Valouev et al. and TWIN with a single thread running on a single core. However, it should be noted that the same parallelization could have been accomplished for both these software methods too. Also, even with parallelization of SOMA, TWIN is still an order of magnitude faster than it.

Table 2.2: Comparsion of the alignment results for TWIN and competing method. The performance of TWIN was compared against SOMA [34] and the method of Valouev et al. [33] using the assembly and optical mapping data for *Yersinia Kristensenii*, rice, and budgerigar. Various assemblers were used to assemble the data for these species. The relevant statistics and information concerning these assemblies and genomes can be found in Table 2.1. The peak memory is given in megabytes (mb). The running time is reported in seconds (s), minutes (m), hours (h), and days.

Genome	Program	Memory	Time	Aligned Contigs
Y. Kristensenii				
	Valouev *et al.*	1.81	.17 s	91
	SOMA	1.71	7.32 s	54
	TWIN	18	.06 s	65
Rice				
	Valouev *et al.*	11.25	2 m 57 s	2,676
	SOMA	7.94	29 m 38 s	2,434
	TWIN	18.25	50 s	3,098
Budgerigar				
	Valouev *et al.*	390	6.5 h	9,814
	SOMA	380.95	77.2 d	9,668
	TWIN	127.112	35 m	9,826

Alignment Verification

We compared the alignments given by TWIN against the alignments of the contigs of an *E. coli* assembly to the *E. Coli* (str. K-12 substr. MG1655) reference genome. Our prior experiments involved species for which the reference genome may have regions that are mis-asssembled and therefore, contig alignments to the reference genome may be inaccurate and cannot be used for comparison and verification of the *in silico* digested contig alignment. The *E. coli* reference

genome is likely to contain the fewest errors and thus, is the one we used for assembly verification. The sequence data consists of approximately 27 million paired-end 100 bp reads from *E. coli* (str. K-12 substr. MG1655) generated by Illumina, Inc. on the Genome Analayzer (GA) IIx platform, and was obtained from the NCBI Short Read Archive (accession ERA000206), and was assembled using SPAdes version 3.0.0 [40] using default parameters. This assembly consists of 160 contigs; 50 of which contain two restriction sites, the minimum required for any possible optical alignment, and complete alignments with minimal (<800 bp) total in/dels relative to the reference genome.

We simulated an optical map using the reference genome for *E. coli* (str. K-12 substr. MG1655) since there is no publicly available one for this genome.

The 50 contigs that contained more than two restriction sites were aligned to the reference genome using BLAT [43]. These same contigs were then *in silico* digested and aligned to the optical map using TWIN. The resulting PSL files were then compared. TWIN found alignment positions within 10% of those found by BLAT for all 50 contigs, justifying that our method is finding correct alignments. We repeated this verification approach with both SOMA and the software from Valouev. All of SOMA's reported alignments had matching BLAT alignments, while of the 49 alignments the software from Valuoev reported, only 18 could be matched with alignments from BLAT.

2.1.5 Discussion and Conclusions

We demonstrated that TWIN, an index-based algorithm for aligning *in silico* digested contigs to an optical map, gave over an order of magnitude improvement to runtime without sacrificing alignment quality. Our results show that we are able to handle genomes at least as large as the budgerigar genome directly, whereas SOMA cannot feasibly complete the alignment for this genome in a reasonable amount of time without significant parallelization, and even then is orders of magnitude slower than TWIN. Indeed, given its performance on the budgerigar genome, and its $\mathcal{O}(m^2n^2)$ time complexity, larger genomes seem beyond SOMA. For example, the loblolly pine

tree genome, which is approximately 20 Gb [44], would take SOMA approximately 84 machine years, which, even with parallelization, is prohibitively long.

Optical mapping is a relatively new technology, and thus, with so few algorithms available for working with this data, we feel there remains good opportunities for developing more efficient and flexible methods. Dynamic programming optical map alignment approaches are still important today, as the assembly of the consensus optical maps from the individually imaged molecules often has to deal with missing or spurious restriction sites in the single molecule maps when enzymes fail to digest a recognition sequence or the molecule breaks. Though coverage is high (e.g. about 1,241 Gb of optical data was collected for the 2.66 Gb goat genome), there may be cases where missing restriction site errors are not resolved by the assembly process. In these rare cases (only 1% of alignments reported by SOMA on parrot contain such errors) they will inhibit TWIN's ability to find correct alignments. In essence, TWIN is trading a small degree of sensitivity for a huge speed increase, just as other index based aligners have done for sequence data. Sirén et al. [45] recently extended the Burrows-Wheeler transform (BWT) from strings to acyclic directed labeled graphs and to support path queries. In Section 2.2 we'll examine an adaptation of this method for optical map alignment that allows for the efficient handling of missing or spurious restriction sites.

In later work [46] we showed an ensemble of optical map alignments produced by TWIN and alignments of reads to contigs could produce superior statistical performance on misassembly detection.

2.2 Kohdista: A Succinct Solution to Raw Optical Map Alignment[7]

In this section, we build on the work from TWIN to solve a noisier form of optical mapping data. In practice, the method developed here could aid approaches like misSEQuel because

[7]Martin D. Muggli, Simon J. Puglisi, and Christina Boucher. A Succinct Solution to Rmap Alignment. In Laxmi Parida and Esko Ukkonen, editors, *18th International Workshop on Al- gorithms in Bioinformatics (WABI 2018)*, volume 113 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 12:1âĂŞ12:16, Dagstuhl, Germany, 2018. Schloss DagstuhlâĂŞ Leibniz-Zentrum fuer Informatik.

misSEQuel takes a whole genome optical map as input, but these themselves must be assembled by an overlap-layout-consensus process from the data described in this section.

2.2.1 Introduction

Genome-wide optical maps are ordered high-resolution restriction maps that give the position of occurrence of restriction cut sites corresponding to one or more restriction enzymes. These genome-wide optical maps are assembled using an overlap-layout-consensus approach using raw optical map data, which are referred to as *Rmaps*. Hence, Rmaps are akin to reads in genome sequencing. To date, however, there is no efficient, non-proprietary method for finding pairwise alignments between Rmaps, which is the first step in assembling genome-wide maps.

Several existing methods are superficially applicable to Rmap pairwise alignments but all programs either struggle to scale to even moderate size genomes or require significant further adaptation to the problem. Several methods exhaustively evaluate all pairs of Rmaps using dynamic programming. One of these is the method of Valouev *et al.* [33], which is capable of solving the problem exactly but requires over 100,000 CPU hours to compute the alignments for rice [47]. The others are SOMA [34] and MalignerDP [48] which are designed only for semi-global alignments instead of overlap alignments, which are required for assembly.

Other methods reduce the number of map pairs to be individually considered by initially finding seed matches and then extending them through more intensive work. These include OMBlast [49], OPTIMA [50], and MalignerIX [48]. These, along with MalignerDP, were designed for a related alignment problem of aligning consensus data but cannot consistently find high quality Rmap pairwise alignments in reasonable time as we show later. This is unsurprising since these methods were designed for either already assembled optical maps or *in silico* digested sequence data for one of their inputs, both having a lower error rate than Rmap data.

Our contributions. In this paper, we present a fast, error-tolerant method for performing pairwise Rmap alignment that makes use of a novel FM-index based data structure. Although the FM-index can naturally be applied to short read alignment [1, 51], it is nontrivial to apply it

to Rmap alignment. The difficulty arises from: (1) the abundance of missing or false cut sites, (2) the fragment sizes require inexact fragment-fragment matches (e.g. 1,547 bp and 1,503 bp represent the same fragment), (3) the Rmap sequence alphabet consists of all unique fragment sizes and is so extremely large (e.g., over 16,000 symbols for the goat genome). The second two challenges render inefficient the standard FM-index backward search algorithm, which excels at exact matching over small alphabets. The first (and most-notable) challenge requires a more complex index-based data structure be used to create an aligner that is robust for insertion and deletion of cut sites. To overcome the mismatch cut site challenge while still accommodating the other two, we develop KOHDISTA, an index-based Rmap alignment program that is capable of finding all pairwise alignments in large eukaryote organisms.

We first abstract the problem to that of approximate-path matching in a directed acyclic graph (DAG). The KOHDISTA method then indexes a set of Rmaps represented as a DAG, using a modified form of the *generalized compressed suffix array (GCSA)*, which is a derivative of the FM-index developed by Sirén *et al.* [45]. The principle insight of our work is that while GCSA is able to efficiently match all similar paths concurrently, it was designed for indexing variations observed in a collection of sequences. In contrast, our work indexes variations that are instead speculative, based on the Rmap error profile. Lastly, we demonstrate that challenges posed by the inexact fragment sizes and alphabet size can be overcome, specifically in the context of the GCSA, via careful use of a wavelet tree [13, 52].

We verify our approach on simulated *E. coli* Rmap data by showing that KOHDISTA achieves similar sensitivity and specificity to Valouev *et al.*, and with more permissive alignment acceptance criteria 90% of Rmap pairs simulated from overlapping genomic regions. We also show the utility of our approach on larger eukaryote genomes by demonstrating that existing published methods require more than 151 hours of CPU time to find all pairwise alignments in the plum Rmap data; whereas, KOHDISTA requires 31 hours. Thus, we present the first fully-indexed method capable of finding all match patterns in the pairwise Rmap alignment problem.

2.2.2 Background

More Details of Optical Mapping.

A detail omitted in TWIN is that the whole genome restriction map R is actually a consensus sequence formed from millions of erroneous Rmap sequences. The optical mapping system produces millions of Rmaps for a single genome. It is performed on many cells of an organism and for each cell there are thousands of Rmaps (each at least 250 Kbp in length in publicly available data). These Rmaps must then be assembled to produce a genome-wide optical map which can then be used in downstream tools such as TWIN. Like the final R sequence, each Rmap is an array of lengths — or fragment sizes — between occurrences of B in A.

There are three types of errors that an Rmap (and hence with lower magnitude and frequency, also the consensus map) can contain: (1) missing and false cuts, which are caused by an enzyme not cleaving at a specific site, or by random breaks in the DNA molecule, respectively; (2) missing fragments that are caused by *desorption*, where small (< 1 Kbp) fragments are lost and so not detected by the imaging system; and (3) inaccuracy in the fragment size due to varying fluorescent dye adhesion to the DNA and other limitations of the imaging process. Continuing again with the example above where $\mathsf{R} = 2, 4, 5, 3, 5$ is the error-free Rmap: an example of an Rmap with the first type of error could be $\mathsf{R}' = 6, 5, 3, 5$ (the first cut site is missing so the fragment sizes 2, and 4 are summed to become 6 in R'); an example of a Rmap with the second type of error would be $\mathsf{R}'' = 2, 4, 3, 5$ (the third fragment is missing); and lastly, the third type of error could be illustrated by $\mathsf{R}''' = 2, 4, 7, 3, 5$ (the size of the third fragment is inaccurately given).

Frequency of Errors. In the optical mapping system, there is a 20% probability that a cut site is missed and a 0.15% probability of a false break per Kbp, i.e., error type (1) occurs in a fragment. Popular resticition enzymes in optical mapping experiments recognize a 6 bp sequence giving an expected cutting density of 1 per 4096 bp. At this cutting density, false breaks are less common than missing restriction sites (approx. $0.25 * .2 = .05$ for missing sites vs. 0.0015 for false sites per bp). The inaccuracy of the fragment sizes, i.e, error type (3), follows a normal distribution with mean and variance assumed to be 0 bp and $\ell\sigma^2$ ($\sigma = .58$ kbp), respectively [33].

2.2.3 The Pairwise Rmap Alignment Problem

Given a genome $\mathsf{A}[1, n]$ and a restriction enzyme's recognition sequence $\mathsf{B}[1, b]$, the optical mapping system produces Rmaps, which are arrays of lengths—or fragment sizes—between occurrences of B in A. The background section provides details on the optical mapping process. Producing Rmap data is an error prone process. Thus, three types of errors can occur: (1) missing and false cuts that delimit fragments; (2) missing fragments; and (3) inaccuracy in the fragment sizes. For example, let $\mathsf{R} = 2, 4, 5, 3, 5$ be an error-free Rmap, then an example of an Rmap with the first type of error could be $\mathsf{R}' = 6, 5, 3, 5$ (the first cut site is missing so the fragment sizes 2, and 4 are summed to become 6 in R'); an example of a Rmap with the second type of error would be $\mathsf{R}'' = 2, 4, 3, 5$ (the third fragment is missing); and lastly, the third type of error could be illustrated by $\mathsf{R}''' = 2, 4, 7, 3, 5$ (the size of the third fragment is inaccurately given)

The pairwise Rmap alignment problem aims to align one Rmap (the *query*) R_q against the set of all other Rmaps in the dataset (the *target*). We denote the target database as $\mathsf{R}_1 \ldots \mathsf{R}_n$, where each R_i is a sequence of m_i fragment sizes, i.e, $\mathsf{R}_i = [f_{i1}, .., f_{im_i}]$. An alignment between two Rmaps is a relation between them comprising groups of zero or more consecutive fragment sizes in one Rmap associated with groups of zero or more consecutive fragments in the other. For example, given $\mathsf{R}_i = [4, 5, 10, 9, 3]$ and $\mathsf{R}_j = [10, 9, 11]$ one possible alignment is $\{[4, 5], [10]\}, \{[10], [9]\}, \{[9], [11]\}, \{[3], []\}$. A group may contain more than one fragment (e.g. $[4, 5]$) when the restriction site delimiting the fragments is absent in the corresponding group of the other Rmap (e.g $[10]$). This can occur if there is a false restriction site in one Rmap, or there is a missing restriction site in the other. Since we cannot tell from only two Rmaps which of these scenarios occurred, for the purpose of our remaining discussion it will be sufficient to consider only the scenario of missed (undigested) restriction sites.

2.2.4 Methods

We now describe the algorithm behind KOHDISTA. Three main insights enable our index-based aligner for Rmap data: 1) abstraction of the alignment problem to a finite automaton; 2) use of the

GCSA for storing and querying the automaton; and 3) modification of backward search to use a wavelet tree in specific ways to account for the Rmap error profile.

Finite Automaton

Continuing with the example in the background section, we want to align $\mathsf{R}' = 6, 5, 3, 5$ to $\mathsf{R}''' = 2, 4, 7, 3, 5$ and vice versa. To accomplish this we cast the Rmap alignment problem to that of matching paths in a finite automaton. A finite automaton is a directed, labeled graph that defines a *language*, or a specific set of sequences composed of vertex labels. A sequence is recognized by an automaton if it contains a matching path: a consecutive sequence of vertex labels equal to the sequence. We represent the target Rmaps as an automaton and the query as a path in this context.

The automaton for our target Rmaps can be constructed as follows. First concatenate the $\mathsf{R}_1 \ldots \mathsf{R}_n$ together into a single sequence with each Rmap separated by a special symbol which will not match any query symbol. Let R^* denote this concatenated sequence. Hence, $\mathsf{R}^* = [f_{11}, .., f_{1m_1}, \ldots, f_{n1}, .., f_{nm_n}]$. Then, construct an initial finite automaton $\mathsf{A} = (V, E)$ for R^* by creating a set of vertices $v_1^i..v_m^i$, one vertex labeled with each fragment length and edges connecting them. Also, introduce to A a *starting vertex* v_1 labeled with $\#$ and a *final vertex* v_f labeled with the character $\$$. All other vertices in A are labeled with integral values. This initial set of vertices and edges is called the *backbone*. The backbone by itself is only sufficient for finding alignments with no missing cut sites in the query. The backbone of an automaton constructed for a set containing R' and R'' would be $\#, 6, 5, 3, 5, 999, 2, 4, 3, 5\$$, using 999 as an unmatchable value. Next, extra vertices ("skip vertices") and extra edges are added to A to allow for the automaton to accept all valid queries. Figure 2.3(a) illustrates the construction of A for a single Rmap with fragment sizes $2, 3, 4, 5, 6$.

Skip Vertices and Skip Edges

We introduce extra vertices labeled with *compound fragments* to allow missing cut sites (first type of error) to be taken into account in querying the target Rmaps. We refer to these as *skip vertices* as they provide alternative path segments which skip past two or more backbone vertices.

Thus, we add a *skip vertex* to A for every $o + 1$ length run of consecutive vertices in the backbone where $1 < o < order$ and *order* is the maximum number of consecutive missed cut sites to be accommodated. First order skip vertices are each labeled with the sum of two consecutive backbone vertices. Second order skip vertices are each labeled with the sum of three consecutive backbone vertices. The vertex labeled with 7 connecting 2 and 5 in 2.3(a) is an example of a skip vertex. Likewise, $5, 9, 11$ are other skip vertices.

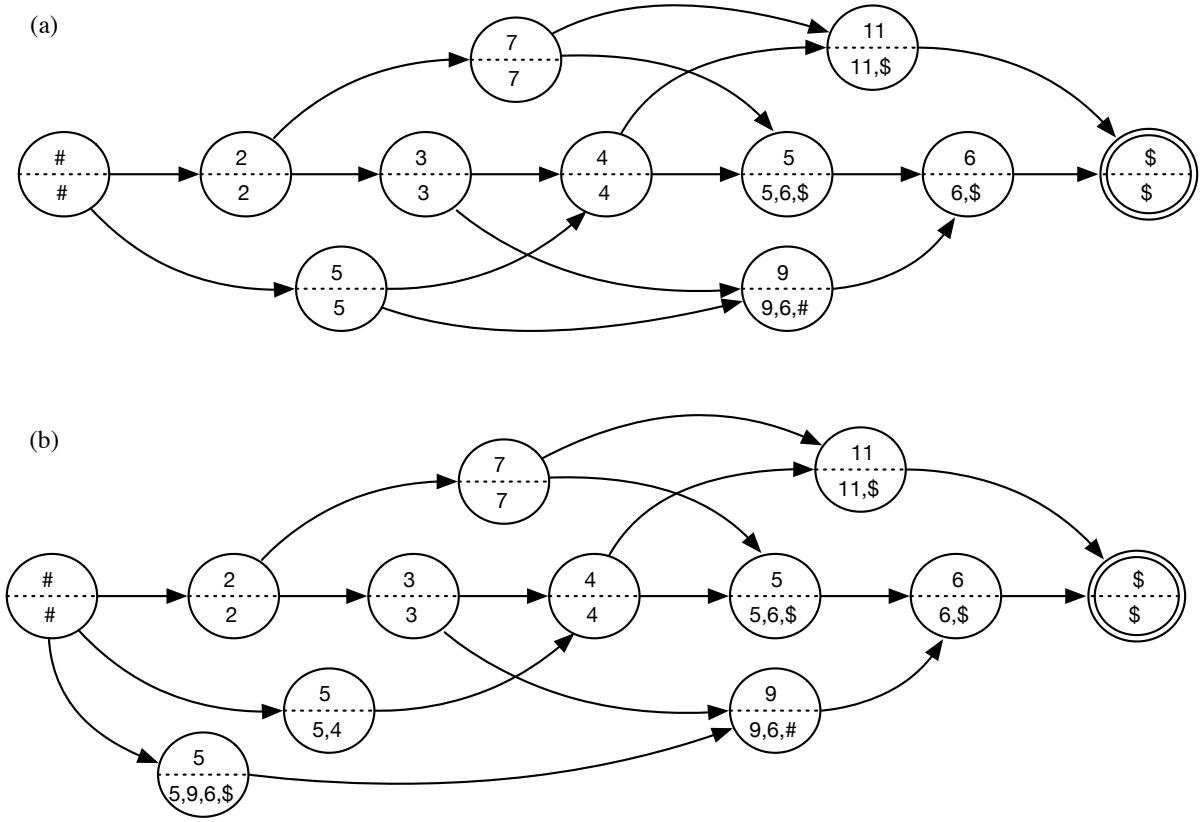


Figure 2.3: An example automaton for an Rmap with fragment size sequence $2, 3, 4, 5, 6$. The memory representation is shown in Table 2.3. The top half of vertices contains the label, which models a fragment size in Kbp. The common prefixes of all suffixes spellable from a vertex is written in the bottom half. Note that there is no ordering of vertices such that all their corresponding suffixes are in lexicographic order; the leftmost vertex labelled with "5" spells suffixes beginning "5,4,..." as well as the suffix "5,9,6,$" while the rightmost 5 spells the suffix "5,6,$". (b) shows the prefix sorted automaton corresponding to the one in (a). The leftmost vertex 5 has been duplicated and the outgoing edges of the previous version have been divided between the new replacement instances. This also divides the suffixes spellable from the prior version. Now the three 5 vertices can be ordered based on their common prefixes as ["5,4,...","5,6,$", "5,9,6, $"].

Table 2.3: Table listing the three arrays storing the automaton shown in Figure 2.3 in memory: BWT, M, and F.

	$	2	3	4	5,4	5,6,$	5,9,6,$	6,$	7	9,6,$	11,$	#
BWT	6,11	#	2	3,5	#	4,7	#	5,9	2	3,5	4,7	$
M	1	10	10	10	1	1	1	1	10	1	1	100
F	10	1	1	10	1	10	1	10	1	10	10	1

Finally, we add *skip edges* which provide paths around vertices with small labels in the backbone. These allow a query with a missing fragment to still match.[8] Hence, the addition of skip edges allow for desorption (the second type of error) to be taken into account in querying the target Rmaps.

Generalized Compressed Suffix Array

We index the automaton with the GCSA [45] for efficient storage and path querying. The GCSA is a generalization of the FM-index for automata and we will explain the GCSA by drawing on the definition of the (more widely known) FM-index.

To generalize the FM-index to automata (from strings), we need to efficiently store the vertices and edges in a manner such that the FM-index properties still hold, allowing the GCSA to support queries efficiently. An FM-index's compressed suffix array for a string X encodes a relationship between each suffix Y and its left extension. Hence, this suffix array can be generalized to edges in a graph that represent a relationship between vertices. The compressed suffix array for a string is a special case where the vertices are labeled with the string's symbols in a non-branching path.

Prefix-sorted Automata

Just as backward search for strings is linked to suffix sorting, backward searching in the BWT of the automaton requires us to be able to sort the vertices (and a special set of the paths) of the automaton in a particular way. In [45] this property is called *prefix-sortedness*. Let $A = (V, E)$ be a finite automaton, let $v_{|V|}$ denote its terminal vertex, and let $v \in V$ be a vertex. We say v is

[8] Different smallness thresholds for query and target bias toward this scenario, avoiding backtracking in the search.

prefix-sorted by prefix $p(v)$ if the labels of all paths from v to $v_{|V|}$ share a common prefix $p(v)$, and no path from any other vertex $u \neq v$ to $v_{|V|}$ has $p(v)$ as a prefix of its label. If all vertices V are prefix-sorted then Automaton A is prefix-sorted. See Figure 2.3 for an example of a non-prefix sorted automaton and a prefix sorted automaton. A non-prefix sorted automaton can be made prefix sorted through a process of duplicating vertices and their incoming edges but dividing their outgoing edges between the new instances (see [45]).

Clearly the prefixes $p(v)$ allow us to sort the vertices of a prefix-sorted automaton into lexicographical order. Moreover, if we consider the list of outgoing edges (u, v), sorted by pairs $(p(u), p(v))$, they are also sorted by the sequences $\ell(u)p(v)$, where $\ell(u)$ denotes the label of vertex u. This (dual sortedness) property allows backward searching to work over the list of vertex labels (sorted by $p(v)$) in the same way that is does for the symbols of a string ordered by their following suffixes in normal backward search for strings.

Each vertex has a set of one or more preceding vertices and therefore, a set of predecessor labels in the automaton. These predecessor label sets are concatenated to form the automaton analog of the BWT, or ABWT. The sets are concatenated in the order defined by the above mentioned lexicographic ordering of the vertices. Each element in ABWT then denotes an edge in the automaton. An array of bits, I[9], marks a '1' for the first element of ABWT corresponding to a vertex and a '0' for all subsequent elements in that set. Thus, the predecessor labels, and hence the associated edges, for a vertex with rank r are ABWT[select(r)..select$(r+1)$]. Another array, O[10], stores the out degree of each vertex and allows the set of vertex ranks associated with a ABWT interval to be found using rank() queries.

Exact Matching: GCSA Backward Search

Exact matching with the GCSA is similar to the standard FM-index backward search algorithm. As outlined in the background section, FM-index backward search proceeds by finding

[9] I was denoted F in the original GCSA paper.

[10] O was denoted M in the original GCSA paper

a succession of lexicographic ranges that progressively match longer and longer suffixes of the query string, starting from the rightmost symbol of the query. The search maintains two items — a lexicographic range and an index into the query string — and the property that the path prefix associated with the lexicographic range is equal to the suffix of the query marked by the query index. Initially, the query index is at the rightmost symbol and the range is $[1..n]$ since every path prefix matches the empty suffix. The search continues using GCSA's backward search step function, which takes as parameters the next symbol (to the left) in the query (i.e. fragment size in R_q) and the current range, and returns a new range. The query index is advanced leftward after each backward search step. In theory, since the current range corresponds to a consecutive range in the ABWT, the backward search could use $\mathsf{select}()$ queries on a bit vector I to determine all the edges adjacent to a given vertex and then two FM-index $\mathsf{LF}()$ queries are applied to the limits of the current range to obtain the new one. GCSA's implementation uses one succinct bit vector per alphabet symbol to encode which symbols precede a given vertex instead of I. Finally, this new range, which corresponds to a set of edges, is mapped back to a set of vertices using $\mathsf{rank}()$ on the M bit vector.

Inexact Matching: GCSA Backward Search Using a Wavelet Tree

We modified GCSA backward search in the following ways: (1) we used a wavelet tree to allow efficient retrieval of substitution candidates; (2) we modified the search process to combine consecutive query fragments into compound fragments so as to match fragments in R^* missing the interposing restriction site; and (3) we introduced backtracking, in order to both try size substitution candidates as well as various combinations of compound fragments. These modifications are further detailed below.

First, in order to accommodate possible errors in fragment size, we determine a set, Z, of candidate fragment sizes that are similar to the next fragment of R_q to be matched in the query. These candidates are determined by enumerating the distinct symbols in the currently active backward

search range of the ABWT[11] using the wavelet tree algorithm of Gagie *et al.* [13]. This method was proposed by Muggli *et al.* [52] for use with an FM-index but was not directly applicable to the originally proposed implementation of GCSA. This is because some of GCSA's theoretical constructs (i.e. I) were substituted in implementation for efficiency reasons. In order to apply the aforementioned wavelet tree method, we thus resurrect the previously theoretical only bit array I (which we encode succinctly) as well as symbol array ABWT (which we encoded with a wavelet tree) into KOHDISTA using the SDSL-Lite library by Gog *et al.* [38].

To accommodate possible restriction sites that are present in the query Rmap but absent in target Rmaps, we generate compound fragments (i.e. new symbols) by summing pairs and triples of consecutive query fragment size and then querying the wavelet tree for substitutions of these compound fragments. This summing of multiple consecutive fragments is complementary to the skip vertices in the target automaton and accommodates missed restriction sites in the target, just as the skip vertices accommodate missed sites in the query.

Lastly, since there may be multiple match candidates in the ABWT interval of R^* for a compound fragment generated from R_q and multiple compound fragments generated at a given position in R_q, we employ the common practice of adding backtracking to backward search (as is done, for example in the works of Li *et al.* and Langmead *et al.*). This is so that each candidate size returned to the search algorithm from the wavelet tree is evaluated; i.e., for a given compound fragment size f generated from R_q, every possible candidate fragment size, f', that can be found in R^* in the range $f - t \ldots f + t$ and in the interval $s \ldots e$ (of the ABWT of R^*) for some tolerance t is used as a substitute in the backward search.

2.2.5 Results and Discussion

We evaluated KOHDISTA against the other available optical map alignment software. Our experiments measured runtime, peak memory, and alignment quality on simulated *E. coli* Rmaps and

[11] Recall that this active range, when applied to a lexicographic range, represents the suffixes whose prefixes are the matched portion of the query, while the same range of the ABWT contains possible extension symbols.

experimentally generated plum Rmaps. All experiments were performed on Intel Xeon computers with $\geq$ 16 GB RAM running 64-bit Linux.

Performance on Simulated E.coli Rmap Data

To verify the correctness of our method, we simulated a read set from a 4.6 Mbp *E. coli* reference genome as follows: we started with 1,400 copies of the genome, and then generated 40 random loci within each. These loci form the ends of molecules that would undergo digestion. Molecules smaller than 250 Kbp were discarded leaving 272 molecules with a combined length equating to 35x coverage depth. The cleavage sites for the XhoI enzyme were then identified within each of these simulated molecules. We removed 20% of these at random from each simulated molecule to model partial digestion. Finally, normally distributed noise was added to each fragment with a standard deviation of .58 kb per 1 kb of the fragment. Simulated molecule pairs having 16 common conserved digestion sites become the "ground truth"[12] data for testing our method with the others. Although a molecule would align to itself, these are not included in the ground truth set. This method of simulation was based on the *E. coli* statistics given by Valouev *et al.* [47] and resulting in a molecule length distribution as observed in publicly available Rmap data from OpGen, Inc.

Most of the tools were designed for less noisy data but in theory could address all the data error types required. For tools with tunable parameters, we tried aligning the *E. coli* Rmaps with combinations of parameters for each method related to its alignment score thresholds and error model parameters. We used parameterization giving results similar to those for the default parameters of Valouev *et al.*'s method to the extent such parameters did not significantly increasing each tool's runtime. These same parameterization were used in the next section on plum data.

Even with tuning, we were unable to obtain pairwise alignments on *E. coli* for two methods. We found OPTIMA only produced self alignments with its recommended overlap protocol and report its resource use in Table 2.4. For MalignerIX, even when we relaxed the parameters to

[12]Due to repeats in the restriction map, and apparent repeats at the resolution attainable through optical measurement, some alignments beyond these are expected.

account for the greater sizing error and mismatch cut site frequency, it was also only able to find self alignments. This is expected as by design it only allows missing sites in one sequence in order to run faster. Thus no further testing was performed with MalignerIX or OPTIMA. We did not test SOMA [34] as earlier investigation indicate it would not scale to larger genomes [52]. We omit TWIN [52] as it needs all cut sites to match.

Results on *E. coli* are presented in Table 2.4. KOHDISTA uses χ^2 and binomial CDF thresholds to prune the backtracking search when deciding whether to extend alignments to progressively longer alignments. More permissive match criteria, using higher thresholds, allows more Rmaps to be reached in the search and thus to be considered aligned, but it also results in less aggressive pruning in the search, thus lengthening runtime. As an example, note that when KOHDISTA was configured with a much relaxed CDF threshold of .5 and a binomial CDF threshold of .7, it found 3,925 of the 4,305 (91%) ground truth alignments, but slowed down considerably. This illustrates the index and algorithm's capability in handling all error types.

Table 2.4: Performance on simulated *E. coli* dataset. KOHDISTA (lax) demonstrates that our indexing and search method is capable of finding the majority of ground truth alignments when the search is pruned to the more relaxed thresholds of $\chi^2 < .02$, Binom. $< .5$.

Method	Time	Memory	Aligns	Recall	Precision
KOHDISTA	20 s.	19.0 MB	907	702 / 4,305 (16%)	702 / 907 (77%)
KOHDISTA (lax)	373 s.	18.3 MB	8,545	3,925 / 4,305 (91%)	3,925 / 8,545 (46%)
Valouev *et al.*	148 s.	4.0 MB	742	699 / 4,305 (16%)	699 / 742 (94%)
MalignerDP	47 s.	6.0 MB	1,959	1,296 / 4,305 (30%)	1,296 / 1959 (66%)
OMBlast	116 s.	2,078 MB	1,008	806 / 4,305 (19%)	806 / 1008 (80%)
RefAligner	31 s.	81.2 MB	992	958 / 4,305 (22%)	948 / 992 (97%)
MalignerIX	4 s.	6.0 MB	0	0 / 4,305 (0%)	0 / 0 (N/A)
OPTIMA	455 s.	10,756.5 MB	0	0 / 4,305 (0%)	0 / 0 (N/A)

Performance on Plum Rmap Data

The Beijing Forestry University and other institutes assembled the first plum (*Prunus mume*) genome using short reads and optical mapping data from OpGen Inc. We test the various available alignment methods on the 139,281 plum Rmaps from June 2011 available in the GigaScience

repository. These Rmaps were created with the BamHI enzyme and have a coverage depth of 135x of the 280 Mbp genome. For the plum dataset, we ran all the methods which approach the statistical performance of the Valouev *et al.* method when measured on *E. coli*. Thus, we omitted MalignerIX and OPTIMA because they had 0% recall and precision on *E. coli*. Our results on this plum dataset are summarized in Table 2.5.

Table 2.5: Performance on Plum.

Method	**Time**	**Memory**	**Alignments**
KOHDISTA	31 hours	7.4 GB	16,109,151
Valouev *et al.*	678 hours	60 MB	6,387
MalignerDP	214 hours	784 MB	568,744
OMBlast	151 hours	12.3 GB	424,730
RefAligner	90 hours	374 MB	10,039

KOHDISTA was the fastest and obtained more alignments than the competing methods. When configured with a χ^2 CDF threshold of .02, it took 31 hours of CPU time to test all Rmaps for pairwise alignments in the plum Rmap data. This represents a 21x speed-up over the 678 hours taken by the exhaustive Valouev *et al.* method. The other non-proprietary methods, MalignerDP and OMBlast, took 214 hours and 151 hours, respectively. These results represent a 6.9x and 4.8x speed-up over MalignerDP and OMBlast. All methods used less than 13 GB of RAM and thus, were considered practical from a memory perspective.

To measure the quality of the alignments, we scored each pairwise alignment using the scoring scheme of Valouev *et al.* and present histograms of these alignment scores in Figure 2.4. For comparison, we also scored and present the histogram for random pairs of Rmaps. The Valouev *et al.* method produces very few but high-scoring alignments and although it could theoretically be altered to produce a larger number of alignments, the running time makes this prospect impractical (678 hours). Although KOHDISTA and RefAligner produce high-quality alignments, RefAligner produced very few alignments (10,039) and required almost 5x more time to do so. OMBlast and Maligner required significantly more time and produced significantly lower quality alignments.

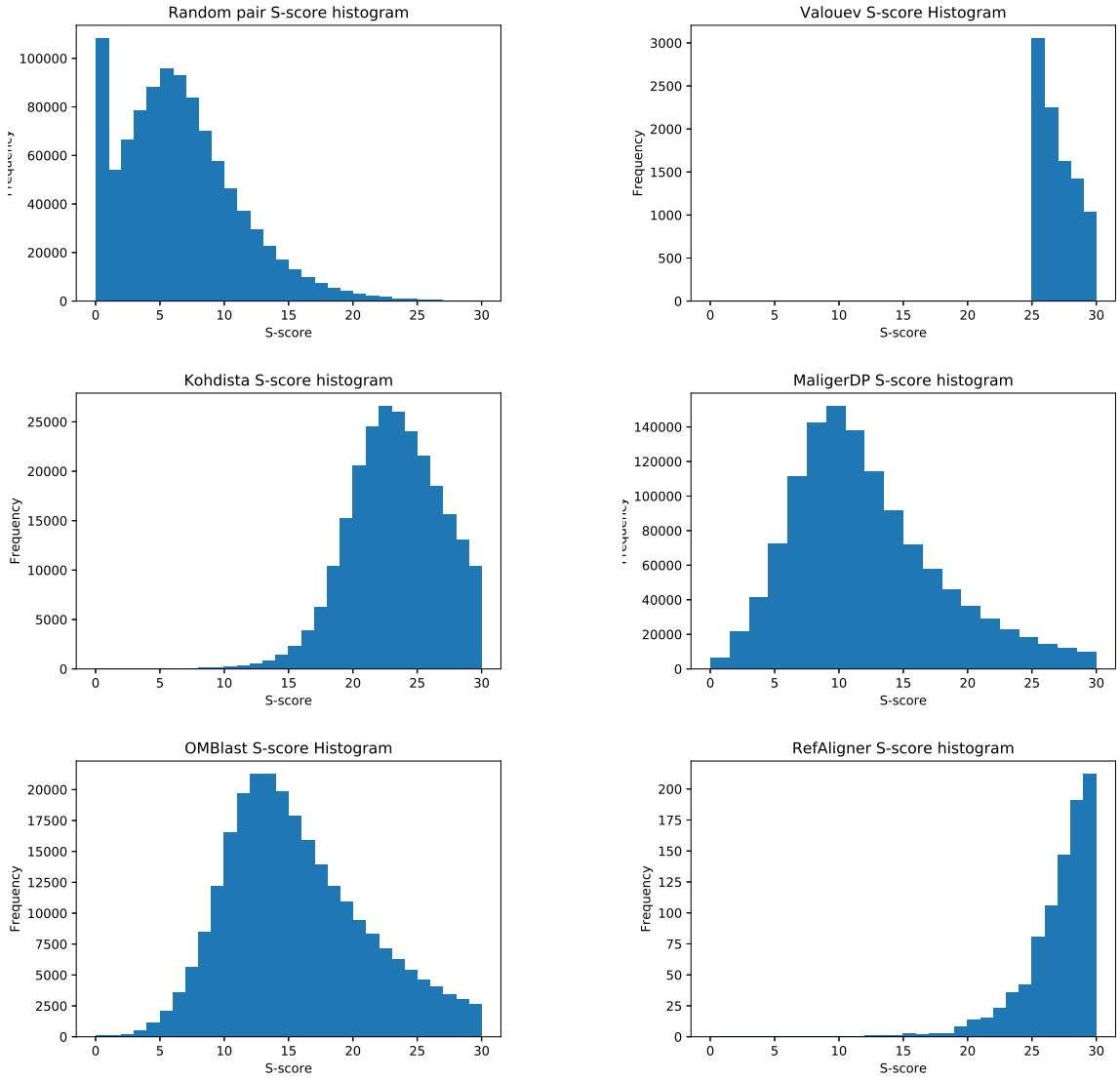


Figure 2.4: All alignments found on plum were realigned using Valouev *et al.*'s dynamic programming method. Their method finds the optimal alignment using a function balancing size agreement and cut site agreement known as an s-score. (a) The s-score distribution for random pairs. (b) The Valouev *et al.* software considers any pair with an s-score > 25 to be aligned. (c) KOHDISTA alignments tend to have significantly higher s-scores than random. (d) MaligerDP alignments tend to have slightly higher s-scores than random. (e) OMBlast alignments tend to have higher s-scores than random. (f) BioNano's commercial RefAligner method alignments tends to have a significantly higher s-scores than random.

2.2.6 Conclusion

In this section, we demonstrate how finding pairwise alignments in Rmap data can be modelled as approximate-path matching in a directed acyclic graph, and combining the GCSA with the wavelet tree results in an index-based data structure for solving this problem. We implement this

method and present results comparing KOHDISTA with competing methods. By demonstrating results on both simulated *E. coli* Rmap data and real plum Rmaps, we show that KOHDISTA is capable of detecting high scoring alignments in efficient time. In particular, KOHDISTA detected the largest number of alignments in 31 hours. RefAligner, a proprietary method, produced very few high scoring alignments (10,039) and requires almost 5x more time to do so. OMBlast and Maligner required significantly more time and produced significantly lower quality alignments. The Valouev *et al.* method produced high scoring alignments but required more than 21x time to do.

2.2.7 Practical Indexing Considerations

Pruning the Search

Alignments are found by incrementally extending candidate partial alignments (paths in the automaton) to longer partial alignments by choosing one of several compatible extension matches (adjacent vertices to the end of a path in the automaton). To perform this search efficiently, we prune the search by computing the χ^2 and binomial CDF statistics of the partial matches and use thresholds to ensure reasonable size agreement of the matched compound fragments, and the frequency of putative missing cut sites. These values alter the precision and recall as well as runtime. The statistical performance tradeoff of KOHDISTA and competing methods is shown in Figure 2.5.

Size Agreement

We use the Chi-square CDF statistic to assess size agreement. This assumes the fragment size errors are independent, normally distributed events. For each pair of matched compound fragments in a partial alignment, we take the mean between of the two as the assumed true length and compute the expected standard deviation using this mean. Each compound fragment deviates from the assumed true value by half the distance between them. These two deviation values contribute two degrees of freedom to the Chi-square calculation. Thus each deviation is normalized by dividing by the expected standard deviation, these are squared, and summed across all compound fragments

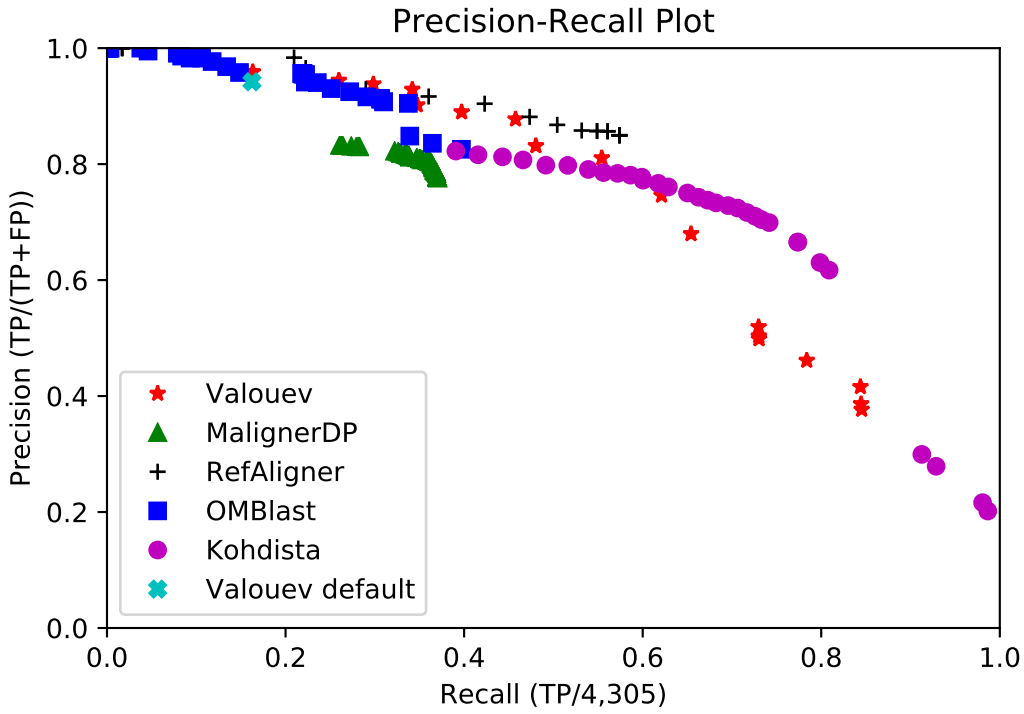


Figure 2.5: Precision-Recall plot of successful methods on simulated *E. coli*

to generate the χ^2 statistic. We use the standard χ^2 CDF function to compute the area under the curve of the probability mass function up to this χ^2 statistic, which gives the probability two Rmap segments from common genomic origin would have a χ^2 statistic no more extreme than observed. This probability is compared to KOHDISTA's chi-squared-cdf-thresh and if smaller, the candidate compound fragment is assumed to be a reasonable match and the search continues.

Cut Site Error Frequency.

We use the Binomial CDF statistic to assess the probability of the number of cut site errors in a partial alignment. This assumes missing cut site errors are independent, Bernoulli processes events. We account for all the putatively conserved cut sites on the boundaries and those delimiting compound fragments in both partially aligned Rmaps plus twice the number of missed sites as the number of Bernoulli trials. We use the standard binomial CDF function to compute the sum of the probability density function up to the number of non-conserved cut sites in a candidate match. Like the size agreement calculation above, this gives the probability two Rmaps of common

genomic origin would have the number of non-conserved sites seen or fewer in the candidate partial alignment under consideration. This is compared to the binom-cdf-thresh to decide whether to consider extensions to the given candidate partial alignment. Thus, given a set of Rmaps and input parameters ρ_L and ρ_U, we produce the set of all Rmap alignments that have a chi-square CDF statistic less than ρ_U and a binomial CDF statistic less than ρ_L. Both of these are subject to the additional constraint of a maximum consecutive missed restriction site run between aligned sites of δ and a minimum aligned site set cardinality of 16.

Pruning Queries.

One side effect of summing consecutive fragments in both the search algorithm and the target data structure is that several successive search steps with agreeing fragment sizes will also have agreeing sums of those successive fragments. In this scenario, proceeding deeper in the search space will result in wasted effort. To reduce this risk, we maintain a table of scores obtained when reaching a particular lexicographic range and query cursor pair. We only proceed with the search past this point when either the point has never been reached before, or has only been reached before with inferior scores.

Wavelet Tree Cutoff.

The wavelet tree allows efficiently finding the set of vertex labels that are predecessors of the vertices in the current match interval intersected with the set of vertex labels that would be compatible with the next compound fragment to be matched in the query. However, when the match interval is sufficiently small (< 750) it is faster to scan the vertices in ABWT directly.

Quantization.

The alphabet of fragment sizes can be large considering all the measured fragments from multiple copies of the genome. This can cause an extremely large branching factor for the initial symbol and first few extensions in the search. To improve the efficiency of the search, the fragment sizes are initially quantized, thus reducing the size of the effective alphabet and the number of

substitution candidates under consideration at each point in the search. Quantization also increases the number of identical path segments across the indexed graph which allows a greater amount of candidate matches to be evaluated in parallel because they all fall into the same ABWT interval during the search. This does, however, introduce some quantization error into the fragment sizes, but the bin size is chosen to keep this small in comparison to the sizing error.

Example Traversal

A partial search for a query Rmap [3 kb, 7 kb, 6 kb] in Figure 2.3 and Table 2.3 given an error model with a constant 1 kb sizing error would proceed with steps: 1. Start with the semi-open interval matching the empty string [0..12). 2. A wavelet tree query on ABWT would indicate the set of symbols {5, 6, 7} is the intersection of two sets: 1.) The set of symbols that would all be valid left extensions of the (currently empty) match string and 2.) The set of size appropriate symbols that match our next query symbol (i.e. 6 kb, working from the right end of our query) in light of the expected sizing error (i.e. 6kb +/- 1 kb). 3. We would then do a GCSA backward search step on the first value in the set (5) which would yield the new interval [4..7). This new interval denotes only nodes where each node's common prefix is compatible with the spelling of our current backward traversal path through the automaton (i.e. our short path of just [5] does not contradict any path spellable from any of the three nodes denoted in the match interval). 4. A wavelet tree query on the ABWT for this interval for values 7 kb +/- 1 kb would return the set of symbols 7. 5. Another backward search step would yield the new interval [8..9). At this point our traversal path would be [7, 5] (denoted as a left extension of a forward path that we are building by traversing the graph backward). The common prefix of each node (only one node here) in our match interval (i.e. [7 kb]) is compatible with the path [7, 5]. This process would continue until backward search returns no match interval or our scoring model indicates our repeatedly left extended path has grown too divergent from our query. At this point backtracking would occur to find other matches (e.g. at some point we would backward search using the value 6 kb instead of the 5 kb obtained in step 2.)

Parameters Used

We tried OPTIMA with both "p-value" and "score" scoring and the allMaps option and report the higher sensitivity "score" setting. We followed the OPTIMA-Overlap protocol of splitting Rmaps into k-mers, each containing 12 fragments as suggested in [50]. For OMBlast, we adjusted parameters maxclusteritem, match, fpp, fnp, meas, minclusterscore, and minconf. For MalignerDP, we adjusted parameters max-misses, miss-penalty, sd-rate, min-sd, and max-miss-rate and additionally filtered the results by alignment score. Though unpublished, for comparison we also include the proprietary RefAligner software from BioNano. For RefAligner we adjusted parameters FP, FN, sd, sf, A, and S. For KOHDISTA, we adjusted parameters chi-squared-cdf-thresh and binom-cdf-thresh. For Valouev, we adjusted score_thresh and t_score_thresh variables in the source. In Table 2.4 we report statistical and computational performance for each method.

OMBlast was configured with parameters meas=3000, minconf=0.09, minmatch=15 and the rest left at defaults. RefAligner was run with parameters FP=0.15, sd=0.6, sf=0.2, sr=0.0, se=0.0, A=15, S=22 and the rest left at deafults. MalignerDP was configured with parameters ref-max-misses=2, query-miss-penalty=3, query-max-miss-rate=0.5, min-sd=1500, and the rest left at defaults.

The software of Valouev *et al.* was run with default parameters except we reduced the maximum compound fragment length (their δ parameter) from 6 fragments to 3. We observed the software of Valouev *et al.* rarely included alignments containing more than two missed restriction sites in a compound fragment.

Chapter 3

Reducing memory by compression

In this section, we examine the compression aspect of the FM-Index. In these applications, the FM-Index is used in place of existing index structures for its space saving advantage.

3.1 VARI: Succinct Colored de Bruijn Graphs[13]

3.1.1 Introduction

In the 20 years since it was introduced to bioinformatics by Idury *et al.* [53], the *de Bruijn graph* has become a mainstay of modern genomics, essential to genome assembly [14,54,55]. The near ubiquity of de Bruijn graphs has led to a number of succinct representations, which aim to implement the graph in small space, while still supporting fast navigation operations. 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. A contig corresponds to a non-branching path through this graph. See [54] for a more thorough explanation of de Bruijn graphs and their use in assembly.

In 2012, Iqbal *et al.* [56] introduced the *colored de Bruijn graph*, a variant of the classical structure, which is aimed at "detecting and genotyping simple and complex genetic variants in an individual or population." The edge structure of the colored de Bruijn graph is the same as the classic structure, but now to each vertex ($(k-1)$-mer) and edge (k-mer) is associated a list of colors corresponding to the samples in which the vertex 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

[13] Martin D Muggli, Alexander Bowe, Noelle R Noyes, Paul S Morley, Keith E Belk, Robert Raymond, Travis Gagie, Simon J Puglisi, and Christina Boucher. Succinct colored de bruijn graphs. *Bioinformatics*, 2017.

this graph corresponds to an undirected cycle, and is shown to be indicative of biological variation by [56]. CORTEX, the implementation of [56], uses the colored de Bruijn graph to develop a method of assembling multiple genomes simultaneously, without losing track of the individuals from which $(k-1)$-mers (and k-mers) originated. This graph is derived from either multiple reference genomes, multiple samples, or a combination of both.

Variant information of an individual or population can be deduced from structure present in the colored de Bruijn graph and the colors of each k-mer. As implied by [56], the ultimate intended use of colored de Bruijn graphs is to apply it to massive, population-level sequence data that is now abundant due to next generation sequencing technology (NGS) and multiplexing. These technologies have enabled production of sequence data for large populations, which has led to ambitious sequencing initiatives that aim to study genetic variation for agriculturally and bio-medically important species. These initiatives include the *Genome 10K* project that aims to sequence the genomes of 10,000 vertebrate species [57], the *iK5* project [58], the 150 Tomato Genome ReSequencing project [59, 60], and the 1001 Arabidopsis project, a worldwide initiative to sequence cultivars of *Arabidopsis* [61]. Hence, the succinct colored de Bruijn graph is applicable in the context of these projects, in that it can assist in variation discovery within a species by analyzing all the data in these projects at once.

In addition to species-specific initiatives, scientific and regulatory agencies are showing increased interest in shotgun metagenomic sequences for public health purposes [62,63], specifically monitoring for antimicrobial resistance (AMR) [64, 65]. AMR is considered one of the top public health threats, with fears that the spread of AMR will lead to increased morbitiy and mortality for many bacterial illnesses [66, 67]. AMR occurs when bacteria express genetic elements that render them impervious to antibiotic treatments. Importantly, these genetic resistance elements can be exchanged between distantly-related bacteria via multiple genetic mechanisms, which makes AMR an inherently population-level phenomenon [68]. Shotgun metagenomic sequencing allows access to the entire microbial population in a sample (the "metagenome"), which is of immense value for tracking and understanding the evolution of resistance elements within and across diverse

bacteria [69]. This metagenomics approach to AMR surveillance has been applied in both human and agricultural settings [70, 71], generating hundreds of samples with terabytes of sequence data for relatively small studies. Given the large number of samples and large size of sequence data involved in these whole-genome and metagenomic projects, it is imperative that the colored de Bruijn graph can be stored and traversed in a space- and time-efficient manner.

Our Contribution

We develop an efficient data structure for storage and use of the colored de Bruijn graph. Compared to CORTEX, the implementation of [56], our new data structure dramatically reduces the amount of memory required to store and use the colored de Bruijn graph, with some penalty to runtime. We demonstrate this reduction in memory through a comprehensive set of experiments across the following three datasets: (1) four plant genomes, (2) 3,765 *Escherichia coli* assemblies, and (3) 87 sequenced metagenomic samples from commercial beef production facilities. We show our method, which we refer to as VARIMERGE (Finnish for color), has better peak memory usage on all these datasets. Our plant reference genomes dataset required 101 GB of RAM for CORTEX to represent while VARIMERGE required only 4 GB. And our largest two datasets contain too many k-mers and colors for CORTEX's data structure to represent in the 512 GB of RAM available on our bioinformatics servers. VARIMERGE is a novel generalization of the succinct data structure for classical de Bruijn graphs due to [72], which is based on the Burrows-Wheeler transform of the sequence reads, and thus, has independent theoretical importance.

In addition to demonstrating the memory and runtime of VARIMERGE, we validate its output using the *E.coli* reference genome and a simulated variant.

Related Work

As noted above, maintenance and navigation of the de Bruijn graph is a space and time bottleneck in genome assembly. Space-efficient representations of de Bruijn graphs have thus been heavily researched in recent years. One of the first approaches was introduced by [73] as part of the development of the ABySS assembler. Their method stores the graph as a distributed hash table

and thus requires 336 GB to store the graph corresponding to a set of reads from a human genome (>38x depth paired-end reads from Illumina Genome Analyzer II, HapMap: NA18507[14]).

[74] reduced space requirements by using a sparse bitvector (by [75]) to represent the k-mers (the edges), and used rank and select operations (to be described later) to traverse it. As a result, their representation took 32 GB for the same data set. Minia, by [76], uses a Bloom filter to store edges. They traverse the graph by generating all possible outgoing edges at each node and testing their membership in the Bloom filter. Using this approach, the graph was reduced to 5.7 GB on the same dataset. Contemporaneously, [72] developed a different succinct data structure based on the Burrows-Wheeler transform [77] that requires 2.5 GB. The data structure of [72] is combined with ideas from IDBA-UD [78] in a metagenomics assembler called MEGAHIT [79]. In practice MEGAHIT requires more memory than competing methods but produces significantly better assemblies. [80] implemented the de Bruijn graph using an FM-index and *minimizers*. Their method uses 1.5 GB on the same NA18507 data. [81] released the Bloom Filter Trie, which is another succinct data structure for the colored de Bruiin graph; however, we were unable to compare our method against it since it only supports the building and loading of a colored de Bruijn graph and does not contain operations to support our experiments. SplitMEM [82] is a related algorithm to create a colored de Bruijn graph from a set of suffix trees representing the other genomes. Lastly, Lin et al. [83] point out the similarity between the breakpoint graph, which is traditionally viewed as a data structure to detect breakpoints between genome rearrangements, and the colored de Bruijn graph.

Roadmap

In the next section, we describe our succinct colored de Bruijn graph data structure, generalizing the stucture for classic de Bruijn graphs presented by [72]. Section 3.1.3 then elucidates the practical performance of the new data structure, comparing it to CORTEX. Section 3.1.4 offers some concluding remarks.

[14]https://www.ncbi.nlm.nih.gov/sra/?term=SRA010896

3.1.2 Methods

Our data structure for colored de Bruijn graphs is based on the succinct representation of individual de Bruijn graphs introduced by [72]—which we refer to as the BOSS representation from the authors' initials—so we start by describing that representation. We note that BOSS is itself a generalization of FM-indexes [12] obtained by extending the Burrows-Wheeler transform (BWT) from strings to the multisets of edge-labels of de Bruijn graphs. We then give a general explanation of how we add colors, and finally give details of our implementation.

BOSS Representation

Consider the de Bruijn graph $G = (V, E)$ for a set of k-mers, with each k-mer $a_0 \cdots a_{k-1}$ representing a directed edge from the node labelled $a_0 \cdots a_{k-2}$ to the node labelled $a_1 \cdots a_{k-1}$, with the edge itself labelled a_{k-1}. Define the nodes' co-lexicographic order to be the lexicographic order of their reversed labels. Let F be the list of G's edges sorted co-lexicographically by their ending nodes, with ties broken co-lexicographically by their starting nodes (or, equivalently, by their k-mers' first characters). Let L be the list of G's edges sorted co-lexicographically by their starting nodes, with ties broken co-lexicographically by their ending nodes (or, equivalently, by their own labels). We refer to the ordering of L as *Vari-sorted*. If two edges e and e' have the same label, then they have the same relative order in both lists; otherwise, their relative order in F is the same as their labels' lexicographic order. Defining the edge-BWT (EBWT) of G to be the sequence of edge labels sorted according to the edges' order in L, so $\mathsf{label}(L[h]) = \mathsf{EBWT}(G)[h]$ for all h, this means that if e is in position p in L, then in F it is in position

$$|\{d : d \in E,\ \mathsf{label}(d) \prec \mathsf{label}(e)\}| + \mathsf{EBWT}(G).\mathsf{rank}_{\mathsf{label}(e)}(p) - 1\,,$$

where $\mathsf{EBWT}(G).\mathsf{rank}_{\mathsf{label}(e)}(p)$ is the number of times $\mathsf{label}(e)$ appears in $\mathsf{EBWT}(G)[1, p]$. It follows that if we have, first, an array S storing $|\{d : d \in E,\ \mathsf{label}(d) \prec c\}|$ for each character c and, second, 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.

Let B_F be the bitvector with a 1 marking the position in F of the last incoming edge of each node, and let B_L be the bitvector with a 1 marking the position in L of the last outgoing edge of each node. Given a character c and the co-lexicographic 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 labelled c. We can then 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[15]. Similarly, we can make similar queries about the incoming edges of a node v in an efficient manner using B_F. With the appropriate implementations of the data structures, we can store G in $(1 + o(1))|E|(\lg \sigma + 2)$ bits, where σ is the size of the alphabet (i.e., 4 for DNA), such that when given a character c and the co-lexicographic rank of a node v, in $\mathcal{O}(\log \log \sigma)$ time we can find the node reached from v by following the directed edge labelled c, if such an edge exists.

If we know the range $L[s..e]$ of k-mers whose starting nodes end with a pattern Y of length less than $(k-1)$, then we can compute the range $F[s'..e']$ of k-mers whose ending nodes end with Yc, for any character c, since

$$\begin{aligned} s' &= |\{d : d \in E,\ \mathsf{label}(d) \prec c\}| + \mathsf{EBWT}(G).\mathsf{rank}_c(s-1) \\ e' &= |\{d : d \in E,\ \mathsf{label}(d) \prec c\}| + \mathsf{EBWT}(G).\mathsf{rank}_c(e) - 1\,. \end{aligned}$$

It follows that, given a node v's label, we can find the interval in L containing v's outgoing edges in $\mathcal{O}(k \log \log \sigma)$ time, provided there is a directed path to v (not necessarily simple) of length at least $k-1$. In general there is no way, however, to use $\mathsf{EBWT}(G)$, B_F and B_L alone to recover the labels of nodes with no incoming edges.

To prevent information being lost and to be able to support searching for any node given its label, Bowe et al. add extra nodes and edges to the graph, such that there is a directed path of length at least $k-1$ to each original node. Each new node's label is a $(k-1)$-mer that is prefixed

[15] In practice, we incorporate the bits of B_F as flags on $\mathsf{EBWT}(G)$ and use them to obtain the colex order of v but omit the discussion here for simplicity. We refer the reader to Bowe *et al.* [72] for a full discussion of this aspect and the supplement for our handling here.

by one or more copies of a special symbol \$ not in the alphabet and lexicographically strictly less than all others. Notice that, when new nodes are added, the node labelled $\$^{k-1}$ is always first in co-lexicographic order and has no incoming edges. Bowe et al. also attach an extra outgoing edge labelled \$, that leads nowhere, to each node with no original outgoing edge. The edge-BWT and bitvectors for this augmented graph are, together, the BOSS representation of G.

Adding Color

We cannot represent the colored de Bruijn graph for a multiset $\mathcal{G} = \{G_1, \ldots, G_t\}$ of individual de Bruijn graphs satisfactorily by simply representing each individual graph separately, for two reasons: first, the memory requirements would quickly become impractical and, second, we should be able to answer efficiently queries such as "which individual graphs contain this edge?" Therefore, we set G to be the union of the individual graphs and build the BOSS representation only for G. As long as most of the k-mers are common to most of the individual graphs, the memory needed to store G is comparable to that need to store an individual graph.

To indicate which edges of G are in which individual graphs, we build and store a two-dimensional binary array C in which $C[i, j]$ 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). (Recall from the description above of BOSS that we consider the edges in G to be sorted lexicographically by the reversed labels of their starting nodes, with ties broken lexicographically by their own single-character labels.) If the individual graphs are sufficiently similar, then we can compress C effectively and store it in such a way that we can still access its individual bits quickly and support fast rank and select queries on the rows. (A select query on the ith row takes an argument r and returns the index j of the rth individual graph that contains the ith edge in G.) In the next subsection we give details of some relatively simple compression strategies that support fast access, rank and select. With these data structures, we can navigate efficiently in any of the individual graphs and switch between them. For example, we can efficiently check whether an edge has a particular color (with an access), count the number of colors it has (with a rank query) or list them (with repeated select queries). We have not yet considered more sophisticated compression schemes that could still offer

fast queries while taking advantage of, e.g., correlations among the variations or grouping of the individual graphs by subpopulation.

Figure 3.1 shows an example of how we represent a colored de Bruijn graph consisting of two individual de Bruijn graphs. Suppose we are at node ACG in the graph, which is the co-lexicographically eighth node. Since the eighth 1 in B_L is $B_L[10]$ and it is preceded by two 0s, we see that ACG's outgoing edges' labels are in $\mathsf{EBWT}[8..10]$, so they are A, C and T. Suppose we want to follow the outgoing edge e labelled C. We see from $C[9, 0..1]$ (i.e., the tenth column in C^{T}) that e appears in the second individual graph but not the first one (i.e., it is blue but not red). There are four edges labelled A in the graph and three Cs in $\mathsf{EBWT}(G)[0..9]$, so e is $F[6]$. (Since edges labelled $ have only one end, they are not included in L or F.) From counting the 1s in $B_F[0..6]$, we see that e arrives at the fifth node in co-lexicographic order that has incoming edges. Since the first node, $$$, has no incoming edges, that means e arrives at the sixth node in co-lexicographic order, CGC.

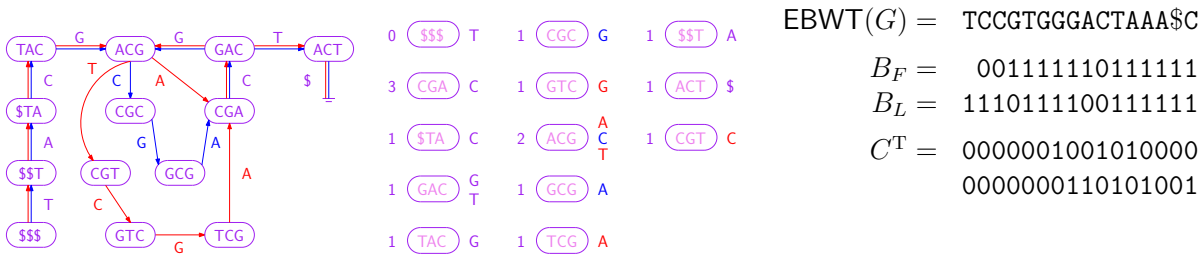


Figure 3.1: Left: A colored de Bruijn graph consisting of two individual graphs, whose edges are shown in red and blue. (We can consider all nodes to be present in both graphs, so they are shown in purple.) **Center:** The nodes sorted into co-lexicographic order, with each node's number of incoming edges shown on its left and the labels of its outgoing edges shown on its right. The edge labels are shown in red or blue if the edges occur only in the respective graph, or purple if they occur in both. **Right:** Our representation of the colored de Bruijn graph: the edge-BWT and bitvectors for the BOSS representation for the union of the individual graphs, and the binary array C (shown transposed) whose bits indicate which edges are present in which individual graphs.

Data Structure

The arsenal of component tools available to succinct data structures designers has grown considerably in recent years [84], with many methods now implemented in libraries. We chose to make heavy use of the succinct data structures library (SDSL)[16] in our implementation.

$\mathsf{EBWT}(G)$, the sequence of edge labels, is encoded in a wavelet tree, which allows us to perform fast rank queries, essential to all our graph navigations. The bitvectors of the wavelet tree and the B bitvector are stored in the Raman-Raman-Rao (RRR) encoding [85]. The rows of the color matrix, C, are concatenated (i.e. C is stored in row-major order) and this single long bit string is then compressed. It is either stored with RRR encoding, or alternately Elias-Fano encoding [75, 86, 87] which supports online construction. Online construction is important for datasets where C is too large to fit in memory in uncompressed form, such as our metagenomic sample dataset. These encodings reduce the size of C considerably because we expect rows to be very sparse and both encodings exploit this sparseness.

Construction

In order to convert the input data to the format required by BOSS (that is, in correct sorted order, including dummy edges and bit vectors), we use the following process. We take care to ensure only subsets of data are needed in RAM at any one time during construction.

Our construction algorithm takes as input the set of (k-mer, color-set) pairs present in the input sets of reads, or alternately, k-mer counts for each color which we convert to the former ourselves. Here, color-set is a bit set indicating which samples the k-mer occurs in. We provide the option to use the CORTEX frontend to generate the (k-mer, color-set). Unfortunately, this also limits the datasets to those that would run through CORTEX. To overcome this, we provide the option to use a list of KMC2 [88] sorted k-mer counts as input. With this option, the k-mers from each k-mer count file in native KMC2 binary format are streamed through a priority queue to produce the union of all k-mer sets; initially one k-mer from each file is tagged with which file it originated from, and

[16] https://github.com/simongog/sdsl-lite

the (k-mer, file ID) pair is added to the queue. The priority queue ensures the lexicographically smallest k-mer instances across all files can be popped off the queue consecutively. All of the k-mer count files contributing a particular k-mer value have their corresponding color recorded as '1' bits in the bit set for that k-mer. Both the k-mer and the bit set are then appended to vectors which optionally are allocated in external memory using the STXXL[17] library. As each k-mer is popped off the queue, another k-mer is added to the queue to take the old k-mer's place (i.e. using the file identified by the popped k-mer's tag). This process continues until all files are read in their entirety. By both streaming data from the source files and streaming it to the external vectors, only a small amount of the data need exist in memory at a time; the priority queue will only contain the number of samples and only one row of the color matrix needs to exist in memory before being written out to disk.

After constructing the initial union set of k-mers and their corresponding color rows, BOSS construction mostly continues as originally described by Bowe *et al.*. The changes from the original construction algorithm are that most of the data optionally resides in external memory and the rows of the color matrix are permuted with their corresponding k-mers as they are sorted. For each of the k-mers we generate the reverse complement (giving it the same color-set as its twin). Then, for each k-mer (including the reverse complements), we sort the (k-mer, color-set) pairs by the first $k-1$ symbols (the source node of the edge) to give the F table (from here, the colors are moved around with rows of F, but otherwise ignored until the final stage). Independently, we sort the k-mers (without the color-sets) by the last $k-1$ symbols (the destination node of the edge) to give the L table.

With F and L tables computed, we calculate the set difference $F - L$ (comparing only the $(k-1)$-length prefixes and suffixes respectively), which tells us which nodes require incoming dummy edges. Each such node is then shifted and prepended with \$ signs to create the required incoming dummy edges ($k-1$ each). These incoming dummy edges are then sorted by the first $k-1$ symbols. Let this table of sorted dummy edges be D. Note that the set difference $L - F$ will

[17] http://http://stxxl.sourceforge.net/

give the nodes requiring outgoing dummy edges, but these do not require sorting, and so we can calculate it as is needed in the final stage.

Finally, we perform a three-way merge (by first $k-1$ symbols) D with F, and $L-F$ (calculated on the fly). For each resulting edge, we keep track of runs of equal $k-1$ length prefixes, and $k-2$ length suffixes of the source node, which allows us to calculate the B_F and B_L bit vectors, respectively. Next, we write the bit vectors, symbols from last column, and count of the second to last column to a packed file on disk, and the colors to a separate file. The color file is then either buffered in RAM and RRR encoded or optionally streamed from disk and then Elias-Fano encoded online (i.e. only the compressed version is ever resident). The time bottleneck in the above process is clearly in sorting the D and F tables, which are of the same size, and are made up of elements of size $\mathcal{O}(k)$. Thus, overall, construction of the data structure takes $\mathcal{O}(k(|F|\log|F|))$ time.

Traversal

We implemented two traversal methods based on those of CORTEX with a modification in light of our intention to apply VARIMERGE to metagenomic reads looking for AMR gene presence.

The first, *bubble calling*, is a simple algorithm to detect sequence variation in genomic data. It consists of iterating over a set of k-mers in order to find places where bubbles start and terminate. When combined with the k-mer color (in a colored de Bruijn graph), this enables identification of places where genomic sequences diverge from one another. The differing region of the two sequences will form the two arms of a bubble, each colored with only one of the two sequence's colors. A bubble is identified when a vertex has two outgoing edges. Each edge is followed in turn to navigate a non-branching path until reaching a vertex with two incoming edges. If the terminating vertex is the same for both paths, we call this a bubble. Colors for the bubbles are determined by looking at the color assignment of the corresponding (k)-mers. Our implementation in VARIMERGE closely follows the pseudocode given by [56].

CORTEX's traversal algorithms were designed for single isolates. For the beef safety experiments, which use metagenomic samples, we implemented a traversal inspired by CORTEX's *path divergence* algorithm. In the original CORTEX path divergence algorithm, bubbles are identified

where a user-supplied reference sequence prescribes a walk through a (possibly tangled) sections of the graph in one arm of a bubble while the alternative arm must be branch free. This branch free requirement on the second arm could be a problem for metagenomic data. Due to the presence of tangle inducing homologous genomes and risk of inferring erroneous, chimeric sequences (which comprise reads from a mix of genomes in the sample), variant detection in metagenomic data is more complex. In the absence of a simple metagenomic-aware traversal algorithm, we implemented a variation of the path divergence algorithm which addresses a simpler problem, primarily for the purpose of measuring performance. This algorithm uses a reference guided approach and allows us to measure the memory footprint at traversal time as well as the time savings of not traversing the entire dataset. For this purpose, we focus specifically on the presence of AMR genes (our reference sequence) rather than variants of those genes; in our derived algorithm we ignore sample path segments leading away from and returning to the AMR gene path. This avoids some of the problems with tangles, incomplete coverage, or read errors. Thus as we traverse the gene path, we simply count the number of samples in each sample group that color the current edge. We note that keeping C in row major order allows us to compute this count in constant time as the difference between two rank queries.

3.1.3 Results

We evaluated VARIMERGE performance on three different datasets, described below. For this evaluation, we compare peak memory, which was measured as the maximum resident set size, and CPU core time, measured as the user+system process time as our metrics. In addition to evaluating performance, we also validated VARIMERGE by the ability to correctly call bubbles known to be present in a simulated dataset.

Our software supports a variety of options. It can consume k-mer counts from either Cortex's binary files or KMC2. For all experiments, we use the KMC2 flow because using Cortex as a front end limits designs to only those that would fit in memory with Cortex. Next, our software can compress the color matrix using either RRR or Elias-Fano encodings. The SDSL-light implementation

allows the color matrix to be compressed in an on-line fashion only using the Elias-Fano encoding. This allows us to process larger designs, as the uncompressed matrix need never fit in RAM, and thus we use this option for all experiments. Finally, STXXL (which holds temporary vectors during data structure construction) allows using internal or external memory. Again, we used the more scalable external memory option for all experiments. All experiments were performed on a machine with AMD Opteron model 6378 processors, having 512 GB of RAM and 64 cores.

Datasets

The three different datasets were chosen in order to test and evaluate the performance of VARIMERGE on a variety of diverse yet realistic data types that are likely to be used as input into VARIMERGE. For the first two datasets which comprise single isolates, we use preassembled genomes. Assembly serves to try correct sequencing errors which could otherwise falsely be detected as variants. To this end, CORTEX includes its own optional data cleaning operations. However, by using instead the output of third party assembly software we can compare the colored de Bruijn graph performance on identical graphs. Characteristics about these datasets are provided in Table 3.1.

Table 3.1: Characteristics of our datasets. The *E. coli* dataset represents 3,765 strains and hence only summary statistics for size and GC content are given. Accession numbers for this dataset as well as download procedure can be found in assembly_summary.txt as discussed in the main text.

Name	Accession Numbers	Aprox. Size	GC Content
Plant Species	Rice (NC_008394 to NC_008405)	430 Mbp	43.42%
	Tomato (NC_015438 to NC_015449)	950 Mbp	43.42%
	Corn (NC_024459 to NC_024468)	2.07 Gbp	35.70%
	Arabidopsis (NC_003070 to NC_003076)	135 Mbp	47.4%
E. coli strains	N/A	avg=5.1 Mbp min=2.9 Mbp max= 7.7 Mbp	50.5%
Beef safety	PRJNA292471	N/A	44.3%

Table 3.2: Data structure construction performance measurements. CPU time is user plus system time as reported by '/bin/time'. (Internal) memory is reported in megabytes and is the maximum resident set size. KMC2 includes both counting and sorting k-mers. VARIMERGE-dBG forms the k-mer union and builds the succinct de Bruijn graph. VARIMERGE-C compresses the color matrix.

	CORTEX		KMC2		VARIMERGE-dBG			VARIMERGE-C	
Dataset	**CPU time**	**Mem.**	**CPU time**	**Mem.**	**CPU time**	**Int. Mem.**	**Ext. Mem.**	**CPU time**	**Mem.**
Plants	2h 25m 27s	109,579	19m 50s	4,335	1h 34m 37s	5,388	156,504	3m 09s	3,528
E. coli (k=32)	N/A	N/A	3h 15m 40s	104	9h 30m 11s	126,777	319,328	53m 54s	42,043
E. coli (k=48)	N/A	N/A	4h 35m 29s	149	10h 47m 46s	128,077	427,460	1h 02m 07s	42,100
E. coli (k=64)	N/A	N/A	5h 05m 27s	189	11h 21m 08s	127,523	522,576	1h 09m 07s	42,134
Beef safety	N/A	N/A	34h 04m 46s	11,688	82h 42m 48s	109,091	4,378,840	6h 44m 12s	217,705

Our first performance dataset comprises reference genomes for four different plant species: *Oryza sativa Japonica* (rice)[18] [89], *Solanum lycopersicum* (tomato)[19] [59, 60], *Zea mays* (corn)[20] [90], and *Arabidopsis thaliana* (Arabidopsis)[21] [91]. This represents a sufficiently large dataset for comparing the performance of VARIMERGE with CORTEX.

Our second performance dataset consists of the set of all 3,765 NCBI GenBank assemblies[22][23] having the organism_name field equal to "Escherichia coli" as of March 22, 2016. To evaluate the effects of varying k-mer size, we ran this dataset with $k = 32, 48, 64$. The union of all assemblies contains 158,501,209 k-mers for k=32, 205,938,139 k-mers for k=48, and 251,764,413 k-mers for k=64. The minimum, maximum, and average assembly lengths are 2,911,360 bp, 7,687,202 bp, and 5,156,744 bp, respectively.

Our third performance dataset consists of 87 metagenomic samples[24] taken at various timepoints during the beef production process from eight pens of cattle in two beef production facilities by [70]. Sequentially, these timepoints were feedlot arrival, feedlot exit, slaughter transport truck, slaughter holding, and slaughter trimmings and sponges. Sample reads were preprocessed using trimmomatic v0.36 by Bolger *et al.* [92]. Although further assembly or error correction would have been possible, it would reduce the biological variation which may be useful for some queries. Furthermore, building the data structure on uncorrected data better stresses our representation method. Samples were then arranged into groups based on the sample timepoints. The original study used these samples to demonstrate the advantages of shotgun metagenomic sequencing in tracking the evolution of antimicrobial resistance longitudinally within a complex environment

[18] http://rice.plantbiology.msu.edu/annotation_pseudo_current.shtml

[19] ftp://ftp.solgenomics.net/tomato_genome/assembly/build_2.50/SL2.50ch00.fa.tar.gz

[20] ftp://ftp.ncbi.nlm.nih.gov/genomes/all/GCF/000/005/005/GCF_000005005.1_B73_RefGen_v3/GCF_000005005.1_B73_RefGen_v3_genomic.fna.gz

[21] ftp://ftp.ensemblgenomes.org/pub/plants/release-34/fasta/arabidopsis_thaliana/dna/

[22] ftp://ftp.ncbi.nlm.nih.gov/genomes/genbank/bacteria/assembly_summary.txt

[23] https://www.ncbi.nlm.nih.gov/genome/doc/ftpfaq/

[24] https://www.ncbi.nlm.nih.gov/bioproject/292471

such as beef production; the results suggested that selective pressures occurred within the feedlot, but that slaughter safety measures drastically reduced both bacterial and AMR levels. In addition to the metagenomic samples, we included 4,062 AMR genes from the previously mentioned gene databases[25]. 23 genes in the databases containing IUPAC codes other than the four bases were filtered out as KMC2 and the succinct de Bruijn graph were configured with a four symbol alphabet. Because we have the reference to guide the traversal, all AMR genes were combined into a single color. By combining AMR genes, the uncompressed color matrix that exists on disk during sorting and as intermediate file is much smaller (still occupying 1.2 TB), thus accelerating the permutation during construction and reducing the external memory and disk space requirements. The union of all samples and genes contains 40,995,794,366 32-mers and the GC content is 44.3%. While our server has enough RAM to represent a dataset with twice the memory footprint, this dataset nearly exhuasted the approximately 10 TB of disk space available when intermediate files were preserved. Thus this dataset is on the order of the upper limit for VARIMERGE in practice.

Finally, for validation purposes, we generated a dataset[26] comprising two genomes: (1) *E. coli* K-12 substraing MG 1655 reference genome, and (2) a copy of the reference genome to which we applied various simulated mutations. We simulated mutations by choosing 100 random loci and either inserting, deleting, or replacing a region of random length ranging from 200-500 bp. For each mutation locus, we record the flanking regions and the two variants (original reference and simulated) as a ground truth bubble.

Time and Memory Usage

To measure VARIMERGE's resource use and compare with CORTEX by Iqbal *et al.* [56] where possible, we constructed the colored de Bruijn graph for the plant dataset, the *E. coli* assembly dataset and the beef safety dataset. Construction time and memory is detailed in Table 3.2. We performed *bubble calling* on the first two and recorded peak memory usage and runtime. Direct

[25]https://meg.colostate.edu/MEGaRes/

[26]https://github.com/cosmo-team/cosmo/tree/VARI/experiments/ecoli_validation

resource comparison with CORTEX was only possible on the smallest dataset, as the largest two have too many k-mers and colors to fit in memory on our machine with CORTEX. Based on the data structure defined in CORTEX's source as well as the supplementry information provided by Iqbal *et al.*, it would have required more than 3 TB of RAM and more than 18 TB of RAM for its hash table entries alone, respectively.

In order to test query performance characteristics, various experiments were performed on all three performance datasets described in the previous subsection. Datasets varied in the number of k-mers in the graph from 158 million to over 40 billion, the number of colors, from 4 to 3,765, and degree of homology from disperate plants to the single *E. coli* species. This diversity shows the space savings achievable when the population is largely homologous, as is the case with the *E. coli* dataset, where the graph component is relatively small, in contrast to the plant dataset, where the graph component is relatively large. As can be seen in Table 3.3, where directly comparable, VARIMERGE used an order of magnitude less than the peak memory that CORTEX required but required greater running time. This memory and time trade-off is important in larger population level data. This is highlighted by our largest two datasets which could not be run with CORTEX. Hence, lowering the memory usage in exchange for higher running time deserves merit in contexts where there is data from large populations.

Table 3.3: Comparison between the peak memory and time usage required to store all the k-mers and run bubble calling on the data in CORTEX and VARIMERGE. The peak memory is given in megabytes (MB) or gigabytes (GB). The running time is reported in seconds (s), minutes (m), and hours (h). The succinct de Bruijn graph and compressed color matrix components of the memory footprint are listed in parenthesis as sdBG and sC, respectively.

			CORTEX		VARIMERGE	
Dataset	No. of k-mers	Colors	Memory	Time	Memory	Time
Plants (k=32)	1,709,427,823	4	100.93 GB	2h 18m	3.53 GB (sdBG=0.89 GB, sC=1.95 GB)	32h 39m
E. coli (k=32)	158,501,209	3,765	N/A	N/A	42.17 GB (sdBG=0.09 GB, sC=38.35 GB)	3h 57m
E. coli (k=48)	205,938,139	3,765	N/A	N/A	42.26 GB (sdBG=0.11 GB, sC=38.42 GB)	4h 38m
E. coli (k=64)	251,764,413	3,765	N/A	N/A	42.32 GB (sdBG=0.13 GB, sC=38.45 GB)	5h 28m
Beef safety (k=32)	40,995,794,366	88	N/A	N/A	245.54 GB (sdBG=27.08 GB, sC=200.34 GB)	N/A

Validation on Simulated *E. coli*

We ran the implementations of bubble calling from both VARIMERGE and CORTEX, using k=32 on the simulated *E. coli* dataset. Both tools reported the same set of 223 bubbles, 55 of which were in the ground truth set. This ensures our software faithfully implements the original data handling capabilities of CORTEX. For biological implications of colored de Bruijn graph variant calls and in particular with parameter choices such as k see Iqbal *et al.* [56].

Observations on Beef Safety Dataset

While the beef safety dataset was primarily used for measuring the scalability of VARIMERGE and to determine if representing a dataset of this type and size was possible, we used VARIMERGE to additionally make observations about the presence of AMR genes in the beef production dataset. As previously described, during our path divergence derived algorithm, we compute a count of how many k-mers in each AMR gene are found across all samples within a sample group. This algorithm need only traverse the AMR genes, so despite the size of the overall dataset, it only took 20 minutes to load and access the necessary parts of the data structure. In contrast, if bubble calling were to run at the same rate for this dataset as for the *E. coli* assembly dataset, it would take 3,001 hours to complete, thus suggesting value in a targeted inquiry approach on datasets of this size.

Since longer genes have more k-mers, the counts are likely to be larger, as are those from larger sample groups. To make these counts comparable, we normalize by both gene length and sample group size. We can then examine the number of genes having a disproportionately large ($>$ 3 std. dev. above mean) shared k-mer count for each gene and sample group combination. The number of such genes with disproportionately large normalized counts in each sample group were: feedlot arrival - 304, feedlot exit - 93, transport truck - 230, slaughter holding - 16, and slaughter trimmings and sponges - 0. This observation supports the conclusion of [70], namely, that antimicrobial interventions during slaughter were effective in reducing AMR gene presence in the trimmings and sponge samples, which represent the finished beef products just before they are shipped to retail outlets for human consumption.

3.1.4 Concluding Remarks

We presented VARIMERGE, which is an implementation of a succinct colored de Bruijn graph that significantly reduces the amount of memory required to store and use the colored de Bruijn graph. In addition to the memory savings, we validated our approach using *E coli.* Moreover, we introduced the use of colored de Bruijn graph for accurately identifying the presence of AMR genes within metagenomic samples, which is an important advance as public health officials increasingly move towards a metagenomic sequence-based approach for surveillance and identification of resistant bacteria [64, 65, 67]. Possible nontrivial extensions to our work include (1) (1) using multi-threading to speed up the bubble calling, (2) compressing the color array C more effectively by taking advantage of correlations among the variations, and (3) applying more sophisticated approaches to metagenomic data.

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

In this section, we explore how the compressed nature of the VARI data structure can be used as its own intermediate representation in the construction of much larger succinct colored de Bruijn graphs.

3.2.1 Introduction

In recent years, there has been an initiative to move toward using whole genome sequencing to accurately identify and track foodborne pathogens (e.g. antibiotic resistant bacteria) [93]. 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. Currently, the GenomeTrakr effort includes over 50,000 samples, spanning several species available through this initiative—a number that continues to rise as datasets are continually added [94]. Unfortunately, methods to analyze this and other datasets are limited due to their size. Existing methods for using this WGS data frequently focus on a method known as Multi Locus Sequence Typing (MLST) [95], which

aligns reads to genes from a reference genome. These alignments identify sets of alleles and thus, are limited to capturing genetic variations that are both shorter than the length of reads and only those variations that align to a reference genome or gene set [96]. Thus, variations that are longer than read length or that exist in the population but not the reference genome go undetected.

Given the limitations of existing methods, we would like to apply advanced methods for identifying variants—such as Cortex [56] and VARI [97]—which are able to detect complex variants without a reference. These methods use a modification of the de Bruijn graph that is referred to as the colored de Bruijn graph. We define the *de Bruijn graph* constructively as follows: a directed edge is created for every unique k-length subsequence (*k-mer*) in the data and an origin and destination vertex are labeled with the prefix and suffix of that k-mer, and after all edges have been created and labeled, the vertices that have the same label are glued into a single vertex. We create a *colored de Bruijn graph* by adding a set of colors (or labels) to each edge (and/or vertex) indicating which sample(s) contain the respective k-mer (and/or $(k-1)$-mer). Iqbal et al. [56] were the first to present the concept of the colored de Bruijn graph and demonstrate how it can be traversed to identify genetic variation between samples.

A bottleneck in applying Cortex to large datasets is the amount of memory required to build and store the colored de Bruijn graph. VARI [97] and Rainbowfish [98] sought to overcome this limitation by improving the storage efficiency of the graph. However, even though these methods store the colored de Bruijn graph in a memory-efficient manner, they are still unable to scale in a manner that is necessary for massive datasets such as GenomeTrakr. The limiting factor for these methods lies in their construction; Both VARI [97] and Rainbowfish [98] must manipulate the (uncompressed) data in external memory in order to build the graph in a memory-efficient manner; making external memory use the bottleneck. Moreover, this increases the construction time of the graph as external memory use is slower than that of RAM. Thus, one way to improve the scalability and enable researchers to construct the colored de Bruijn graph on massive datasets 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 graphs until a single

graph remains. While partitioning the data and building small graphs is possible, there exists no method to succinctly merge (colored) de Bruijn graphs.

Our contributions.

Thus, we present VARIMERGE that enables construction of massive colored de Bruijn graph through a process of partitioning the data into smaller sets, building the colored de Bruijn graph in a memory-efficient manner for each parition, and merging colored de Bruijn graphs. Each of the colored de Bruijn graphs is stored using the FM-index in the same manner as VARI [97]. We review this representation in Section 3 of the paper [72]. 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.

By using VARIMERGE, we build a colored de Bruijn graph for 16,000 strains of Salmonella that were collected and housed at NCBI as part of the GenomeTrakr database. This represents the first and only large-scale assembly based analysis of the GenomeTrakr data [95] and to the best of our knowledge, the largest dataset for which the (colored) de Bruijn graph has been constructed. The most recent unrelated large-scale construction is due to Holley et al. [99], which presents a de Bruijn graph construction for 473 clinical isolates of *Pseudomonas aeruginosa* (NCBI BioProject PRJEB5438). Our GenomeTrakr dataset is over 30 times this size of this latter one. The construction of this colored de Bruijn graph required a total of 254 G of RAM, 2.34 TB of external memory, and less than 72 hours of CPU time. VARI and Rainbowfish could–at least, in theory–construct the graph for this large of a dataset but would require over 10 TB of disk space and more computing time. Moreover, our results that compare VARIMERGE with Bloom Filter Trie demonstrate that it would require significantly more memory to construct and store and the colored de Bruijn graph on this dataset.

Therefore, VARIMERGE is superior with respect to the memory and disk usage. It is more memory-efficient that Bloom Filter Trie. Thus, it has the memory-efficiency of competing succinct

representations (e.g., VARI and Rainbowfish) but it removes the extensive disk constraints these method have, making VARIMERGE practical for massive datasets.

3.2.2 Related Work

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 [73]. In 2011, Conway and Bromage [74] reduced these space requirements by using a sparse bitvector (by Okanohara and Sadakane [75]) to represent the k-mers, and used rank and select operations (to be described shortly) to traverse it. Minia [76] 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 [72] developed a succinct data structure based on the Burrows-Wheeler transform (BWT) [10]. This data structure is discussed in more detail in the next section. This data structure of Bowe et al. [72] is combined with ideas from IDBA-UD [78] in a metagenomics assembler called MEGAHIT [79]. Chikhi et al. [80] 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. [99] introduced the Bloom Filter Trie, which is another succinct data structure for the colored de Bruijn graph. SplitMEM [82] is a related algorithm that creates a colored de Bruijn graph from a set of suffix trees representing the other genomes. Lastly, VARI [97] and Rainbowfish [98] 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 [82] 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. [100] improved on this method with two alternative construction methods, using the compressed suffix tree and using BWT. TwoPaCo [101] 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. [99] 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 [98] 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 than VARI, it uses VARI as a preprocessing step in its construction, so it is still limited to VARI's construction capacity.

Lastly, two other approaches are worthy of note because they merge the BWT of a set of strings. BWT-Merge by Sirén [102] 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, Holt and McMillan developed MSBWT [103] which merges the BWTs of multiple strings in a method similar to our own except applied to strings instead of graphs.

3.2.3 Preliminaries

As previously mentioned, in 2017 Muggli et al. [97] presented VARI, which is a representation of the colored de Bruijn graph using BWT. Our proposed method, VARIMERGE, efficiently merges

de Bruijn graphs that are represented in this manner. Therefore, we first define some basic notation and definitions concerning BWT, then we show how the de Bruijn graph can be stored using BWT, and finally, we show how the *colored* de Bruijn graph can be stored succinctly. We refer the reader to the full paper by Muggli et al. [97] for a more detailed discussion of the representation.

Storage of the Colored de Bruijn Graph

We use the same representation as in VARI. Given this representation we can traverse the graph and recover incoming and outgoing edges. Next, we demonstrate how the labels (k-mers) can be recovered using this data structure.

Label recovery.

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* (or recovered) from this representation. As previously mentioned, we can traverse the graph in a forward or reverse manner and recover incoming and outgoing edges of a given node v. Given this efficient traversal, we can recover the label of v by traversing the graph in a backward direction starting from v; given the label of v is a $(k-1)$-mer we traverse backward $k-1$ times. Therefore, we must 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.

3.2.4 Method

In this section, we give an overview of our approach for merging colored de Bruijn graphs that are stored using the VARI data structure, and in the next section, give the merge algorithm

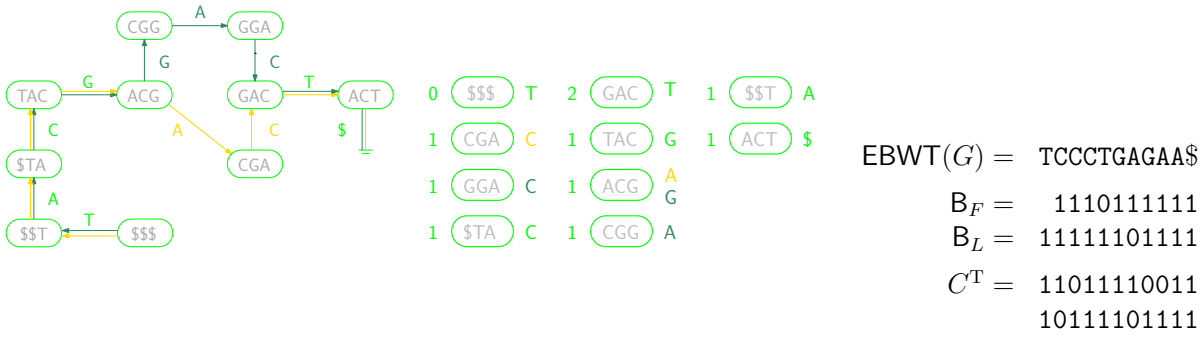


Figure 3.2: Left and center: A colored de Bruijn graph consisting of two individual graphs, whose edges are shown in yellow and green. All nodes to be present in both graphs are shown in lime. **Right:** The VARI representation of the colored de Bruijn graph: the edge-BWT and bitvectors for the union of the individual graphs, and the binary array C (shown transposed) whose bits indicate which edges are present in which individual graphs.

in explicit detail. In both sections, 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 $\mathsf{EBWT}(G)_1$, B_{L1}, B_{F1} and C_1 and $\mathsf{EBWT}(G)_2$, B_{L2}, B_{F2} and 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}, the array B_{FM}, and the color matrix C_M.

A Naive Merge Algorithm

We begin by describing a naive merge procedure to motivate the use of the succinct merge algorithm. We recall from Section 3.2.3 that VARI does not store the edge labels (k-mers) of G– rather, they have to be computed from the succinct representation. We denote the edge labels for G_1, G_2, and G_M as L_1, L_2, and L_M, respectively. For example, if we want to reconstruct the k-mer AGAGAGTTA contained in G_1 which is 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 G_1 and G_2 by reconstructing L_1 and L_2, merging them into L_M and computing the succinct representation of L_M, i.e., $\mathsf{EBWT}(G)_M$. We

note that this algorithm requires explicitly building L_1, L_2 and L_M and thus, has a significant memory footprint. See Figure 3.4 for pseudocode of this algorithm.

The Succinct Merge Algorithm

We are now ready to describe the merge algorithm used as a component of VARIMERGE. The trick of the merge algorithm is to build the succinct data structure for G_M without constructing L_1 and L_2, which will in turn reduce memory costs enormously.

Intuitive Explanation of Succinct Merging.

Before we give a detailed explanation of our algorithm we take a step back–abstract away the complexities of the succinct de Bruijn graph–and consider the simpler 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 draw the reader's attention to the fact our succinct colored de Bruijn graph representation is a space-efficient representation of the list of sorted k-mers (and $(k-1)$-mers). Thus, we can apply this general algorithm but alter it in in the following ways: 1) the nested grouping of the strings is rather a flat partitioning of the two lists into intervals, which is updated each time a character is processed, and 2) the actual merging is reserved for the end once all characters have been processed and their needed information accumulated into set of partitions of each list.

Overview of the Algorithm.

Now we return to the problem of merging succinct colored de Bruijn graphs. We refer to $\mathsf{EBWT}(G)_M$ and C_M as the *primary components* of the data structure and B_{FM} and B_{LM} as *secondary components*. We describe how to merge the primary components, and leave the details of how to merge the secondary components to the supplement. The algorithm consists of two steps: (1) a planning step which plans the merge, and (2) a final execution step which executes the planned merge. In the planning step, we output a list of non-overlapping intervals for L_1 and one for L_2. We refer to these lists as a *merge plan*, which is then used to execute the merge. There are k iterations of the planning algorithm (where k corresponds to the k-mer value). At each iteration of the algorithm a single character of the edge labels (k-mers) is processed, and the merge plan is revised. After k iterations, we execute the merge plan.

The Planning Step.

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. 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 the next character[27] of each edge label in L_1 and L_2. An overview of the The Planning Step is given in Figure 3.5. Thus, in order to fully describe the planning stage, we define (1) how the characters of the edge labels are computed (e.g. *GetCol*(i, G_1) in Figure 3.8), and (2) how P_1 and P_2 are revised based on these characters (e.g. *RefinePlan*$(P_1, P_2, Col_1, Col_2, i)$ in Figure 3.8).

Computing the next character 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 next characters of L_1 and L_2 using two temporary character vectors Col_1^i and Col_2^i,

[27]We recall that FM-index stores the last character of each edge label and we do not have access to L_1 and L_2. Therefore, we are processing the characters of L_1 and L_2 from right to left. Thus, the "next" character is the preceding character of an edge label.

which are of length $|\mathsf{L}_1|$ and $|\mathsf{L}_2|$, respectively. Conceptually, we 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 $Col_2^i[j] = \mathsf{L}_2[j][k]$. Since we do not explicitly build or store L_1 and L_2, we must compute Col_1^i and Col_2^i. We leave the details of computing Col_1^i and Col_2^i based on the succinct de Bruijn graph to the supplement (see Subsection 1).

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 $n = 1, .., |P_1|$, 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_1^i and Col_2^i—e.g., for each run of `A`, `C`, `G`, `T` or `$` (See Figure 3.6). Next, we formally define this computation.

Thus, we partition 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 a character c in $\{\$, \mathtt{A}, \mathtt{C}, \mathtt{G}, \mathtt{T}\}$, 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)[28]. 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 Figure 3.9 in the supplement for the pseudocode. We crafted the method above to maintain the property described in the following observation.

[28]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

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.

The Execution 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 abbreviated 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 Figure 3.9 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 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.

Computational Complexity

The following theorem demonstrates the efficiency of our approach.

Theorem 1. *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 our merge algorithm constructs the merged de Bruijn graph G_M in $\mathcal{O}(m \cdot \max(k, t))$-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 that 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 $\mathcal{O}(m \max(k, t))$-time. Hence, if $k \leq t$ the merge algorithm will take $\mathcal{O}(mk)$-time; otherwise it will take $\mathcal{O}(mt)$-time (since populating C_M will dominate in this case). □

Details of Merge Plan

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 $e_{succ}^1, .., e_{succ}^n$, 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 $e_{succ}^1, .., e_{succ}^n$ 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_1^i (See Figure 3.7).

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 $\mathsf{EBWT}(G)_1$, so $\mathsf{EBWT}(G)_1$ 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 $\mathsf{EBWT}(G)_1$.

Here, 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.

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

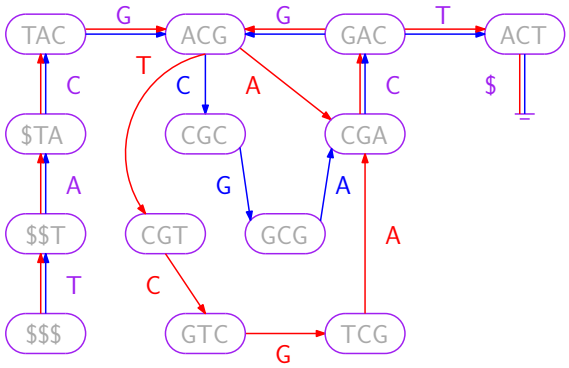
are not strictly those that share the same destination; only those edges that are demarcated and share the same final symbol. Thus, in addition to keeping pointers into $\mathsf{EBWT}(G)_1$ or $\mathsf{EBWT}(G)_2$, we also maintain a vector of counters which contain the number of characters for each (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.

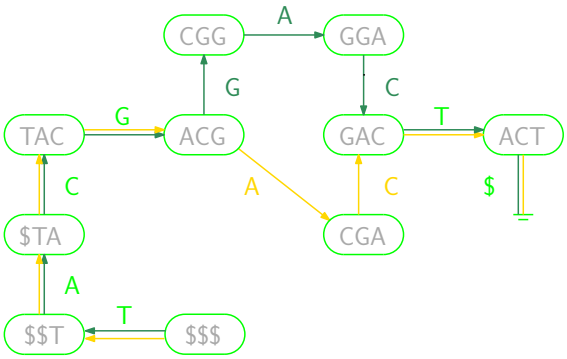
We produce D_M using the merge plan after the first refinement iteration. The intervals at this point are identical to that encoded in D_1 and D_2 so we use the latter rather than consume more space with another copy of an intermediate merge plan. Then, like for B_L we increment a variable tracking position 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.

3.2.5 Discussion

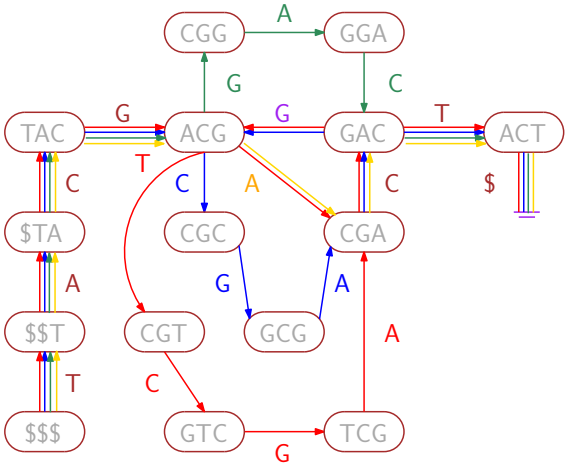
In this section, we present our experimental results on *E. coli* and GenomeTrakr data. We show the scalability of VARIMERGE by demonstrating the time and computational resources needed to build the colored de Bruijn graph for 16,000 strains of salmonella. Next, in order to validate the correctness of 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. We use the SDSL-Lite library [38] to store all succinct vectors.



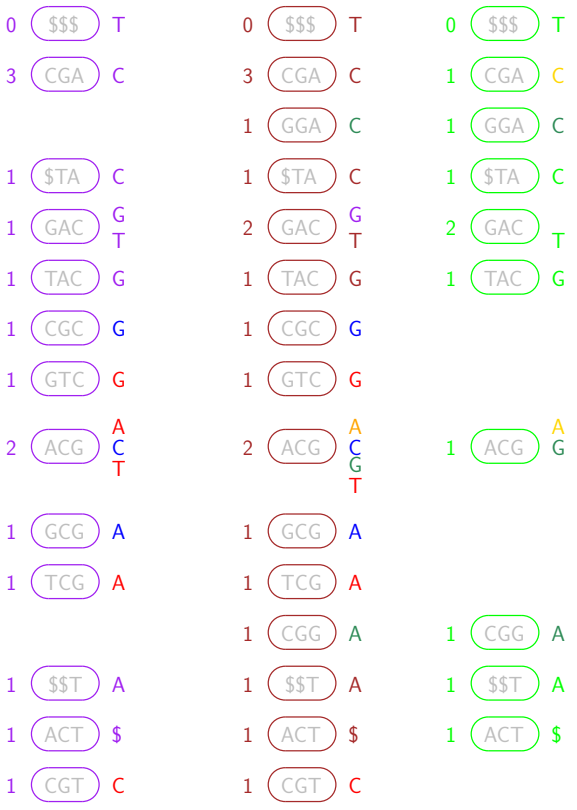
(a) A colored de Bruijn graph G_1.



(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.

Figure 3.3: **(a):** A colored de Bruijn graph consisting of two individual graphs, whose edges are shown in red and blue. (We can consider all nodes to be present in both graphs, so they are shown in purple.) **(b):** A second colored de Bruijn graph, whose edges are green and yellow and lime represents presence in both graphs. **(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.

$\mathsf{L}_1 \leftarrow \emptyset$
$\mathsf{L}_2 \leftarrow \emptyset$
Populate L_1 and L_2 (See "Label Recovery" of Subsection 3.2.3)
Merge L_1 and L_2 into L_M.
Create $\mathsf{EBWT}(G)_M$, B_{LM}, B_{FM} from L_M
Create C_M from C_1 and C_2

Figure 3.4: Naive Merge Algorithm. Because L_1 and L_2 are explicitly constructed a large amount of memory is needed.

▷ Initialize plan to single intervals covering entire $EBWT(G)$s.
$P_1 \leftarrow ([1, |EBWT(G)_1|])$
$P_2 \leftarrow ([1, |EBWT(G)_2|])$
for all $i \in \{1..k\}$ **do**
  $Col_1 \leftarrow$ ***GetCol***(i, G_1)
  $Col_2 \leftarrow$ ***GetCol***(i, G_2)
  ▷ "RefinePlan" is given in the later in the text.
  $(P'_1, P'_2) \leftarrow$ ***RefinePlan***$(P_1, P_2, Col_1, Col_2, i)$
  $(P_1, P_2) \leftarrow (P'_1, P'_2)$

Figure 3.5: The planning step to merge G_1 and G_2.

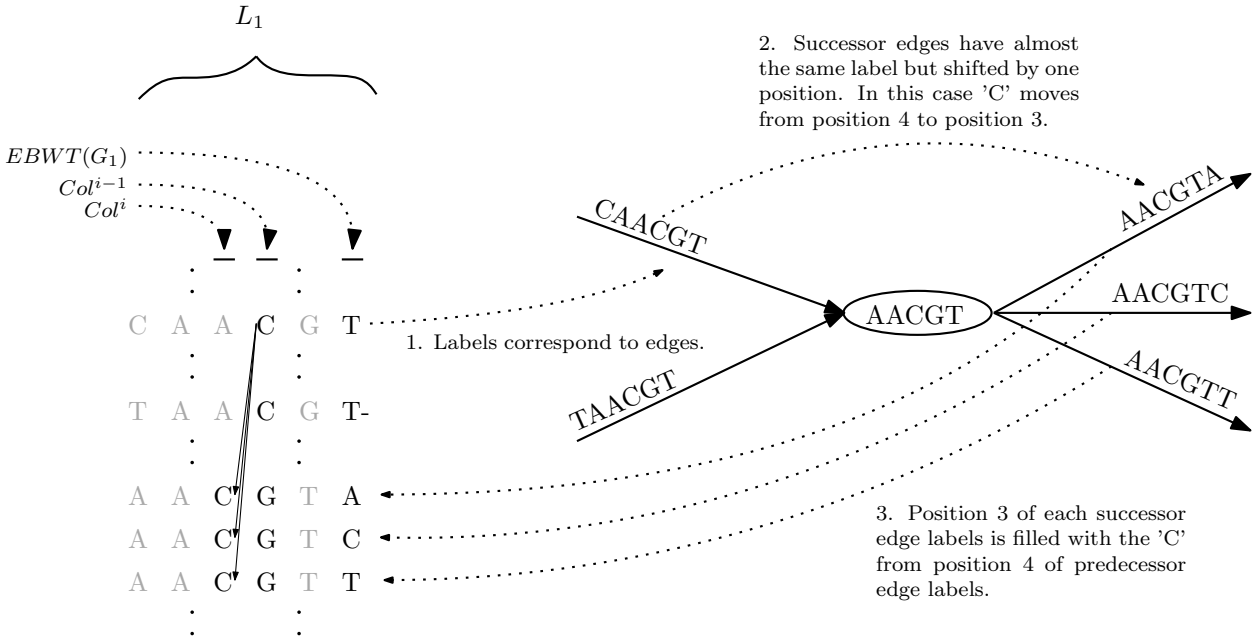


Figure 3.6: Method for populating Col^i based on Col^{i-1} and graph navigation. Black nucleotides represent data that is in memory and valid. Grey represents data that is stored in external memory in VARI but is computed as needed and only exists ephemerally in VARIMERGE. Thus, only three columns are ever present in memory, which is a significant memory savings relative to the full set of edge labels. The three resident vectors are 1.) $EBWT(G_1)$ (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.

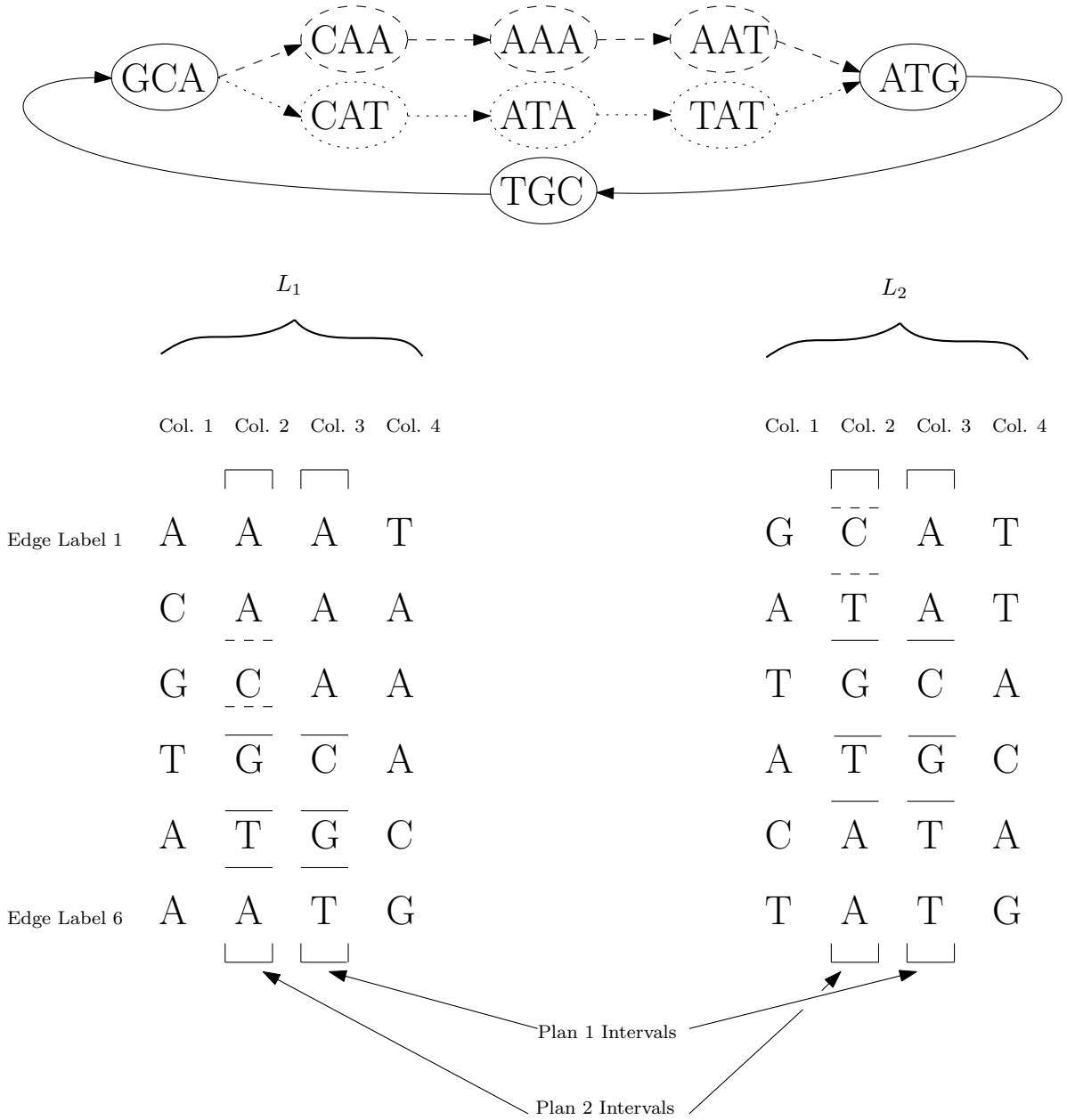


Figure 3.7: Two lists of (conceptual) edge labels and the corresponding de Bruijn graph. Dotted and dashed lines denote edges exclusive to one graph. Solid lines are common to both. The merged graph will contain all components. Plans are refined in decreasing sort precedence order: (Col. 3, Col. 2,. 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 first interval in the L_1 half of Plan 1 is further partitioned into three sub intervals in Plan 2 (for 'A', 'C', and an empty one for 'T') because the first interval in the L_2 half of Plan 1 has a 'T'). 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. For example, TCGA is in the second interval for both graphs in Plan 1, while it is in the fourth interval in Plan 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))
return $(SubP_1, SubP_2)$

procedure REFINEPLAN(P_1, P_2, Col_1, Col_2, i)
$P'_1 \leftarrow ()$
$P'_2 \leftarrow ()$
▷ For each interval in the (equal length) plans...
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*$(SubP_2)$
▷ Capture snapshots of important intermediate plan states.
if $i \in \{1, k-1, k-2\}$ **then**
$S_i \leftarrow (P'_1, P'_2)$
return (P'_1, P'_2)

procedure VARIMERGEPLAN(G_1, G_2)
▷ Initialize plan to single intervals covering entire $EBWT(G)$s.
$P_1 \leftarrow ([1, |EBWT(G)_1|])$
$P_2 \leftarrow ([1, |EBWT(G)_2|])$
▷ Iterate through "edge label matrix" columns in sort precedence order
for all $i \in \{1..k\}$ **do**
$Col_1 \leftarrow$ *GetCol*(i, G_1)
$Col_2 \leftarrow$ *GetCol*(i, G_2)
$(P'_1, P'_2) \leftarrow$ *RefinePlan*$(P_1, P_2, Col_1, Col_2, i)$
$(P_1, P_2) \leftarrow (P'_1, P'_2)$

Figure 3.8: 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. For completeness we give all the pseudocode, including that given in the main paper.

procedure VARIMERGEEXECUTE(G_1, G_2)

▷ Phase 2: Execute plan

▷ For each interval in the (equal length) plans...

for all $j \in \{1..|P_1|\}$ **do**

$NTcounts \leftarrow [0,0,0,0,0]$

$flagcounts \leftarrow [0,0,0,0,0]$

$EBWT(G)_M \leftarrow (), B_{LM} \leftarrow (), flags_M \leftarrow ()$

$G1ptr \leftarrow 1, G2ptr \leftarrow 1$

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*($flagcounts[EBWT(G)_1[G1ptr]] \neq 0$)

$flagcounts[EBWT(G)_1[G1ptr]] \leftarrow +1$

$G1ptr \leftarrow +1$

if $|P_2[j]| = 1$ **then**

$G2ptr \leftarrow +1$

else

$EBWT(G)_M$.*Append*($EBWT(G)_2[G2ptr]$)

B_{LM}.*Append*(*IntervalLast*($G2ptr, S_{k-1}[2]$))

$flags_M$.*Append*($flagcounts[EBWT(G)_2[G2ptr]] \neq 0$)

$flagcounts[EBWT(G)_2[G1ptr]] \leftarrow +1$

$G2ptr \leftarrow +1$

▷ 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*($\mathbf{G2ptr}, \mathbf{S_{k-2}[2]}$) **then**

$flagcounts \leftarrow [0,0,0,0,0]$

$D_M \leftarrow$ *PrefixSum*($NTcounts$)

Figure 3.9: Algorithm to execute the merge plan.

Table 3.4: Comparison between building a succinct colored de Bruijn for the same 8,000 Salmonella strains using VARI versus VARIMERGE.

	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(8k-1)	2.4 B	8,000	271 GB	30 h 49 m	0.63 GB	117 GB	6 h 28 m	114 GB	271 GB	4.6 TB	37 h 27 m	114 GB
VARIMERGE(8k-1)	2.4 B	8,000	137 GB	21 h 27 m	0.63 GB	117 GB	5 h 3 m	114 GB	137 GB	1.5 TB	26 h 30 m	114 GB

Table 3.5: Breakdown of the components of VARIMERGE which was listed in the table above.

	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(4k-1)	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(4k-2)	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(4k-1, 4k-2)	2.4 B	8,000	10 GB	2 h 1 m	0.63 GB	117 GB	1 h 2 m	106 GB	117 GB	N/A	3 h 3 m	106 GB
VARIMERGE(8k-1)	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 3.6: Additional statistics for building a Salmonella 16,000 strain succinct colored de Bruijn graph.

	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(4k-3)	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(4k-4)	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(4k-3, 4k-4)	3.8 B	8,000	17 GB	2 h 59 m	1.00 GB	118 GB	57 m	107 GB	118 GB	N/A	3 h 56 m	108 GB
MERGE(8k-1, 8k-2)	5.8 B	16,000	25 GB	4 h 53 m	1.60 GB	254 GB	2 h 10 m	232 GB	254 GB	N/A	7 h 3 m	233 GB
VARIMERGE(16k)	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

Large-scale Construction using GenomeTrakr

We demonstrate the scalability of VARIMERGE by constructing the succinct de Bruijn graph for 16,000 Salmonella strains from 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 the longer paired end 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 median. 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 (e.g. 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 (4k-1) as well as for a set of just one assembly. Next, we ran our proposed merge algorithm 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. Our proposed algorithm took 49 minutes, no external memory, and 5 GB of RAM to merge the 4,000 color graph with the 1 color graph. This is considerably faster than it would take to build a 4,001 color graph from scratch. 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 (4k-2) 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. 8k-1) using our proposed algorithm in 2 hours 1 minutes, no external memory, and 10 GB of RAM. Thus the VARIMERGE method required a combined 137 GB of RAM, 26 hours 30 minutes of runtime to produce the 8k-1 graph.

In contrast, running VARI on the same 8,000 strains (8k-1) 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. 4k-3 + 4k-4 = 8k-2) and then merged this 8,000 sample graph with the aforementioned 8,000 graph to produce a succinct colored de Bruijn graph of 16,000 samples (i.e. 8k-1 + 8k-2 = 16k-1).

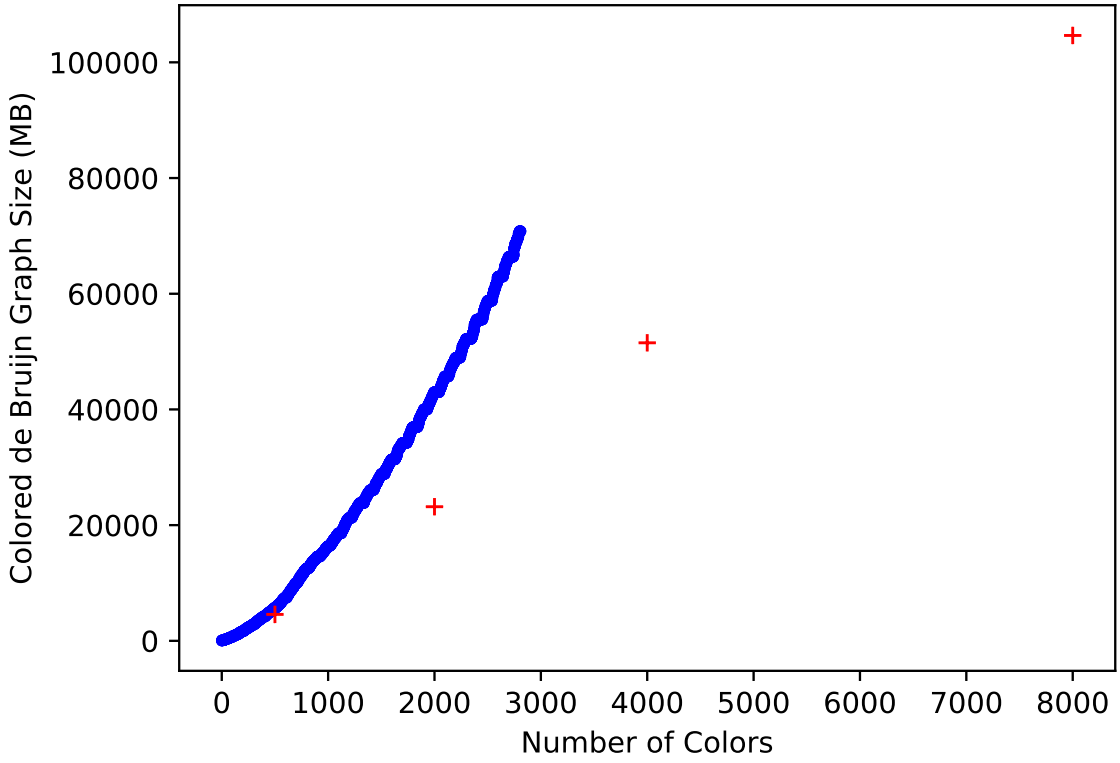


Figure 3.10: Comparison between Bloom Filter Trie (blue dots) and VARIMERGE (red pluses) on isolates from GenomeTrakr. We ran Bloom Filter Trie on 4,000 isolates and plot VARIMERGE results up through 16,000 isolates.

Comparison to Bloom Filter Trie

In addition to demonstrating scalability, we used the Samonella strains from GenomeTrakr to directly compare the data structure space and construction memory of VARIMERGE with Bloom Filter Trie [99] (BFT). We observe both tools have a small memory overhead in construction above

the final data structure. We found super-linear growth in BFT construction memory and that BFT produced a graph 105 GB in size after 4,000 samples were inserted. This is only slightly less space than the 106 GB the VARI data structure requires to represent twice as many samples. Holley et al. [99] 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 homogeneous 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. Though Holley *et al.* compared to Sequence Bloom Trees by Solomon *et al.* [104], we do not because the Sequence Bloom Tree software is designed for transcript querying rather than variant detection.

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 correctness of the merged graph: First, we generated all k-mers for each reference genome, counted all unique k-mers with KMC2 [105], 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 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.

3.2.6 Conclusions

In this section, we propose to further increase the scalability of succinct colored de Bruijn graphs by developing a method to merging smaller graphs in a resource-efficient manner. This allows the colored de Bruijn graph to be constructed for massive size datasets. In addition, our

algorithm provides an efficient means to update a succinct colored de Bruijn graph with additional data as it becomes available. This is useful for example, in the GenomeTrakr database, which is continually being updated with more data on a monthly (or even weekily basis) and the search for a foodborne outbreak requires the analysis of the complete dataset. Thus, rather than rebuilding the colored de Bruijn graph on the new (complete) version of the GenomeTrakr data, dynamically updating the existing one would ensure ideal use of time and resources.

Lastly, our merge algorithm may be applicable to to other prefix-only compressed suffix arrays such as GCSA by Sirén *et al.* [106] and XBW by Ferragina *et al.* [107]. This merits future investigation as these data structures are of both theoretical and practical interest.

Chapter 4

Conclusion

In this work we've seen that the FM-Index can be usefully applied to various genomics problems. Both TWIN and KOHDISTA exploit the FM-Index's ability to find all matches in the target data concurrently as opposed to exhaustively searching all regions of the target data serially. Both VARI and VARIMERGE exploit the compressed nature of the FM-Index to reduce memory consumption.

As both KOHDISTA and VARI represent graphs of many sequences (Rmaps and samples, respectively) it is useful to consider how they differ, and if each data structure would have a useful application in the domain where we've seen the other used. First, we should consider the several notable differences between these structures the impact their application. In the succinct colored de Bruijn graph, common substrings between two samples give rise to $(k-1)-mers$ glued together and any differing regions give rise to bubbles in the graph. Thus the glued $(k-1)-mers$ are effectively seed matches in an alignment and traversal of the graph reveals the mismatching regions containing insertions, deletions, or substitutions. These alignments are found entirely within the data structure after construction and can thus take the form of many-to-many alignments. In contrast, in the GCSA we find alignments between a single query sequence which is not part of the data structure and all compatible matches within the data structure are found concurrently, thus representing a one-to-many alignment.

We might also consider how repeats in the underlying data are handled differently by the two structures. As previously discussed, an important construction step of GCSA is inducing the prefix sortedness property, such that vertices form a totally ordered set based on the lexicographic rank of their corresponding suffix set. Repeats introduce suffixes that may have identical prefixes interfering with total ordering. Thus, the more repeats that are present in a dataset, the more work must occur during construction of a GCSA structure (i.e. prefix doubling enough for unambiguous

lexicographic order) and the resulting structure may consequently be larger. We can also consider each vertex in a de Bruijn graph as representing a suffix in a set of spellable strings, but always just the k length prefix of those strings. As such, in contrast, in the succinct colored de Bruijn graph, all repeated regions longer than k in length result in a collection of single vertices, each with a unique label. So since all vertices are unique, they always have a total order. In essence, the succinct colored de Bruijn graph deals with repeats by collapsing all instances to a single k-length string and keeping track of their origin in the color matrix. In contrast, GCSA deals with repeats by keeping separate vertices for each copy of a repeat, at the expense of expanding the size of the graph to induce a total order.

Given this, we might consider scenarios where we allow gluing of similar fixed length segments of the KOHDISTA graph and keeping track of the origin Rmaps in a color matrix. Under such a scenario, we would also have to consider the sizing error problem. However, the notion of speculating error could be extended to quantization, where additional vertices are introduced to capture the possibility a given Rmap fragment's true size would yield an adjacent quantization bin. The complementary application, finding sequence alignments to sets of finished genomes is already embodied in the original GCSA work.

Future work could include further compressing the color matrix by marking which color matrix rows are identical to their predecessor in compressible runs in the graph, thus obviating the need to store any '1' bits for those rows. This would be akin to colored compact de Bruijn graphs which need only associate one set of colors with a compacted node instead of with each of the non-compacted nodes it comprises.

As well, the colored de Bruijn graph could be made variable order (in concert with the work of Boucher *et al.* [108] by orienting the color matrix in column major order. If the suggested radix merge method is further developed, it could potentially be used to generate the required LCP array in a variable order succinct colored de Bruijn Graph.

Additionally, one of the key features that made the FM-Index techniques work for optical mapping is that the errors in the data were constrained about the given data – sizing error was

always distributed about the true size and missing sites could be speculated about. This suggests this same technique could be applied to other data sources, such as sequencing platforms that are error prone in the length homopolymer runs, which could be indexed as run lengths stored in a wavelet tree.

Bibliography

[1] H. Li and R. Durbin. Fast and accurate short read alignment with Burrows–Wheeler transform. *Bioinformatics*, 25(14):1754–1760, 2009.

[2] Niranjan Nagarajan and Mihai Pop. Sequence assembly demystified. *Nature Reviews Genetics*, 14(3):157–167, 2013.

[3] R Staden. A new computer method for the storage and manipulation of dna gel reading data. *Nucleic Acids Research*, 8(16):3673–3694, 1980.

[4] Saul B Needleman and Christian D Wunsch. A general method applicable to the search for similarities in the amino acid sequence of two proteins. *Journal of molecular biology*, 48(3):443–453, 1970.

[5] Eugene W Myers. Toward simplifying and accurately formulating fragment assembly. *Journal of Computational Biology*, 2(2):275–290, 1995.

[6] Pavel A Pevzner, Haixu Tang, and Michael S Waterman. An eulerian path approach to dna fragment assembly. *Proceedings of the National Academy of Sciences*, 98(17):9748–9753, 2001.

[7] Udi Manber and Gene Myers. Suffix arrays: a new method for on-line string searches. *siam Journal on Computing*, 22(5):935–948, 1993.

[8] U. Manber and G. W. Myers. Suffix arrays: A new method for on-line string searches. *SIAM J. Sci. Comput.*, 22(5), 1993.

[9] Michael Burrows and David J Wheeler. A block-sorting lossless data compression algorithm. *Technical Reports*, 1994.

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

[11] Paolo Ferragina and Giovanni Manzini. Opportunistic data structures with applications. In *Foundations of Computer Science, 2000. Proceedings. 41st Annual Symposium on*, pages 390–398. IEEE, 2000.

[12] P. Ferragina and G. Manzini. Indexing compressed text. *J. ACM*, 52(4):552–581, 2005.

[13] T. Gagie, G. Navarro, and S. J. Puglisi. New algorithms on wavelet trees and applications to information retrieval. *Theor. Comput Sci*, 426-427, 2012.

[14] R. Ronen, C. Boucher, H. Chitsaz, and P. Pevzner. Sequel: Improving the accuracy of genome assemblies. *Bioinformatics*, 28(12):i188–i196, 2012.

[15] C. Alkan, S. Sajjadian, and E.E. Eichler. Limitations of next-generation genome sequence assembly. *Nat. Methods*, 8(1):61–65, 2010.

[16] S.L. Salzberg. Beware of mis-assembled genomes. *Bioinformatics*, 21(24):4320–4321, 2005.

[17] C. Aston and D.C. Schwartz. *Optical mapping in genomic analysis*, pages 1–17. John Wiley and Sons, Ltd, 2006.

[18] Dimalanta et al. A microfluidic system for large dna molecule arrays. *Anal. Chem.*, 76(18):5293–5301, 2004.

[19] R. K. Neely, J. Deen, and J. Hofkens. Optical mapping of DNA: single-molecule-based methods for mapping genome. *Biopolymers*, 95(5):298–311, 2011.

[20] T. Anantharaman and B. Mishra. A probabilistic analysis of false positives in optical map alignment and validation. In *Proc. of WABI*, pages 27–40, 2001.

[21] Harper VanSteenHouse, 2013.

[22] B. Teague et al. High-resolution human genome structure by single-molecule analysis. *Proc. Natl. Acad. Sci.*, 107(24):10848–10853, 2010.

[23] S. Reslewic et al. Whole-genome shotgun optical mapping of *rhodospirillum rubrum*. *Appl. Environ. Microbiol.*, 71(9):5511–5522, 2005.

[24] S. Zhou et al. A whole-genome shotgun optical map of *yersinia pestis* strain KIM. *Appl. Environ. Microbiol.*, 68(12):6321–6331, 2002.

[25] S. Zhou et al. Shotgun optical mapping of the entire *leishmania major* Friedlin genome. *Mol. Biochem. Parasitol.*, 138(1):97–106, 2004.

[26] Shiguo Zhou et al. Validation of rice genome sequence by optical mapping. *BMC Genomics*, 8(1):278, 2007.

[27] S. Zhou et al. A single molecule scaffold for the maize genome. *PLoS Genet.*, 5(11):e1000711, 11 2009.

[28] D. M. Church et al. Lineage-specific biology revealed by a finished genome assembly of the mouse. *PLoS Biology*, 7(5):e1000112+, 2009.

[29] Y. Dong et al. Sequencing and automated whole-genome optical mapping of the genome of a domestic goat (*capra hircus*). *Nat. Biotechnol.*, 31(2):136–141, 2013.

[30] J. T. Howard et al. *De Novo* high-coverage sequencing and annotated assemblies of the budgerigar genome, 2013.

[31] S. Chamala et al. Assembly and validation of the genome of the nonmodel basal angiosperm *amborella*. *Science*, 342(6165):1516–1517, 2013.

[32] Helga Thorvaldsdòttir, James T. Robinson, and Jill P. Mesirov. Integrative genomics viewer (igv): High-performance genomics data visualization and exploration. *Brief. Bioinform.*, 14(2):178–92, 2013.

[33] Anton Valouev et al. Alignment of optical maps. *J. Comp. Biol.*, 13(2):442–462, 2006.

[34] N. Nagarajan, T. D Read, and M. Pop. Scaffolding and validation of bacterial genome assemblies using optical restriction maps. *Bioinformatics*, 24(10):1229–1235, 2008.

[35] H.C Lin et al. Agora: Assembly guided by optical restriction alignment. *BMC Bioinformatics*, 12, 2012.

[36] M. Antoniotti, T. Anantharaman, S. Paxia, and B. Mishra. Genomics via optical mapping iv: sequence validation via optical map matching. Technical report, New York University, 2001.

[37] Keith R Bradnam et al. Assemblathon 2: evaluating *de novo* methods of genome assembly in three vertebrate species. *GigaScience*, 2(1):1–31, 2013.

[38] Simon Gog and Matthias Petri. Optimized succinct data structures for massive data. *Software Pract. Expr.*, to appear.

[39] A. Gurevich, V. Saveliev, N. Vyahhi, and G. Tesler. Quast: quality assessment tool for genome assemblies. *Bioinformatics*, 29(8):1072–1075, 2013.

[40] A. Bankevich and others. Spades: a new genome assembly algorithm and its applications to single-cell sequencing. *J. Comp. Biol.*, 19(5):455–477, 2012.

[41] Y. Kawahara et al. Improvement of the *oryza sativa nipponbare* reference genome using next generation sequence and optical map data. *Rice*, 6(4):1–10, 2013.

[42] J. R. Miller et al. Aggressive assembly of pyrosequencing reads with mates. *Bioinformatics*, 24:2818–2824, 2008.

[43] J.W. Kent. Blat–the blast-like alignment tool. *Genome Res.*, 12(4):656–664, 2002.

[44] A Zimin and others. Sequencing and assembly of the 22-gb loblolly pine genome. *Genetics*, 196(3):875–890, 2014.

[45] J. Sirén, N. Välimäki, and V. Mäkinen. Indexing graphs for path queries with applications in genome research. *IEEE/ACM Trans Comput Biol Bioinform*, To appear, 2014.

[46] Martin D Muggli, Simon J Puglisi, Roy Ronen, and Christina Boucher. Misassembly detection using paired-end sequence reads and optical mapping data. *Bioinformatics*, 31(12):i80–i88, 2015.

[47] A. Valouev, D.C. Schwartz, S. Zhou, and M.S. Waterman. An algorithm for assembly of ordered restriction maps from single DNA molecules. *Proc Natl Acad Sci*, 103(43):15770–15775, 2006.

[48] L. M. Mendelowitz et al. Maligner: a fast ordered restriction map aligner. *Bioinformatics*, 32(7):1016–1022, 2016.

[49] Alden King-Yung Leung et al. Omblast: alignment tool for optical mapping using a seed-and-extend approach. *Bioinformatics*, page btw620, 2016.

[50] Davide Verzotto et al. Optima: Sensitive and accurate whole-genome alignment of error-prone genomic maps by combinatorial indexing and technology-agnostic statistical analysis. *GigaScience*, 5(1):2, 2016.

[51] B. Langmead, C. Trapnell, M. Pop, and S.L. Salzberg. Ultrafast and memory-efficient alignment of short dna sequences to the human genome. *Genome Biology*, 10(3):R25–10, 2009.

[52] M. Muggli et al. Efficient indexed alignment of contigs to optical maps. In *Proceedings WABI*, pages 68–81, 2014.

[53] R.M Idury and M.S. Waterman. A new algorithm for dna sequence assembly. *J. Comp. Biol.*, 2(2):291–306, 1995.

[54] P. E. Compeau, P. A. Pevzner, and G. Tesler. How to apply de bruijn graphs to genome assembly. *Nature Biotechnology*, 29:987–991, 2011.

[55] M. Muggli, S.J. Puglisi, R. Ronen, and C. Boucher. Misassembly detection using paired-end sequence reads and optical mapping data. *Bioinformatics*, 31(12):i80–i88, 2015.

[56] 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.

[57] Genome 10K Community of Scientists. Genome 10k: A proposal to obtain whole-genome sequence for 10,000 vertebrate species. *Journal of Heredity*, 100(6):659–674, 2009.

[58] Robinson et al. Creating a buzz about insect genomes. *Science*, 331(6023), 2011.

[59] M. Causse et al. Whole genome resequencing in tomato reveals variation associated with introgression and breeding events. *BMC Genomics*, 14:791, 2013.

[60] M. Kobayashi et al. Genome-wide analysis of intraspecific DNA polymorphism in "micro-tom", a model cultivar of tomato (solanum lycopersicum). *Plant Cell Physiology*, 55(2):445–454, 2014.

[61] D. Weigel and R. Mott. The 1001 genomes project for Arabidopsis thaliana. *Genome Biology*, 10(5):107, 2009.

[62] EMBL-EBI Metagenomics. Local surveillance of infectious diseases and antimicrobial resistance from sewage, 2016.

[63] Ruth R Miller, Vincent Montoya, Jennifer L Gardy, David M Patrick, and Patrick Tang. Metagenomics for pathogen detection in public health. *Genome Medicine*, 5(9):1, 2013.

[64] F. Baquero et al. Metagenomic epidemiology: a public health need for the control of antimicrobial resistance. *Clinical Microbiology and Infection*, 18(4):67–73, 2012.

[65] Jesse A. Port, Alison C. Cullen, James C. Wallace, Marissa N. Smith, and Elaine M. Faustman. Metagenomic frameworks for monitoring antibiotic resistance in aquatic environments. *Environmental. Health Perspectives*, 122(3), 2014.

[66] The White House. National action plan for combating antibiotic-resistant bacteria. *Washington, DC*, 2015.

[67] Food and Agricultural Organization of the United Nations. The FAO action plan on antimicrobial resistance 2016-2020, 2016.

[68] Fernando Baquero, Ana-Sofia P Tedim, and Teresa M Coque. Antibiotic resistance shaping multi-level population biology of bacteria. *Frontiers in Microbiology*, 4:15, 2013.

[69] R Craig MacLean, Alex R Hall, Gabriel G Perron, and Angus Buckling. The population genetics of antibiotic resistance: integrating molecular mechanisms and treatment contexts. *Nature Reviews Genetics*, 11(6):405–414, 2010.

[70] Noelle R Noyes, Xiang Yang, Lyndsey M Linke, Roberta J Magnuson, Adam Dettenwanger, Shaun Cook, Ifigenia Geornaras, Dale E Woerner, Sheryl P Gow, Tim A McAllister, et al. Resistome diversity in cattle and the environment decreases during beef production. *eLife*, 5:e13195, 2016.

[71] Paula King, Long K Pham, Shannon Waltz, Dan Sphar, Robert T Yamamoto, Douglas Conrad, Randy Taplitz, Francesca Torriani, and R Allyn Forsyth. Longitudinal metagenomic analysis of hospital air identifies clinically relevant microbes. *PLoS ONE*, 11(8):e0160124, 2016.

[72] A. Bowe, T. Onodera, K. Sadakane, and T. Shibuya. Succinct de Bruijn graphs. In *Proc. WABI*, pages 225–235, 2012.

[73] J.T. Simpson et al. Abyss: A parallel assembler for short read sequence data. *Genome Res.*, 19(6):1117–1123, 2009.

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

[75] D. Okanohara and K. Sadakane. Practical entropy-compressed rank/select dictionary. In *Proc. ALENEX*, pages 60–70. SIAM, 2007.

[76] 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.

[77] M. Burrows and D. J. Wheeler. A block sorting lossless data compression algorithm. Technical Report 124, Digital Equipment Corporation, 1994.

[78] 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.

[79] 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.

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

[81] Guillaume Holley, Roland Wittler, and Jens Stoye. Bloom filter trie–a data structure for pan-genome storage. *Algorithms in Bioinformatics*, pages 217–230, 2015.

[82] Shoshana Marcus, Hayan Lee, and Michael C Schatz. Splitmem: A graphical algorithm for pan-genome analysis with suffix skips. *Bioinformatics*, 30(24):3476–3483, 2014.

[83] Y. Lin, S. Nurk, and P. Pevzner. What is the difference between the breakpoint graph and the de bruijn graph? *BMC Genomics*, 15(Suppl 6):S6, 2014.

[84] Gonzalo Navarro. *Compact Data Structures - A Practical Approach*. Cambridge University Press, 2016.

[85] R. Raman, V. Raman, and S. Rao Satti. Succinct indexable dictionaries with applications to encoding k-ary trees, prefix sums and multisets. *ACM Trans. Algorithms*, 3:43, 2007.

[86] Peter Elias. Efficient storage and retrieval by content and address of static files. *Journal of the ACM*, 21(2):246–260, 1974.

[87] Robert Mario Fano. *On the number of bits required to implement an associative memory*. Massachusetts Institute of Technology, Project MAC, 1971.

[88] Sebastian Deorowicz, Marek Kokot, Szymon Grabowski, and Agnieszka Debudaj-Grabysz. KMC 2: Fast and resource-frugal k-mer counting. *Bioinformatics*, 31(10):1569–1576, 2015.

[89] T. Tanaka et al. The rice annotation project database (RAP-DB): 2008 update. *Nucleic Acids Research*, 36:D1028–33, 2008.

[90] P. Schnable et al. The B73 maize genome: Complexity, diversity, and dynamics. *Science*, 326:1112–1115, 2009.

[91] D. Swarbreck et al. The Arabidopsis information resource (TAIR): gene structure and function annotation. *Nucleic Acids Research.*, 36:D1009–14, 2008.

[92] Anthony M Bolger, Marc Lohse, and Bjoern Usadel. Trimmomatic: a flexible trimmer for Illumina sequence data. *Bioinformatics*, 30(15):2114–2120, 2014.

[93] Heather A Carleton and Peter Gerner-Smidt. Whole-genome sequencing is taking over foodborne disease surveillance. *Microbe*, 11:311–317, 2016.

[94] 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.

[95] James B Pettengill, Arthur W Pightling, Joseph D Baugher, Hugh Rand, and Errol 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.

[96] Martin CJ Maiden, Melissa J Jansen Van Rensburg, James E Bray, Sarah G Earle, Suzanne A Ford, Keith A Jolley, and Noel D McCarthy. Mlst revisited: the gene-by-gene approach to bacterial genomics. *Nature Reviews. Microbiology*, 11(10):728, 2013.

[97] Martin D Muggli, Alexander Bowe, Noelle R Noyes, Paul S Morley, Keith E Belk, Robert Raymond, Travis Gagie, Simon J Puglisi, and Christina Boucher. Succinct colored de bruijn graphs. *Bioinformatics*, 2017.

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

[99] Guillaume Holley, Roland Wittler, and Jens Stoye. Bloom filter trie–a data structure for pan-genome storage. In *International Workshop on Algorithms in Bioinformatics*, pages 217–230. Springer, 2015.

[100] Uwe Baier, Timo Beller, and Enno Ohlebusch. Graphical pan-genome analysis with compressed suffix trees and the burrows–wheeler transform. *Bioinformatics*, 32(4):497–504, 2015.

[101] Ilia Minkin, Son Pham, and Paul Medvedev. Twopaco: An efficient algorithm to build the compacted de bruijn graph from many complete genomes. *Bioinformatics*, page btw609, 2016.

[102] Jouni Sirén. Burrows-wheeler transform for terabases. In *Data Compression Conference (DCC), 2016*, pages 211–220. IEEE, 2016.

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

[104] Brad Solomon and Carleton Kingsford. Large-scale search of transcriptomic read sets with sequence bloom trees. *bioRxiv*, page 017087, 2015.

[105] Sebastian Deorowicz, Marek Kokot, Szymon Grabowski, and Agnieszka Debudaj-Grabysz. Kmc 2: Fast and resource-frugal k-mer counting. *Bioinformatics*, 31(10):1569–1576, 2015.

[106] 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.

[107] P. Ferragina, F. Luccio, G. Manzini, and S. Muthukrishnan. Compressing and indexing labeled trees, with applications. *JACM*, 57(1):4, 2009.

[108] Christina Boucher, Alex Bowe, Travis Gagie, Simon J Puglisi, and Kunihiko Sadakane. Variable-order de bruijn graphs. In *Data Compression Conference (DCC), 2015*, pages 383–392. IEEE, 2015.