USING THE MULTI-STRING BURROW-WHEELER TRANSFORM FOR HIGH-THROUGHPUT SEQUENCE ANALYSIS

James Matthew Holt

A dissertation submitted to the faculty at the University of North Carolina at Chapel Hill in partial fulfillment of the requirements for the degree of Doctor of Philosophy in the Department of Mathematics in the College of Arts and Sciences.

Chapel Hill
2016

ABSTRACT

James Matthew Holt: Using the Multi-String Burrow-Wheeler Transform for High-Throughput Sequence Analysis
(Under the direction of Leonard McMillan)

The throughput of sequencing technologies has created a bottleneck where raw sequence files are stored in an un-indexed format on disk. Alignment to a reference genome is the most common pre-processing method for indexing this data, but alignment requires *a priori* knowledge of a reference sequence, and often loses a significant amount of sequencing data due to biases. Sequencing data can instead be stored in a lossless, compressed, indexed format using the multi-string Burrows Wheeler Transform (BWT).

This dissertation introduces three algorithms that enable faster construction of the BWT for sequencing datasets. The first two algorithms are a merge algorithm for merging two or more BWTs into a single BWT and a merge-based divide-and-conquer algorithm that will construct a BWT from any sequencing dataset. The third algorithm is an induced sorting algorithm that constructs the BWT from any string collection and is well-suited for building BWTs of long-read sequencing datasets. These algorithms are evaluated based on their efficiency and utility in constructing BWTs of different types of sequencing data. This dissertation also introduces two applications of the BWT: long-read error correction and a set of biologically motivated sequence search tools. The long-read error correction is evaluated based on accuracy and efficiency of the correction.

Our analyses show that the BWT of almost all sequencing datasets can now be efficiently constructed. Once constructed, we show that the BWT offers significant utility in performing fast searches as well as fast and accurate long read corrections. Additionally, we highlight several use cases of the BWT-based web tools in answering biologically motivated problems.

This dissertation is dedicated to my wife Caroline for her unwavering support through graduate school and my son Luke for providing motivation to actually graduate when I said I would. He's also adorable.

ACKNOWLEDGEMENTS

First and foremost, I would like to thank my savior Jesus Christ for providing me and my family with all the things we have needed. Without him, I would not be where I am today.

I would also like to thank my advisor Leonard McMillan for his support and enthusiasm for bioinformatics. I also want to thank the members of my committee Jan Prins, Vladimir Jojic, Fernando Pardo-Manuel de Villena, and Yun Li for their support and constructive feedback. I wish to thank past and present members of Computational Biology lab who have provided both help and occasional lab entertainment: Leonard McMillan, Jeremy Wang, Catie Welsh, Shunping Huang, Ping Fu, Katy Kao, Zhaoxi Zhang, Piyush Aggarwal, Seth Greenstein, Sebastian Sigmon, and Anwica Kashfeen. I also wish to thank all of the members of the Systems Genetics group, especially Fernando Pardo-Manuel de Villena and Andrew Parker Morgan for their numerous collaborations and willingness to explain genetics.

I'd like to thank all of the members of the UNC Computer Science staff for their administrative and technical assistance. Thank you to the NIH for the grants that have funded my education and research at UNC. I also want to thank my mother Kim, father Barney, and sister Allison for all their encouragement, support, and love for all of my life. Thank you to all my friends and family for the fun, laughter, and motivation. I'd like to thank my son Luke for being a little bundle of joy. Finally, I'd like to thank my wife Caroline for endless support and love throughout everything. Love you, Caroline.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1

INTRODUCTION

1.1 The computational bottleneck in bioinformatics

The throughput of next generation sequencing (NGS) technologies has increased at such a rate that it is now on the cusp of outpacing downstream computational and analysis pipelines [Kahn, 2011]. The result is a bottleneck where huge datasets are held on secondary storage (disk) while awaiting processing. Raw sequence (ex. FASTQ) files, composed of sequence and quality strings, are the most common intermediate piling up at this bottleneck. Moreover, the rapid development of new analysis tools has led to a culture of archiving raw sequence files for reanalysis in the future. This hoarding tradition reflects an entrenched notion that the costs of data generation far exceeds the costs of analysis. The storage overhead of this bottleneck can be somewhat alleviated through the use of compression. However, most analyses require additional computational stages to decompress datasets prior to their use, which further impacts the throughput of subsequent analyses. This, in turn, has led to the need for algorithms which can operate directly on compressed data [Loh et al., 2012].

The goal of next generation sequencing is to read in sequences of DNA, RNA, or processed DNA such that they can be analyzed. Generally speaking, these sequences are composed of long chains of four different nucleotides that are commonly represented by four characters: A, C, G, and T. In complex organisms, the strands of DNA (called chromosomes) can be millions of nucleotides long leading to whole genomes that are billions of nucleotides long. Currently, NGS technologies are not capable of directly reading these long strands of DNA. Instead, they rely on fragmenting the sequence into smaller pieces and reading those smaller pieces using the sequencing technology. These smaller reads are usually hundreds or thousands of nucleotides long and have an expected error rate from the sequencing technology. A typical raw sequence file will include both the nucleotide sequence read by the technology and a quality score indicating the confidence in each nucleotide in the sequence.

Others have previously proposed representing the raw sequencing data using the Burrows Wheeler Transform (BWT) and FM-index [Mantaci et al., 2005, Bauer et al., 2011, Cox et al., 2012a]. The BWT is more compressible than raw sequencing data, and the FM-index is created in a way that is suitable for direct queries by downstream tools even when the BWT is compressed. Furthermore, the BWT is a lossless transform of the sequences that only needs to be computed once for any given dataset. Once computed, the FM-index enables arbitrary, exact matching searches for k length strings in $O(k)$ steps. These searches can be used to count the frequency of the k length string and/or extract all sequenced reads that contain the k length string as a substring.

Despite the benefits of compression and arbitrary searches, there are relatively few efficient algorithms for constructing the BWT of raw sequencing data. While many algorithms exist for constructing the BWT of a single, long string, only a handful exist for constructing the BWT of a string collection such as a sequencing dataset. The ones that do exist also tend to work well for only a particular type of dataset. For example, the BCR algorithms [Bauer et al., 2011, 2013] work well for short, uniform length reads like those produced by an Illumina[1] sequencer. Unfortunately, the algorithm requires more computation time as the reads become longer, leading to lengthy computations for long read sequencing datasets like those produced by PacBio[2] or nanopore[3] sequencers. This has created a barrier of entry for researchers who do not have efficient ways to compute the BWT and FM-index for their datasets.

Once a BWT and FM-index are constructed, they can be applied as the underlying data structures for accessing raw sequencing data, enabling methods that solve biological problems like *de novo* assembly [Simpson and Durbin, 2010] or splice junction detection [Cox et al., 2012b]. Unfortunately, it is not immediately obvious how to use the structures unless one is first familiar with the subroutines used to compute FM-index queries. As a result, there is a second barrier of entry where researchers are unsure how to query the data stored in a BWT and use it in a biologically meaningful way.

In this dissertation, three new algorithms for the multi-string BWT are introduced: a multi-string BWT merge algorithm, a merge-based multi-string BWT construction algorithm, and an induced

[1]http://www.illumina.com
[2]http://www.pacb.com
[3]https://nanoporetech.com

sorting based multi-string BWT construction algorithm. These three methods are primarily geared towards addressing the problem of efficiently computing BWTs of long read sequencing datasets. This dissertation also describes two applications of the BWT and FM-index to addressing biological problems: long-read sequence correction and sequence visualizations. Long-read sequence correction is an automated process that reduces the noise caused by sequencing errors in long-read datasets, enabling downstream analyses such as *de novo* assembly and alignment to operate on a cleaner set of inputs. The sequence visualizations enable researchers to directly access raw sequence datasets stored on a remote server while masking the complexities of the BWT and FM-index queries. These tools provide a direct interface to the sequenced reads without losing information from a preprocess like alignment or assembly. Since the BWT is a lossless transform that only need to be computed and indexed once for a dataset, it enables researchers to focus their efforts on designing biologically informative queries instead of repeatedly preprocessing the data.

1.2 Thesis statement

The multi-string Burrows-Wheeler Transform and associated FM-index are efficient data structures for storing and indexing raw sequencing data. By developing methods to build these structures for different types of sequencing data, we enable alternate methods for solving biologically relevant problems without loss of the underlying, raw sequence data.

1.3 Contributions

This dissertation introduces three main BWT algorithms: an algorithm for merging multiple BWTs into a single BWT, a merge-based multi-string BWT construction algorithm, and an induced-sorting based multi-string BWT construction algorithm. Both construction algorithms tend to be better suited for long read sequencing datasets than other construction algorithms. This dissertation also introduces two methods that use the BWT as an underlying data structure for querying a sequence dataset: a long-read error correction method that uses the BWT and FM-index of a short-read sequencing dataset and a collection of sequence-centric visualization tools for directly accessing raw sequencing data through a BWT and FM-index.

1.4 Structure

Chapter 2 provides an overview and background of the BWT and FM-index. Chapter 3 describes an algorithm for merging one or more BWTs together into a single BWT. Chapters 4 and 5 describe two algorithms for constructing multi-string BWTs of long-read sequencing datasets that respectively use a merge-based approach and an induced sorting approach. Chapter 6 describes a method for long-read error correction using the BWT of short-read sequencing datasets. Chapter 7 describes how to use the BWT to create sequence-centric web tools for accessing and visualizing raw sequencing data. Chapter 8 provides a summary of this dissertation and possible areas of future work.

CHAPTER 2

BACKGROUND

2.1 The suffix array

The suffix array is a data structure that enables searches for substrings in a single string [Manber and Myers, 1993]. Let T be a string of length N that is terminated by a unique end-of-string character, '\$', that is lexicographically smaller than all other symbols in T. One can naïvely construct the suffix array of T by first enumerating all of its suffixes, $S_0 = T[0 : N]$, $S_1 = T[1 : N]$, ..., $S_{N-2} = T[N-2 : N]$, and $S_{N-1} = T[N-1 : N]$, and then sorting them lexicographically. The suffix indices after the sort is the suffix array of T. While the suffix array is defined as the list of suffix indices, it is often visually represented as the actual suffixes. Table 2.1 illustrates the construction of the suffix array for the string "ALABAMA\$".

The suffix array is a useful data structure primarily because it organizes similar substrings so that they are adjacent to each other in the structure. Since suffix arrays represent a sorted list, a standard binary search can be used to search for exact matching substrings within the text T. Manber and Myers [1993] show that this binary search can be performed in $O(k + \log(N))$ time for a text of length N and a substring of length k.

The main disadvantage of a suffix array is the memory cost for storing the structure. The array stores N values that are at most $(N - 1)$. As a result, the total storage cost of the array is $O(N * \log(N))$ bits. For any long text, the size of the structure may be too large to fit in main memory. For example, the human genome is approximately three billion basepairs long, so the suffix array for the human genome requires approximately 12 GB of memory. While this may not be prohibitively large for modern computers, there are smaller structures that can perform faster searches.

2.2 The Burrows-Wheeler Transform

The Burrows-Wheeler Transform (**BWT**) was originally introduced as a method for permuting the symbols in a string to improve its compressibility [Burrows and Wheeler, 1994]. Interestingly,

Index	Suffix	Sorted Suffixes	Suffix Array
0	ALABAMA$	$	7
1	LABAMA$	A$	6
2	ABAMA$	ABAMA$	2
3	BAMA$	ALABAMA$	0
4	AMA$	AMA$	4
5	MA$	BAMA$	3
6	A$	LABAMA$	1
7	$	MA$	5

Table 2.1: Suffix array example. This table demonstrates how to naïvely construct the suffix array for a string “ALABAMA$". First, list every suffix of the input string in the “Suffix" column. Then, lexicographically sort those suffixes in the “Sorted Suffixes" column. In the “Suffix Array" column, each sorted suffix is paired with its corresponding start index in our original string. For example, the suffix “$" starts at index 7 in “ALABAMA$", so the first value in the suffix array is 7. The array of numerical values forming this column is the suffix array for “ALABAMA$".

Index	Rotations	Sorted Rotations	Suffix Array	BWT
0	ALABAMA$	$ALABAM**A**	7	**A**
1	LABAMA$A	A$ALABA**M**	6	**M**
2	ABAMA$AL	ABAMA$A**L**	2	**L**
3	BAMA$ALA	ALABAMA**$**	0	**$**
4	AMA$ALAB	AMA$ALA**B**	4	**B**
5	MA$ALABA	BAMA$AL**A**	3	**A**
6	A$ALABAM	LABAMA$**A**	1	**A**
7	$ALABAMA	MA$ALAB**A**	5	**A**

Table 2.2: BWT example. This table demonstrates how to naïvely construct the BWT for a string “ALABAMA$". First, list every rotation of the input string in the “Rotations" column. Then, lexicographically sort those rotations in the “Sorted Rotations" column. The values in the “BWT" column are generated by taking the corresponding sorted rotation and storing the symbol preceding that rotation (this is also the last symbol in the rotation, bolded for visual aid). Note that changing the suffixes to rotations has no effect on the values stored in the suffix array.

the permutation is directly related to a suffix array for the same string. Given a single string and its suffix array, the BWT is a concatenation of the symbols preceding each suffix represented by the suffix array. Alternatively, by representing the suffixes as rotations (cyclic suffixes), the BWT can be thought of as a concatenation of the last characters in each rotation represented by the suffix array. Thus the BWT implicitly represents a suffix array. Table 2.2 demonstrates how to construct the BWT for the string “ALABAMA$".

The BWT enables compression because it tends to create long substrings of the same symbol called *runs* [Burrows and Wheeler, 1994]. This tendency to create long runs is largely due to its relationship to the suffix array. Since the suffix array represents the sorted order of all rotations of the input, rotations that start with similar substrings will be adjacent in the array. Additionally, the expectation is that similar substrings are preceded by the same symbol. Since the BWT is defined as the concatenation of these predecessor symbols, the expectation is that long runs will occur in the BWT when the same substring is repeated multiple times in the original string. For example, consider a typical English text document. Words such as “the" will occur many times through the document. In the suffix array, this will appear as suffixes starting with “he " and whose rotations end with symbol ‘t’. As a result, there will likely be long runs of ‘t’ symbols caused by the prevalence of the substring “the" in the text.

Given a BWT of a text, it can be compressed using run-length encoding [Burrows and Wheeler, 1994]. Wherever a run occurs in the BWT, the run can be replaced with a numerical value and a single occurrence of the symbol that makes the run. For example, the string “ALABAMA$" becomes BWT “AML$BAAA" which can be stored using run-length encoding as “AML$B3A". As the length of runs in the BWT increases, the relative compression gained from run-length encoding also increases.

2.3 The Full-text Minute-space index

While the BWT is an interesting data structure for compressing long texts, it is not inherently searchable like the suffix array. However, the suffix array that is implicitly represented by the BWT can be accessed through an auxiliary data structure called the Full-text Minute-space index (**FM-index**) [Ferragina and Manzini, 2001]. The FM-index takes advantage of a relationship between the BWT and suffix array called the Last-to-First Mapping (**LF-mapping**) property [Ferragina and Manzini, 2001]. The LF-mapping property states that j-th occurrence of a symbol c in the BWT (the Last column in a sorted rotation) corresponds in a 1-to-1 manner with the j-th suffix that begins with symbol c in the suffix array (the First column in a sorted rotation). For example, in Table 2.3, the first ‘B’ in the BWT occurs at index 0 and corresponds to the first suffix starting with ‘B’ at index 4. Similarly, the second ‘B’ in the BWT occurs at index 1 and corresponds to the second suffix starting with ‘B’ at index 5. The FM-index pre-calculates and stores these 1-to-1 relationships to enable fast access to the implicit suffix array.

Given a BWT, B, of length N and an alphabet, Σ, of length σ, the FM-index, F, is defined as a two-dimensional matrix of values with length $(N + 1)$ and width σ. For an index i and symbol c, $F[i][c]$ is the total number of occurrences of c before index i in the BWT. Given this definition, the

Index	Suffix Array	BWT	FM-index		
			\$	A	B
0	\$ABABA**B**	**B**	0	0	0
1	AB\$ABA**B**	**B**	0	0	1
2	ABAB\$A**B**	**B**	0	0	2
3	ABABAB**\$**	**\$**	0	0	3
4	B\$ABAB**A**	**A**	1	0	3
5	BAB\$AB**A**	**A**	1	1	3
6	BABAB\$**A**	**A**	1	2	3
Total	—	—	1	3	3
Offsets	—	—	0	1	4

Table 2.3: FM-index example. A sample BWT and FM-index for the string "ABABAB\$". The sorted rotations from the suffix array are shown for visual aid. The FM-index, F, is shown on the far right. For index i and symbol c, $F[i][c]$ is the number of occurrences of c before i in the BWT. Additionally, the last entry in the FM-index (labeled "Total") is the total number of times each symbol occurs in the BWT. These totals are used to derive an "Offsets" array, O, such that $O[c]$ is the index of the first suffix in the suffix array that begins with c.

entire FM-index can be constructed in a linear pass over the BWT. Initialize $F[0]$ to all zeros. For all i from 0 to N, copy $F[i]$ into $F[i+1]$ and increment only $F[i+1][c]$ where $c = B[i]$. Given this definition, the last entry in the FM-index is the total number of occurrences of all characters in the BWT. These total counts are used to trivially calculate Offsets array, O, of length σ such that for all $c \in \Sigma$, $O[c]$ is the first suffix in the implicit suffix array that begins with c. Table 2.3 shows the BWT, FM-index, and Offsets for string "ABABAB\$".

Given a BWT and FM-index, the corresponding suffix array can be implicitly accessed by applying the LF-mapping property. Recall that the LF-mapping property relates the predecessor symbol stored in the BWT to a particular suffix in the suffix array. As a result, substrings that are contained within the suffix array can be accessed without explicitly storing the suffix array. There are two key functions that use the FM-index to access the implicit suffix array in this way.

2.3.1 Retrieving a suffix

The first function is `recoverSuffix(...)` shown in Algorithm 1. The main idea behind this algorithm is to repeatedly calculate the predecessor at a particular index in the BWT by using the FM-index. Given an index i, lines 1-3 find the symbol preceding the implicit suffix at index i and the corresponding index of that suffix the BWT. In lines 4-7, the process of finding predecessor symbols and suffixes is repeated until it returns to the original index i. Since this calculation is done in reverse by finding predecessor symbols, the final step is to reverse the order of the predecessor symbols to obtain the actual suffix. A full example of this method is show in Table 2.4.

The time to find any given predecessor symbol and corresponding index in the BWT is $O(1)$. Since this method recovers the entire suffix as a rotation, it will require a total of $O(N)$ steps to recover the suffix of length N. Note that when this method is called on index 0 (the only suffix starting with the end-of-string symbol '\$'), the method returns a suffix that can be trivially shifted to obtain the original text used to construct the BWT.

```
Data: BWT $B$, FM-index $F$, Offsets $O$, index $i$
Result: The rotational suffix located at index $i$ in the implicit suffix array
Function recoverSuffix($B$, $F$, $O$, $i$)
1    $c \leftarrow B[i]$;
2    $j \leftarrow O[c] + F[i][c]$;
3    $ret \leftarrow c$ ;
4    while $i \neq j$ do
5        $c \leftarrow B[j]$;
6        $j \leftarrow O[c] + F[j][c]$;
7        $ret$.append($c$) ;
     end
8    return reverse($ret$) ;
```

Algorithm 1: Given a BWT and FM-index, this function accesses the implicit suffix array through a reverse traversal of the BWT. Line 1 retrieves the symbol preceding the suffix at index i in the implicit suffix array. Line 2 calculates the index of the suffix corresponding to that symbol and stores it in j. Line 3 saves the symbol as part of a string. Since the suffixes are cyclic (i.e. rotations) by the BWT definition, lines 4-7 repeats the retrieval of predecessors until it reaches the original index i, at which point the algorithm has cycled through the entire suffix in reverse order. Since the symbols are discovered in reverse order, the final step is to reverse the entire string in line 8 before it is returned. An example execution of this process is shown in Table 2.4.

Variable	Lines 1-3	Loop 1	Loop 2	Loop 3	Loop 4	Loop 5	Loop 6
c	B	A	B	A	B	A	\$
j	4	1	5	2	6	3	0
ret	B	BA	BAB	BABA	BABAB	BABABA	BABABA\$

Table 2.4: `recoverSuffix(...)` example. This table shows the state of variables in the `recoverSuffix(...)` function of Algorithm 1 when called using the BWT, FM-index, and Offset array from Table 2.3 and $i = 0$. Lines 1-3 calculate the symbol and suffix preceding the suffix at index 0 in the BWT. Each iteration of the loop calculates the next symbol and suffix preceding the current index j. At the end of loop 6, $j = i$, so the function returns the reverse of ret, "\$ABABAB". In addition to being the correct suffix at index 0 in the BWT, this result can be used to trivially derive the original text for creating the BWT.

2.3.2 Calculating suffix ranges

The second function is `getRange(...)` shown in Algorithm 2. Given a substring p of length k (called a k-**mer**), this method solves for a range in the implicit suffix array such that all suffixes in the range share the k-mer prefix p. The base case is the empty string range defined as $[0, N)$ for a BWT of length N (lines 1-3). In the loop, symbols from the substring are retrieved in reverse order (lines 4-5). For iteration $i \in [1, k]$, the symbol is used to calculate a new range corresponding to the i-mer suffix of p. At the end of iteration i, the values $[l, h)$ store the range of suffixes in the implicit suffix array that are prefixed by the i-mer substring $p[k - i : k)$. In other words, the algorithm implicitly pre-pends symbols to the empty string range until it obtains the substring p. A full example of this method is show in Table 2.5.

As with `recoverSuffix(...)`, this method traverses the values stored in the BWT in reverse order. For any given range, a symbol can be implicitly pre-pended to that range in $O(1)$ steps (lines 5-7). Thus, the range for any arbitrary k-mer can be solved in $O(k)$ steps. Additionally, the count or frequency of a k-mer can be derived trivially as the length of the range, $(h - l)$.

```
Data: BWT $B$, FM-index $F$, Offsets $O$, Prefix $p$
Result: The range of suffixes $[l, h)$ in the suffix array that have the prefix $p$
Function getRange($B$, $F$, $O$, $p$)
1       $k \leftarrow$len($p$);
2       $l \leftarrow 0$;
3       $h \leftarrow$len($B$);
4       for $i \in [1..k]$ do
5           $c \leftarrow p[k - i]$;
6           $l \leftarrow O[c] + F[l][c]$;
7           $h \leftarrow O[c] + F[h][c]$;
        end
8       return $(l, h)$;
```

Algorithm 2: Given a BWT of length N and FM-index, this function calculates the range of indices in the BWT corresponding to suffixes in the implicit suffix array that have a particular prefix p of length k. It starts with the full range $[0, N)$ and implicitly prepends characters from p in reverse order. In other words, the method calculates the range corresponding to suffixes $p[k-1 : k)$, $p[k-2 : k)$, ..., $p[1 : k)$, and finally p. This method takes advantage of the LF-mapping property stored in the FM-index to solve for any given k-length substring (or k-mer) in $O(k)$ steps. An example execution of this process is shown in Table 2.5.

2.3.3 Sampling the FM-index

While the FM-index is useful for the methods described above, it requires a significant amount of memory to store the entire structure. For an alphabet of length σ and a BWT of length N, the

Variable	Lines 1-2	Loop 1	Loop 2	Loop 3	Loop 4
c	—	B	A	B	A
l	0	4	1	5	2
h	7	7	4	7	4
partial substring	“"	B	AB	BAB	ABAB

Table 2.5: `getRange(...)` example. This table shows the state of variables in the `getRange(...)` function of Algorithm 2 when called using the BWT, FM-index, and Offset array from Table 2.3 and $p =$“ABAB". The algorithm starts with the full range ($l, h = [0, 7)$) corresponding to the empty string. The range is modified such that symbols are implicitly prepended to the substring (shown in the partial substring row) with each execution of the main loop. The correct range for suffixes of the substring p are solved in reverse order, and the final execution of the loop solves the range $[2, 4)$ for the full substring “ABAB".

FM-index requires $O(N\sigma)$ integer values. Even for a restricted genomic alphabet, this is too large to be practical.

Fortunately, there is a relatively easy way to sample the FM-index to reduce storage costs by sampling the FM-index [Ferragina and Manzini, 2001]. Given BWT B and FM-index F, recall that the values in $F[i+1]$ were defined earlier based on the values stored at $F[i]$ and $B[i]$. As a result, given an FM-index entry at index $i < j$, the BWT can be linearly traversed to calculate the FM-index entry at index j. If the FM-index is sampled such that every s-th entry is stored, the total memory requirements of the FM-index reduces to $O(\frac{N\sigma}{s})$. Note that this comes at a cost of increasing the time to get any particular FM-index to $O(s)$ due to the linear traversal of B.

2.3.4 Application to bioinformatics

One of the most common ways to index short-read datasets is through a process called alignment. In alignment, the goal is to match a sequencing read (a short string) to a reference genome (a much longer string) such that the edit distance (number of symbol changes, insertions, or deletions) between the read and the reference are minimized. In this way, each read is indexed by a position in the reference genome. In DNA-seq alignment, there are often millions or billions of reads that need to be aligned to the reference genome.

Two prominent alignment tools, BWA [Li and Durbin, 2009] and Bowtie [Langmead et al., 2009], use the BWT and FM-index to quickly align reads. Both methods first construct a BWT and FM-index of the reference genome. Additionally, there is some amount of bookkeeping (i.e. a suffix array) required to store the location of some or all of the suffixes in the original reference genome. Each read is split into short k-mer seeds (e.g. a 100 basepair read may be split into four 25-mer seeds). The tools then search for exact matches to the k-mer seeds in the BWT. If a seed is found, the methods identify the location of the seed in the original reference genome and perform a more extensive local alignment that allows for inexact matching of the full read to the reference. In general, these methods are quite fast because they use the $O(k)$ search algorithm made possible by the BWT and FM-index. Additionally, the method has been adapted for RNA-seq alignment in at least one publicly available tool, Tophat [Trapnell et al., 2009].

2.4 The Multi-String Burrows-Wheeler Transform

Thus far, the BWT has only been discussed in the context of a single input string. However, there are many reasons one might wish to encode a string collection. For example, a reference genome for a complex organism often has multiple strings where each string represents a chromosome for the organism. The BWT of a single string was well-defined because there was only one end-of-string symbol. However, with multiple strings, the relative ordering of the symbols can be permuted leading to many different (and conflicting) orderings of the strings.

While there are many definitions, we adopt the multi-string BWT definition used by Bauer et al. [2011]. In general, this version defines the lexicographic order of end-of-string characters as the lexicographic order of the strings in the collection. In other words, for two strings s_i and s_j in the collection that are terminated with end-of-string symbols $\$_i$ and $\$_j$ respectively, $(s_i < s_j) \iff (\$_i < \$_j)$. This defines an unambiguous order of all suffixes represented by the BWT.

Given the above definition, one can naïvely construct the BWT of a string collection in a method very similar to the naïve construction method for a single string. First, all rotations of all strings are generated. Then, all rotations are sorted lexicographically. Finally, the predecessor symbols to each sorted rotation are concatenated and stored as the BWT for the collection. Additionally, the FM-index for a multi-string BWT is calculated in the exact same way as a single-string BWT. A simple example of a BWT and FM-index for a string collection is shown in Table 2.6.

2.4.1 BWT functions on a string collection

Both `recoverSuffix(...)` and `getRange(...)` functions can be used unchanged on the BWT and FM-index of a string collection. Additionally, the chosen definition of a multi-string BWT enables the extraction of individual strings in the collection using the `recoverSuffix(...)` function. Recall that the definition orders the '\$' character by lexicographic order of the strings in the collection. As a result, if the algorithm starts with a suffix from string s_i, all suffixes that are found through a reverse

Index	Suffix Array	BWT	FM-index		
			$	A	B
0	$ABA**B**	**B**	0	0	0
1	$BBA**B**	**B**	0	0	1
2	AB$A**B**	**B**	0	0	2
3	AB$B**B**	**B**	0	0	3
4	ABAB**$**	**$**	0	0	4
5	B$AB**A**	**A**	1	0	4
6	B$BB**A**	**A**	1	1	4
7	BAB$**A**	**A**	1	2	4
8	BAB$**B**	**B**	1	3	4
9	BBAB**$**	**$**	1	3	5
Total	—	—	2	3	5
Offsets	—	—	0	2	5

Table 2.6: Multi-string BWT example. A sample BWT and FM-index for the string collection, ABAB$, BBAB$. The rotations of each string in the collection are sorted to create the suffix array. The BWT is the concatenation of predecessor symbols (bolded). The FM-index F is defined as before for index i and symbols c such that $F[i][c]$ is the number of occurrences of c before i in the BWT. Note that the order of suffixes beginning with '$' corresponds exactly to the lexicographic ordering of the strings in the collection.

Variable	Lines 1-3	Loop 1	Loop 2	Loop 3	Loop 4
c	B	A	B	A	$
j	5	2	7	4	0
ret	B	BA	BAB	BABA	BABA$

Table 2.7: `recoverSuffix(...)` string collection example. This table shows the state of variables in the `recoverSuffix(...)` function of Algorithm 1 when called using the BWT, FM-index, and Offset array from Table 2.6 and $i = 0$. Note that this traversal never "jumps" into the suffixes created by other strings in the collection. After the final loop, ret is reversed and the lexicographically smallest rotation ("$ABAB") corresponding to the lexicographically smallest string in the collection ("ABAB$") is returned.

traversal are guaranteed to be from string s_i. In other words, in a correct traversal, it is impossible to "jump" from one string in the collection to a different string in the collection. Given this result, calling `recoverSuffix(...)` will only return the rotation for a specific string. An example of the `recoverSuffix(...)` method run on a BWT of a string collection is shown in Table 2.7.

The `getRange(...)` functionality is largely the same as before. However, instead of searching for occurrences of a substring in one string, this function will solve for all occurrences of the substring in all strings in the collection. Again, the count or frequency of a particular substring in the collection is trivially derived as the length of the range returned by this function.

2.4.2 Representing De Bruijn graphs

One way of traversing the information in a string collection is to use an auxiliary data structure called the de Bruijn graph. In a de Bruijn graph, every k-length substring from the collection is represented as a single node [Bruijn, 1946]. The graph also has directed edges drawn between nodes such that the $(k-1)$ suffix of the source node is identical to the $(k-1)$ prefix of the destination node [Bruijn, 1946]. This graph representation reduces the long strings into shorter substrings of a fixed length, allowing for algorithms to then intelligently traverse the graph to perform tasks like *de novo* assembly [Simpson and Durbin, 2010] or read correction [Salmela and Rivals, 2014]. An example de Bruijn graph containing two strings is shown in Figure 2.1.

Generally speaking, de Bruijn graphs are usually explicitly stored for a fixed, small k using either a hash based method or a bloom filter [Chikhi and Rizk, 2013]. Additionally, these graphs are often pruned such that k-mers with low frequency are removed from the graph. The BWT and FM-index can actually be used to implicitly represent a de Bruijn for all sizes of k up to the length of the stored strings. When a node in the graph needs to be accessed, the `getRange(...)` function (see Section 2.3.2) can be used to retrieve the k-mer frequency. If the k-mer frequency is above the pruning threshold, then the k-mer node is considered present in the de Bruijn graph, otherwise it is absent. Additionally, since the `getRange(...)` function is not constrained to a particular k size, this method can be used to represent all de Bruijn graphs for a string collection without needing to explicitly calculate or store the graph.

2.4.3 Application to bioinformatics

A multi-string BWT can be used to replace the concatenation method for building a BWT of the reference genome. However, when built for a reference genome, these two structures are actually very similar simply because the number of strings in a reference genome is relatively small. Most of the implicit suffixes will be in the same order and the ones that are not will likely have little to no effect on the alignment process. Instead, others have previously proposed storing raw sequencing data in a BWT [Mantaci et al., 2005, Bauer et al., 2011, Cox et al., 2012a]. There are two main benefits to applying the BWT to raw sequencing data: compression and indexing.

First, recall that BWTs achieve compression through the process of run-length encoding and that those runs are formed due to repeated substrings within the text. In sequencing data, one source of repeated substrings is true repeated sequence in the genome. Additionally, sequencing runs are

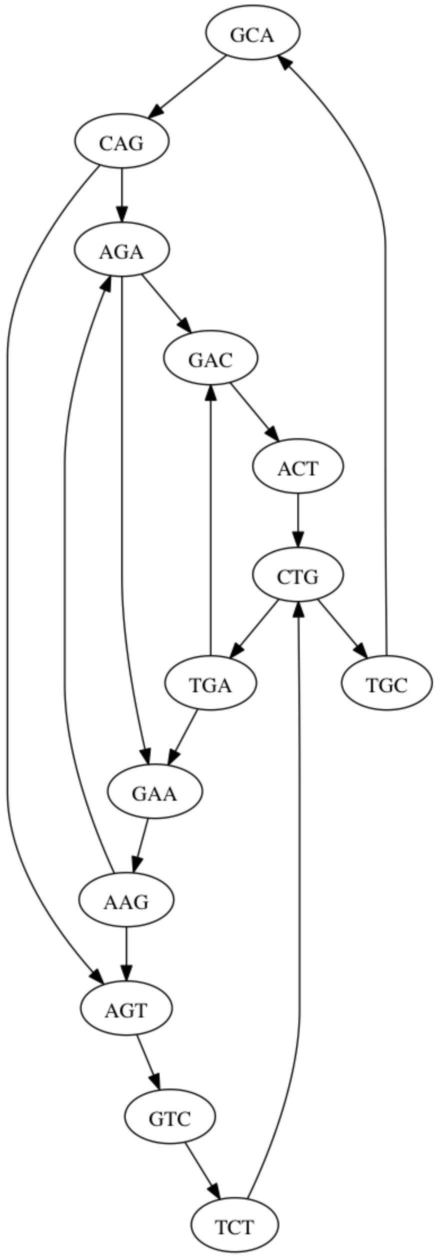


Figure 2.1: Example de Bruijn graph. This figure shows the 3-mer de Bruijn graph for two strings: “CAGACTGCAGT" and “CAGTCTGAAGT". Every 3-mer from the two strings is a node in the graph. An edge indicates that the 2-mer suffix of the source 3-mer is the same as the 2-mer prefix of the destination 3-mer. For example, node “AGA" has suffix “GA", so it has edges to both “GAC" and “GAA".

typically amplified prior to sequencing leading to many copies of the genome that is often referred to as the coverage. However, sequencing technologies are not perfect and generate errors during the sequencing process. These errors work against the compression of BWTs essentially by creating random noise that can break runs apart and reduce the compression ratio.

The second main benefit is that by building an FM-index, the sequencing data becomes searchable. In the context of functions already discussed, this means that there are methods to count the frequency of any given k-mer and to extract every read from a sequencing data that contains a particular substring for further analysis. Additionally, the FM-index of genomic strings is relatively small because the alphabet is generally limited to six characters (‘\$’, ‘A’, ‘C’, ‘G’, ‘N’, and ‘T’). BWTs of sequencing datasets have been used by other authors for splice junction detection [Cox et al., 2012b] and *de novo* assembly [Simpson and Durbin, 2010].

CHAPTER 3

MERGING BURROWS-WHEELER TRANSFORMS

The BWT and FM-index are useful data structures for analyzing genomic sequencing data. They enable compression of the data while simultaneously indexing it for downstream analysis. While there are several methods for constructing the BWT of single strings or string collections, there are few methods for incrementally adding strings to a BWT without recalculating the entire data structure.

In this chapter, we address the problem of merging two or more BWTs such that the result is a BWT containing the combined string collections from each input BWT [Holt and McMillan, 2014a]. Additionally, we require the strings in the resulting BWT to be annotated such that the origin of each string (in terms of which input it came from) can be identified later. The reasons for merging include adding new information to an existing dataset (more data from a sequencer), combining different datasets for comparative analysis, and improving compression.

3.1 Related work

Other researchers have addressed problems related to merging BWTs, and three of these are of particular interest. The first is actually a BWT construction algorithm which incrementally constructs a BWT in blocks and then merges those blocks together [Ferragina et al., 2012]. The algorithm creates partial BWTs in memory. These partial BWTs are not truly independent BWTs because they reference suffixes not included in the partial BWT (they either were processed previously or will be processed later). The partial BWTs are then merged into a final BWT on disk by comparing the suffixes either implicitly or explicitly depending on the location of the suffixes in the partial BWTs. Their algorithm is primarily applicable to constructing a BWT of very long strings. However, it could be adapted for merging by inserting one string at a time as if the string collection were one long string. The memory overhead of the modified algorithm requires reconstituting the string collections for all but one of the inputs (the one used as the starting point), and then iteratively going through each string one at a time until the merged result was constructed.

The second algorithm is also a BWT construction algorithm that creates a multi-string BWT by incrementally inserting a single symbol from each string “columnwise" until all symbols are added to the multi-string BWT [Bauer et al., 2011, 2013]. Given a finished BWT, they also describe how to add new strings to the BWT using this algorithm. This algorithm could be adapted to solve the proposed BWT merging problem by keeping one input in the BWT format and decoding all of the other inputs into their original string collections. Then, their construction algorithm would merge each collection into the BWT in this “columnwise" manner. As with the first algorithm, the main issue with this approach is storage overhead of decoding each BWT into its original string collection.

The third algorithm is a suffix array merge algorithm which computes the combined suffix array for two inputs [Sirén, 2009]. These suffix arrays are actually represented as two multi-string BWTs. The algorithm searches for the strings of one collection in the other BWT to determine a proper interleave of the first suffix array into the second. Once the interleave is calculated, the merged BWT is trivially assembled. The algorithm requires an auxiliary index (such as the FM-index) to support searching. As discussed in Section 2.3, the memory overhead of an unsampled FM-index is $O(N)$ where N is length of the BWT. While sub-sampling of the FM-index reduces this memory cost, it will also impact search performance. Moreover, this algorithm is ill-suited to multiple datasets (more than two) to merge. In this case, the algorithm performs multiple merges until only one dataset remains.

Our algorithm merges two or more BWTs directly without any search index or the need to reconstitute any string or suffix of the input BWTs. The only auxiliary data structures required are two interleave arrays which identify the input source of each symbol in the final result, so the only auxiliary data structures used by the algorithm are stored as part of the result. The merging is accomplished by permuting the interleaves of the input BWTs, which is proven to be equivalent to a radix sort [Knuth, 1998] over the suffixes of the string collections.

3.2 Approach

Similar to previous approaches [Bauer et al., 2011, 2013, Ferragina et al., 2012, Sirén, 2009], our approach also takes advantage of the fact that a BWT implicitly represents a suffix array (see Section 2.2). Intuitively, when two or more BWTs are merged, the relative orders of the suffixes within each original BWT do not change because they must maintain a stable sort order in the output [Sirén, 2009, Ferragina et al., 2012]. This means that the merged BWT can be defined as an interleaving of the original input BWTs such that the result implicitly represents a combined, fully sorted suffix array [Sirén, 2009].

Our merging algorithm is an iterative method that converges to the correct interleave of two or more BWTs [Holt and McMillan, 2014a]. It starts by assuming the interleave is just a concatenation of BWTs. Then, in each iteration, it adjusts the current interleave during a pass through the BWT inputs. Each iteration acts as a single pass of an implicit radix-sort [Knuth, 1998] that corrects the interleave. After the first iteration, the first symbols of each suffix ('A', 'C', 'G', etc.) are grouped. After the second iteration, all identical di-mer suffixes ('AA', 'AC', 'AG', etc.) are grouped. Eventually, the interleave converges to the correct solution, which is detected by two consecutive iterations resulting in identical interleaves (in other words, the interleave did not change). Finally, the two BWTs are merged based on the converged interleave and the result is stored as a single BWT. This method can be extended to merge any number of BWTs simultaneously in $O(LCS * N)$ time where LCS is the longest common substring between any two BWTs and N is the total combined length of the merged output.

3.3 The merge algorithm

3.3.1 Merge variables

Let two input BWTs, B_0 and B_1, be defined over a finite alphabet, Σ, with lexicographically ordered symbols $\$ < c_1 < c_2 < ... < c_\sigma$. A string is defined as a series of k symbols from this alphabet terminated with a special end-of-string symbol, '\$'. Let S be a collection of such strings, $S = \{s_1, s_2, ..., s_m\}$. All BWTs are assumed to be in the form described in Section 2.4 such that any string can be reconstituted by prepending the predecessors repeatedly until the starting index is reached (each string forms a cycle in the BWT). In a single pass through all input BWTs, the total number of occurrences of each symbol is counted and used to calculate a list of offsets into the final BWT indicating the first suffix in the output starting with each symbol. These counts and offsets are a tiny subset of the FM-index (see Section 2.3), but for the unknown output BWT, rather than the known input BWTs.

Overall, the goal of the algorithm is to construct an interleave of the two input BWTs such that the implicit, combined suffix array is in sorted order. We call this interleave the "correct" interleave because it defines how to merge the BWTs such that no BWT properties are violated in the merged output. It begins by constructing an initial interleave of the input BWTs that is simply a concatenation of the inputs. Then, a series of iterations are performed on the interleave. Each iteration acts like one pass of a most-significant-symbol radix sort over the implicit suffix array represented by the interleaved BWTs. After one iteration, the interleave will be such that all suffixes are lexicographically sorted by their first symbol. After two iterations, the interleave will be such that all suffixes are lexicographically sorted by their first two symbols. The third is the first three symbols. These iterations will continue until there is no change in the interleave, indicating that the implicit suffix array has converged to a correct interleave.

Given two BWTs, $B_0 = \text{bwt}(S_0)$ and $B_1 = \text{bwt}(S_1)$, of length m and n respectively, note that the target result, $B_2 = \text{bwt}(\{S_0, S_1\})$, can be trivially constructed if the interleave of B_0 and B_1 is known. As such, the primary goal of our proposed method is to calculate this interleave. Let I be auxiliary array called the interleave that is a series of zeroes and ones of length $(m + n)$ such that a zero corresponds to a symbol in B_2 originating from B_0 and a one corresponds to a symbol originating from B_1. There are exactly m zeroes and exactly n ones in I. As the merge algorithm progresses, this interleave will be corrected until it converges to the final interleave. This I array is similar to the interleave array described in Sirén [2009].

Let totals array T be a list of numbers such that for each symbol c in Σ, $T[c] = \text{count}(c, B_0) + \text{count}(c, B_1)$. In short, T is a total combined count of each symbol in the two BWTs. Additionally, let offsets array O be defined for each symbol c such that $O[c]$ is the position of the first suffix in the merged BWT, B_2, that starts with c, which can be calculated by adding the T values for all symbols lexicographically before c. Note that this is equivalent to the offsets component of the FM-index for the final merged BWT (see Section 2.3).

3.3.2 Iterations of the merge

The main iteration function of the merge algorithm, `mergeIter(...)`, is defined in Algorithm 3. Repeated calls to this procedure converge to the correct interleaving of B_0 and B_1 resulting in B_2. To prove this claim, we first show that a correct interleave will not change as a result of this function.

Lemma 3.3.1. *Given a correct interleave I, of two BWTs, B_0 and B_1, into a single BWT B_2, and the offsets array O, for each symbol $c \in \Sigma$ into B_2, then mergeIter(I, B_0, B_1, O) will return the same interleave, I, as was passed into it.*

Data: I - the current interleave of B_0 and B_1; B_0 - the first BWT to merge; B_1 - the second BWT to merge; O - for each $c \in \Sigma$, $O[c]$ is the position of the first suffix starting with c in the merged BWT

Result: $INext$ - the new interleave

```
Function mergeIter($I$, $B_0$, $B_1$, $O$)
    // initial conditions
    $INext \leftarrow [null]*\mathrm{len}(I)$;
    $currentPos0 \leftarrow 0$;
    $currentPos1 \leftarrow 0$;
    $tempIndex \leftarrow O$;
    // iterate through each bit value in $I$
    forall the $b \in I$ do
        // $b$ is a bit representing the input BWT
        if $b = 0$ then
            $c \leftarrow B_0[currentPos0]$;
            $currentPos0 \leftarrow currentPos0 + 1$;
        else
            $c \leftarrow B_1[currentPos1]$;
            $currentPos1 \leftarrow currentPos1 + 1$;
        end
        // copy $b$ into the next position for symbol $c$
        $INextPos \leftarrow tempIndex[c]$;
        $INext[INextPos] \leftarrow b$;
        // update the $tempIndex$ to match the FM-index
        $tempIndex[c] \leftarrow tempIndex[c] + 1$;
    end
    return $INext$;
```

Algorithm 3: Each call to this function performs one iteration of the merge algorithm. Additionally, this function can be used to verify that an interleave has converged to the correct final interleave.

Proof: This lemma follows from the properties of a BWT. The lemma assumes that the ordering of interleave I results in a correct BWT B_2, for a collection of strings $S_2 = \{S_0, S_1\}$. In the initial condition, the current positions are both 0 and the *tempIndex* value corresponds to the offsets into the merged BWT array. First, it's easily noted that after each iteration, $i = 1..(m + n)$, $currentPos0 + currentPos1 = i$. This is because as i increments, one of the *currentPos* values is also incremented at the end of the loop. Additionally, if given an FM-index F and offsets O for the BWT represented by I, note that $tempIndex = O + F[i]$ after each iteration. Before any iterations, the *tempIndex* is set to O and $F[0]$ is all zeros, so the initial condition holds. With each iteration i, a symbol c is determined by the interleave and used to identify the index $tempIndex[c]$ to increment such that the condition $tempIndex = O + F[i]$ holds at each iteration. Finally, after each iteration, one value of *INext* is changed corresponding to the current $tempIndex[c]$. Since $tempIndex[c] = O + F[i]$, this implies that the LF-mapping property (see Section 2.3) of the FM-index also holds, so each iteration is effectively setting $INext[tempIndex[c]] = I[tempIndex[c]]$. After doing this for all values b in I, the algorithm has set every value in *INext* to its corresponding value in I. Alternatively, if there exists a position, j, such that $INext[j] \neq I[j]$, then the original assumption that I is a correct interleave must be false because there is at least one position where the implied FM-index causes a suffix originating from B_0 to have a predecessor symbol originating from B_1 (or vice-versa) which cannot happen in a correct BWT. ■

The point of Lemma 3.3.1 is that given an interleave I_i, its covergence can be verified by executing `mergeIter(...)` once and comparing the resulting interleave I_{i+1} with the original input interleave I_i. If $I_i = I_{i+1}$, then the interleave has converged on the correct solution.

3.3.3 Convergence in the merge algorithm

The function, `mergeIter(...)`, is also a convenient subroutine that performs one iteration in the merge of two BWTs. To apply this function for a full merge requires simple setup and an outer-loop to test for convergence. The pseudocode for the `bwtMerge(...)` operation is presented in Algorithm 4.

As described in Section 2.2, the BWT implicitly represents a sorted suffix array. The BWT can also generate partial suffixes or the first i symbols (i-mer prefix) of the suffix. We will refer to these as i-suffixes. Given a BWT, the symbols of the BWT can be sorted using a radix sort to recover all 1-suffixes in lexicographic order. Then, if the BWT is prepended to the sorted 1-suffixes and

```
Data: $B_0$ - the first BWT to merge; $B_1$ - the second BWT to merge; $\Sigma$ - lexicographically
      ordered valid symbols in the BWTs
Result: The correct interleave of $B_0$ and $B_1$ to form a new BWT
Function bwtMerge($B_0$, $B_1$, $\Sigma$)
    // initial pass to calculate offsets $O$
1   $off \leftarrow 0$;
2   forall the $c \in \Sigma$ do
3       $T[c] \leftarrow$count$(c, B_0)$+count$(c, B_1)$;
4       $O[c] \leftarrow off$;
5       $off \leftarrow off + T[c]$;
    end
    // initialize the $ret$ array to zeroes followed by ones
6   $I \leftarrow null$;
7   $ret \leftarrow [0]*\text{len}(B_0) + [1]*\text{len}(B_1)$;
8   while $I \neq ret$ do
        // copy the old interleave and re-iterate
9       $I \leftarrow ret$;
10      $ret \leftarrow$ mergeIter$(I, B_0, B_1, O)$;
    end
    return $ret$;
```

Algorithm 4: This function will calculate the interleave for merging two BWTs into a single output BWT. The algorithm repeatedly calls `mergeIter(...)` until it detects convergence in the interleave. Once converged, the resulting BWT can be trivially created using the output interleave.

sorted again using only the prepended BWT characters, all 2-suffixes in the BWT are recovered in lexicographic ordering. If this process is repeated for i iterations, all i-suffixes in the BWT can be recovered. This is equivalent to doing a least significant symbol radix sort of all i-suffixes in the suffix array. Since the algorithm is sorting the prefixes of the suffixes by increasing the prefix length i, it is really performing a most-significant-symbol radix sort on the full suffixes in the suffix array.

Each iteration of the `while` loop in `bwtMerge(...)` is equivalent to performing an iteration of the most-significant-symbol radix sort on the full suffixes. The I array indicates the current interleave of symbols at the start of an iteration. Then, the interleave bits are copied into the next available location for their corresponding symbol using the FM-index values stored in *tempIndex*. With each iteration in the loop, the sorting of suffixes is extended by one symbol until I converges to a correct interleave. Note that the actual suffixes are never explicitly reconstructed or stored. In the example executions in Tables 3.1 and 3.2, the suffixes are shown after each iteration for illustration only. The final merged BWT is created trivially using the interleave from `bwtMerge(...)`.

Lemma 3.3.2. *Given an initial interleaving, I_0, of two BWTs, B_0 and B_1, such that all zeroes* *come before all ones, after i executions of `mergeIter(...)`, all corresponding suffixes in the suffix array implied by I_i are stably sorted up to their first i symbols.*

Proof: Using induction, consider the initial condition, $i = 0$. In this base case, all 0-suffixes are identical (empty string) and the ordering is all zeros followed by all ones. Now, consider iteration $(i + 1)$ where the interleaving, I_i, is a stable sort of the first i symbols of the suffixes of the corresponding BWTs. In the next iteration, the algorithm performs a pass over I_i retrieving the corresponding predecessor symbol for each bit in I_i. If two suffixes have different start symbols, then those suffixes are automatically ordered correctly because each will be placed into the appropriate bin for that symbol. As a property of the radix sort, all $(i + 1)$-suffixes starting with the same symbol, c, are placed sequentially in the output in lexicographical ordering. Given that the i-suffixes are already stably sorted, if the symbol c is found at two indices, x and y where $x < y$, then the corresponding $(i + 1)$-suffixes must be of the form cX and cY where X is an i-suffix that lexicographically precedes the other i-suffix Y. This implies that the corresponding $(i + 1)$-suffixes are also in sorted order in the new interleave I_{i+1}. ■

Theorem 3.3.3. *Given that the longest common substring of two string collections stored in separate BWTs, B_0 and B_1, is of length k, and that the initial interleave, I_0, is a single series of zeros followed by ones, then the `bwtMerge(...)` algorithm will converge in $k + 1$ or fewer iterations. Additionally, this convergence can be detected by iterating until interleave I_i stops changing.*

Proof: Using Lemma 3.3.2, it's known that after $k + 1$ iterations, all $(k + 1)$-suffixes will be lexicographically sorted. If the longest common substring (LCS) is of length k, then it follows that after $k + 1$ iterations the suffixes will be sorted up to their first $(k + 1)$ symbols. This means that further iterations will not change the sort order of the suffixes and that the interleave has converged. Additionally, we know from Lemma 3.3.1 that convergence can be detected (i.e. the interleave can be verified as correct) through an additional iteration. ■

Given that the combined size of the two input BWTs is $N = n + m$, this algorithm uses $O(N)$ bits of additional memory due to the creation of two interleave arrays I and $INext$. Assuming a fixed alphabet, all other variables are constant sized. For practical purposes, all large arrays (B_0, B_1) are actually stored on disk due to their size.

The algorithm detects convergence after at most $(LCS + 1)$ iterations. If all strings are of length

L, the algorithm will converge after at most $(L + 1)$ iterations. Additionally, this algorithm does allow for variable length strings. In fact, it is completely unaware of the string lengths present in either BWT. However, the one caveat to Theorem 3.3.3 is the determination of LCS when the input strings are of variable length. In some instances, there are string collections that can result in iterations up to $(2 * L + 1)$ iterations due to the contents of the strings. The reason for this is the cyclic nature of strings in the BWT and the similarity between two BWTs. Consider two strings 'AA\$' and 'AAA\$'. At first glance, the $LCS = 3$, 'AA\$'. However, because of the cyclic nature of the strings when represented in a BWT, the true $LCS = 5$, 'AA\$AA', which can be generated by starting from the first symbol in the first string and the second symbol in the second string and cycling back through when there are no more symbols. For the real-data experiments presented in this chapter, all strings were of identical length, so this caveat wasn't an issue even in the presence of identical strings in each input BWT.

An example of an entire execution of `bwtMerge(...)` is shown in Table 3.1. For this basic example, only two single-string BWTs were used in the merge. Iteration 0 represents the initial condition and all subsequent iterations are the result from executing the `mergeIter(...)` function once. After three iterations, the algorithm has converged and verified the convergence.

3.3.4 Generalizing the merge algorithm for more than two BWTs

Until now, the algorithm has only been discussed and demonstrated using exactly two input BWTs and a bit-vector to distinguish the originating BWT for each position in the merged BWT. A basic extension of this technique repeatedly merges f input BWTs according to any binary tree resulting in a single merged BWT at the tree's root. This requires $f - 1$ merges. However, the faster way is to extend the I array to multiple bits allowing for multiple BWTs to be merged simultaneously. For example, a byte array supports the merging of up to 256 multi-string BWTs simultaneously. Given f input BWTs, the above proofs can be extended by starting with an initial I_0 consisted of a series of 0's, series of 1's, ..., series of $f - 1$'s as the initial condition and an initial offsets calculated in a pass over all f inputs. Additionally, variables corresponding to a specific BWT (such as B_0, $currentPos1$, etc.) can be condensed into arrays of length f that can be indexed by the interleave value, b, at each position. Then, the algorithm can iterate as before until convergence is reached. This algorithm extension allows for an arbitrary number of multi-string BWTs to be merged simultaneously by using $O(N * log(f))$ bits of storage for interleave I_i. An example execution

initial state			iteration 1			iteration 2			iteration 3		
I	S	B	I	S	B	I	S	B	I	S	B
0		A	0	$	A	0	$A	A	0	$AC	**A**
0		C	1	$	A	1	$C	A	1	$CA	**A**
0		$	0	A	C	0	A$	C	0	A$A	**C**
0		C	0	A	$	1	A$	A	1	A$C	**A**
0		A	1	A	A	1	AA	A	1	AA$	**A**
1		A	1	A	A	1	AA	C	1	AAA	**C**
1		A	1	A	C	0	AC	$	0	ACC	**$**
1		A	0	C	C	0	CA	C	0	CA$	**C**
1		C	0	C	A	1	CA	$	1	CAA	**$**
1		$	1	C	$	0	CC	A	0	CCA	**A**

Table 3.1: Two-way merge example. The above table shows the state after each iteration, i, for merging two BWTs each containing one string, "ACCA$" and "CAAA$" respectively. Their respective starting BWT strings are "AC$CA" and "AAAC$" which are shown in their concatenated initial state in the left most columns. At each iteration, the table shows the interleave, I, the i-suffixes, S, and what the merged BWT, B, is with that interleave. After the first iteration, there are three bins of zeroes followed by ones representing the three i-suffixes of length one: '$', 'A', and 'C'. The second iteration puts all 2-suffixes in their correct bins. After iteration 2, the algorithm has converged to the correct solution early. Iteration 3 will detect no change in the interleave, and the merged BWT in bold is stored as the final solution. Note that in each iteration i, the i-suffixes are in sorted order and all i-suffix bins containing both zeroes and ones have all zeros before all ones.

of this extension is shown in Table 3.2.

The merged BWT is a complete interleave of the multiple BWTs into a single dataset. Since distinguishing the source of a particular BWT symbol may be important, the final interleave array I_i can be stored as an auxiliary component to the merged BWT. Alternatively, if only the source of a particular string (as opposed to each individual suffix) is required, the length of the I array can be truncated as described in Section 3.4.3. This allows for a merged BWT to differentiate the original source BWT for a particular string in later analyses.

3.3.5 Asymptotic performance

We summarize the algorithmic complexities for merging two BWTs in Table 3.3. Notice that the performance of this algorithm is not directly affected by string length or the number of strings in either BWT. For a constant N, increasing string length and decreasing the number of strings will not affect run time unless there is also an increase in the LCS between the two BWTs. Furthermore, the memory and disk requirements for this algorithm are relatively low. In memory, the algorithm requires only $O(N)$ bits for each interleave. Assuming all BWTs are uncompressed, the algorithm requires $O(LCS * N * log(\sigma))$ bits read from disk. Since the final merged BWT is only written once,

initial state			iteration 1			iteration 2			iteration 3			iteration 4		
I	S	B	I	S	B	I	S	B	I	S	B	I	S	B
0		C	0	$	C	0	$A	C	0	$AC	C	0	$ACA	**C**
0		C	1	$	C	2	$A	A	2	$AC	A	2	$ACC	**A**
0		$	2	$	A	1	$C	C	1	$CA	C	1	$CAA	**C**
0		A	0	A	C	2	A$	C	2	A$A	C	2	A$AC	**C**
0		A	0	A	$	1	AA	C	1	AAC	C	1	AAC$	**C**
1		C	1	A	C	0	AC	C	0	AC$	C	0	AC$A	**C**
1		C	1	A	A	0	AC	$	1	AC$	A	1	AC$C	**A**
1		A	2	A	C	1	AC	A	0	ACA	$	0	ACAC	**$**
1		A	2	A	$	2	AC	$	2	ACC	$	2	ACCA	**$**
1		$	0	C	A	0	C$	A	0	C$A	A	0	C$AC	**A**
2		A	0	C	A	1	C$	A	1	C$C	A	1	C$CA	**A**
2		C	1	C	A	0	CA	A	2	CA$	C	2	CA$A	**C**
2		$	1	C	$	1	CA	$	1	CAA	$	1	CAAC	**$**
2		C	2	C	C	2	CA	C	0	CAC	A	0	CAC$	**A**
2		A	2	C	A	2	CC	A	2	CCA	A	2	CCA$	**A**

Table 3.2: Three way merge example. The above table shows the state after each iteration, i, for merging three BWTs, each containing one string: “ACAC$", “CAAC$", and “ACCA$" respectively. Their respective starting BWT strings are “CC$AA", “CCAA$", and “AC$CA". At each iteration, the table shows the interleave, I, the i-suffix, S, and what the merged BWT, B, is with that interleave. Each iteration corrects the suffixes by one symbol which can be seen in the S columns of each iteration. After three iterations, the ordering is correct. Iteration 4 detects convergence of the interleave and the resulting BWT is shown in bold on the far right.

CPU time	$O(LCS * N)$
Max Memory bits	$O(N)$ - Two Interleaves
Disk I/O bits	$O(LCS * N * log(\sigma))$ - Input BWTs $O(N * log(\sigma))$ - Output Merged BWT $O(N)$ - Output Final Interleave

Table 3.3: Merge asymptotic performance. The asymptotic run time, maximum memory use, and disk I/O for the merge algorithm of two BWTs. LCS is the longest common substring between the two input BWTs, N is the total number of symbols from both inputs, and σ is the number of symbols in the alphabet (including ‘$’).

it only writes $O(N * log(\sigma))$ bits to disk.

3.4 Results

3.4.1 Merge times

The asymptotic runtime for the merge algorithm is $O(LCS * N)$. We developed a Cython implementation of the merge algorithm to test the actual performance on genomic datasets. First, we performed an experiment using real reads from mouse data provided by the Sanger Institute[1].

We chose two samples, WSB/EiJ and CAST/EiJ, to use for the merge. From these samples, we extracted all reads from each dataset that were aligned to the mitochondria (the reason for this choice becomes apparent in Section 3.4.2). The annotated mouse mitochondria is approximately 16,299 base pairs long. Between the two samples, there were over 1.6 million reads where each read was 100 basepairs long (over 10,000x coverage combined). We sampled reads from each set proportionally, created separate BWTs (one for WSB/EiJ, one for CAST/EiJ), and performed a merge of the two BWTs. The results of the merge execution times with respect to the total number of input sequences (reads) are shown in Figure 3.1. In these sampled datasets, the LCS term of the expected run-time is essentially constant at $LCS = 101$, the read length. As a result, there is a linear relationship between the total number of sequences that are merged and the total wall clock time to perform the merge.

3.4.2 Compression

One motivation for merging BWTs is to improve the compression. As discussed in Section 2.2, the BWT was proposed as a method for data compression because it tends to create long runs of repeated symbols that can be used by many compression schemes [Burrows and Wheeler, 1994]. The reason long runs form is that the BWT has a tendency to cluster similar suffixes together, and there is an expectation that those patterns have the same predecessor symbol (the value actually stored in the BWT). Therefore, if the two input BWTs to a merge contain similar substrings, then the compression should increase due to the ability to form longer runs. The redundancy of genomic sequencing data results from two factors: the datasets themselves are individually over-sampled and the genomes of distinct organisms tend to share genomic features reflecting a common origin.

[1]ftp://ftp-mouse.sanger.ac.uk/REL-1302-BAM-GRCm38

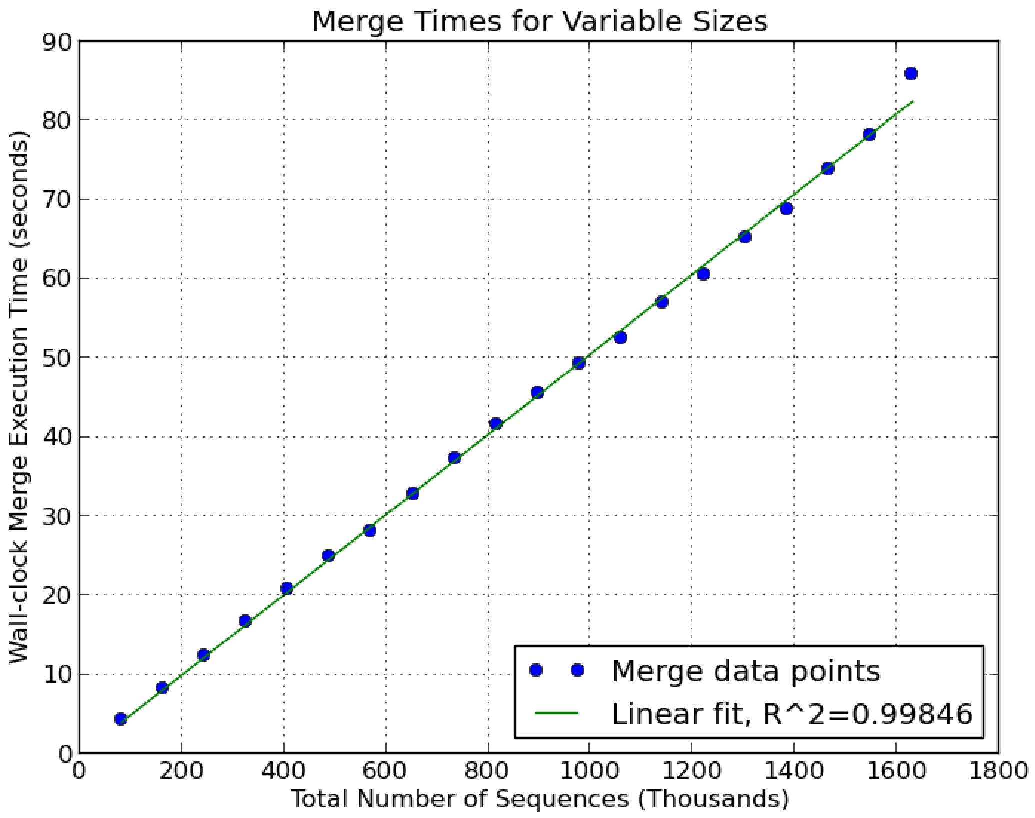


Figure 3.1: Merge execution times. This plot shows the relationship between the total size of the input BWTs being merged and the wall-clock time to execute the merge. Each data point is a merge between two BWTs (CAST/EiJ and WSB/EiJ) where each BWT contains a randomly sampled collection of read sequences that were aligned to the mouse mitochondria. In general, the wall-clock execution time follows a linear trend with the total size of the two inputs.

We define an average run-length (RL) metric in order to measure compressibility of the BWT. RL is defined as $\frac{N}{R}$ where N is the total number of symbols in the BWT and R is the number of contiguous symbol runs in that BWT (including runs of length 1). This metric basically represents the compression potential of a BWT where it is better to have a larger average run length. This metric emphasizes the impact of merging on compressibility rather than a particular subsequent compression method (ex. run-length encoding, move-to-front transforms, variable-length coding, Lempel-Ziv [Ziv and Lempel, 1978], etc.).

To demonstrate compressibility, we used the high coverage mitochondria data described in Section 3.4.1. Each dataset was sampled at lower coverages, merged into a single BWT, and then analyzed to identify the impact on average RL. The results of this experiment are shown in Figure 3.2. In general, there is a faster growth in average RL at lower coverages that becomes more constant at higher coverages.

We also performed three other merge experiments using full RNA-seq datasets from Crowley et al. [2015]. The first combined two mouse biological replicates, which were both WSB/EiJ inbred samples. The second was performed on two samples from diverse mouse subspecies CAST/EiJ inbred and PWK/PhJ inbred mouse samples. The final experiment merged eight biological replicates, all of type CAST/EiJ. In all three experiments, the strings were 100 basepair paired-end reads.

Each BWT file was analyzed both separately and as a merged BWT file as shown in Table 3.4. In all three scenarios, the compressibility was improved. The first and third experiments demonstrated that merging biological replicates leads to increased compressibility primarily due to an increase in coverage (the genomes are expected to be the same). The second experiment demonstrated that even with divergent samples of the same species, there is still enough shared sequence to improve the overall compressibility.

Given that average RL is defined as $\frac{N}{R}$, the total number of bases, N, before and after a merge is constant. If average RL is increasing, then it must be the case that the number of runs, R, is decreasing. The main reason for this is the combining of pre-existing runs as datasets are merged together. To show this, the distributions of run-lengths for the eight-way merge experiment is shown in Figure 3.3 both before and after the merge. In this plot, there are fewer “short" runs and more “long" runs in the merged file, indicating an increase in average run length.

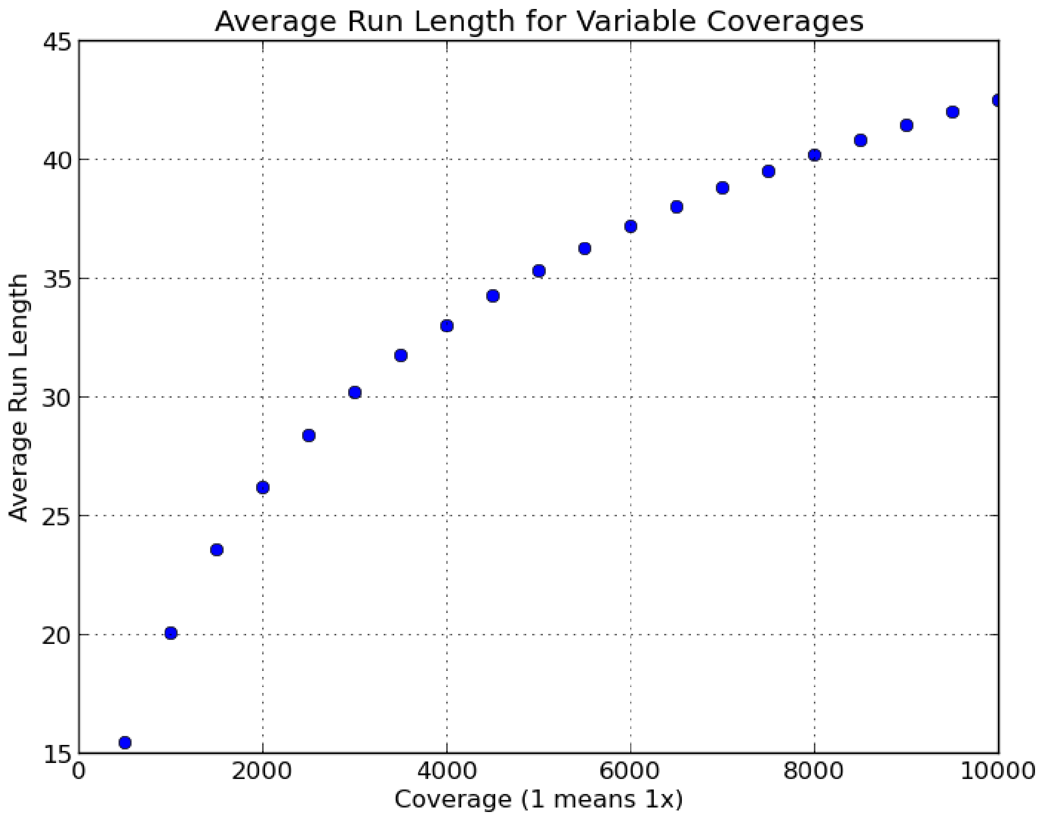


Figure 3.2: Average run length by coverage. This plot shows the average length of runs in the merged BWT at different levels of coverage. Note that as the coverage is increasing, the average run-length increases with it. This effectively means greater compressibility with respect to the original data size. Note that there is faster growth at lower coverages before it eventually settles into a more linear growth at higher coverages.

BWT(s)	Symbols	RLE Entries	Average RL
HH1361 individual	$6.68 * 10^9$	$1.13 * 10^9$	5.902
HH1380 individual	$6.32 * 10^9$	$.926 * 10^9$	6.825
HH1361 + HH1380	$13.00 * 10^9$	$2.05 * 10^9$	6.317
HH's Merged	$13.00 * 10^9$	$1.83 * 10^9$	**7.086**
FF0683 individual	$8.94 * 10^9$	$1.11 * 10^9$	8.000
GG1240 individual	$14.20 * 10^9$	$1.36 * 10^9$	10.401
FF0683 + GG1240	$23.14 * 10^9$	$2.48 * 10^9$	9.320
FF merged with GG	$23.14 * 10^9$	$2.20 * 10^9$	**10.475**
FF0683 individual	$8.94 * 10^9$	$1.11 * 10^9$	8.000
FF0684 individual	$7.97 * 10^9$	$1.48 * 10^9$	5.361
FF0685 individual	$13.11 * 10^9$	$1.47 * 10^9$	8.890
FF0727 individual	$7.98 * 10^9$	$1.58 * 10^9$	5.019
FF0728 individual	$13.64 * 10^9$	$1.65 * 10^9$	8.267
FF0754 individual	$18.36 * 10^9$	$2.04 * 10^9$	8.957
FF0758 individual	$13.13 * 10^9$	$1.92 * 10^9$	6.816
FF6136 individual	$10.34 * 10^9$	$2.00 * 10^9$	5.146
FF total individuals	$93.46 * 10^9$	$13.3 * 10^9$	7.026
FF's Merged	$93.46 * 10^9$	$9.47 * 10^9$	**9.865**

Table 3.4: Compressibility of merges. This table shows the average run-length (RL) metrics for RNA samples before and after merging. The datasets are all inbred mouse datasets of type CAST/EiJ (FF), PWK/PhJ (GG), or WSB/EiJ (HH). Experiments are grouped into blocks. Each experiment compares the merged results (in bold) to the totals for separate files. Note that in all experiments there is a decrease in the number of run length entries and increase in average run-length when moving from individual files to a single merged file indicating that the merged version is more compressible than separate files.

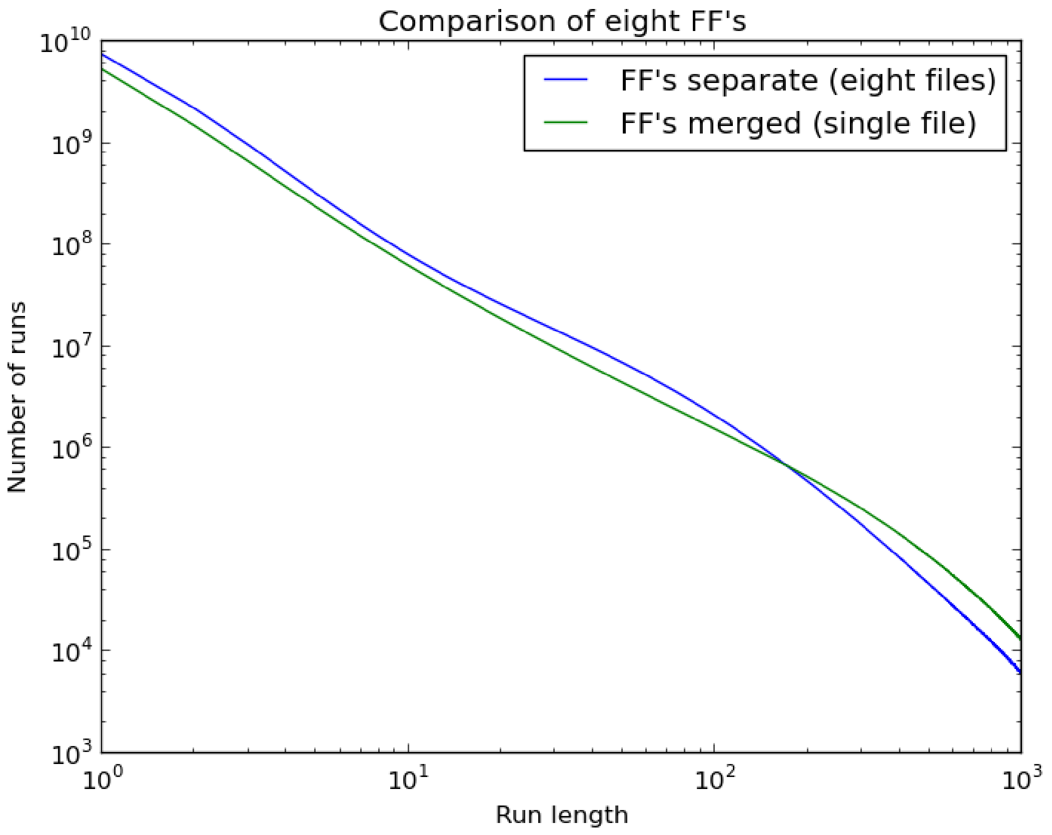


Figure 3.3: Run-length distribution in a merge. This plot shows the distribution of run-lengths for eight separate CAST/EiJ (FF) RNA-seq BWT files (blue) and a single merged BWT file containing all eight samples (green) from the third experiment in Table 3.4. Note that for the merged file, there are more runs of longer length and fewer runs of shorter length. This is because the merged BWT has brought the similar components of each BWT together leading to longer runs.

3.4.3 Interleave storage

Thus far, we have ignored the storage requirements for the interleave vector. The interleave can be naïvely stored as the final interleave array I_i from the merging algorithm which requires $O(N * log_2(f))$ bits of space, where N is the number of symbols and f the number of input BWTs.

After the merge finishes, the portion of the interleave array corresponding to suffixes starting with the '\$' is the only thing necessary to associate every read with its origin. In other words, you only need one interleave value per string. A particular symbol's origin can be recovered by tracing backwards through the BWT until the '\$' symbol is found, so the tradeoff to reduce the I size is run-time speed. Using this smaller index means that the space for the I array will be $O(R * log_2(f))$ where R is the number of strings in the merged BWT. Finding the origin of a symbol will take $O(L)$ time instead of $O(1)$ where L is the length of the string. While only one value per read is required, this is still a relatively large array in practice, and it is unknown whether it is well-suited to any particular compression scheme.

3.5 Potential applications of merged BWTs

One motivation for merging BWTs is to improve upon the compression achieved by separate BWTs. Depending on the types of data being merged, the merged BWT and its associated interleave are also useful for asking certain biological questions. The most basic benefit is performing a single query in place of multiple queries to separate datasets. For example, the comparison algorithm proposed by Cox et al. [2012b] performs two queries to separate BWTs to find splice junctions. In their method, one dataset contained DNA and the other contained RNA for the same sample. Since the sequences in each dataset are naturally similar, the combined version should compress well. Furthermore, as separate files, the algorithm needs $O(f * k)$ time to search f BWTs for a given k-mer which is reduced to $O(k)$ when a merged BWT is used instead. In this regard, the merging provides a speedup in downstream analyses in addition to the compression.

BWTs in general can also be applied to *de novo* sequence assembly. In fact, some existing assemblers already use a BWT as their underlying indexing data structure [Simpson and Durbin, 2010, 2012]. Several *de novo* assembly techniques currently use the De Bruijn graph as the underlying data structure [Pevzner et al., 2001, Zerbino and Birney, 2008, Butler et al., 2008, Simpson et al., 2009, Salikhov et al., 2014]. BWTs can be used as efficient and compact De Bruijn graph representations with enhanced functionality. The presence, count, and sample origin of individual k-mers is determined using the BWT's FM-index. The k-mer size can be varied without any modifications to the BWT, and the surrounding context (i.e. the containing read fragment) of each k-mer is accessible. A De Bruijn graph constructed from a merged BWT for a species would include separate paths for haplotypes, thus representing a pan genome of the merged population [Rasko et al., 2008].

Merging datasets into a single BWT constructed from biological replicates can be used to increase statistical power in *de novo* assembly and other analyses as well. Such datasets can also be used to examine the consistency between replicates as well as the variants between diverse samples without the overhead of aligning. Robasky et al. [2014] discuss the advantages of using replicates to help reduce errors and biases in experiments. With the eight-way merge of biological replicates from Table 3.4, the merged BWT and the corresponding interleave can be used to calculate the abundance and variance of a given k-mer for all replicates simultaneously. Robasky et al. [2014] also mention how using replicates from different platforms can be useful to reduce bias. In addition to this benefit, combining different datasets in *de novo* assembly is useful for extending contigs. For example, a BWT consisting of short reads (such as Illumina) could be merged with long reads (such as PacBio) to produce a merged BWT with the ability to query both datasets.

Alignment is another common use for reads. Given a reference genome, a BWT can be used to search for evidence of genome subsequences in the reads. In this situation, the counts from the query would be similar to pileup heights from an alignment. Regions with lower-than-expected counts can be re-examined by selecting reads from nearby regions and generating a consensus, and thereby detecting variants including SNPs and indels, as if we were aligning the genome to the reads instead of the reads to the genome. Additionally, there's potential for algorithms that merge BWTs from raw sequencing files with a BWT of the reference genome. Ideally, this merged BWT would cluster the read suffixes near genome suffixes and allow the interleave to be a starting point for alignment.

Finally, this algorithm can be used as the basis for a multi-string BWT construction algorithm. Given a string collection, small BWTs can be constructed in parallel and then merged together to form the final BWT. The next chapter discusses one such implementation of a merge-based construction algorithm.

CHAPTER 4

MULTI-STRING BWT CONSTRUCTION VIA MERGING

In Chapter 2, we introduced methods for constructing the BWT from a string collection or suffix array. However, these methods are not practical as they involve an expensive sorting of all suffixes of all strings. There are more efficient methods for BWT construction [Mantaci et al., 2005, Sirén, 2009, Bauer et al., 2011, Cox et al., 2012a, Bauer et al., 2013, Li, 2014], but their performance is subject to various constraints. In particular, the first construction algorithms [Bauer et al., 2011, 2013] focused on building the BWT for short, uniform length strings such as the reads from an Illumina sequencer. As a result, datasets with long or variable length strings such as the reads from a long-read sequencer (i.e. PacBio or nanopore) were either incompatible with the BWT construction method or required significant computation time.

In this chapter, we describe an alternative BWT construction algorithm [Holt and McMillan, 2014b] that is based on the BWT merging algorithm described in Chapter 3 [Holt and McMillan, 2014a]. The algorithm's performance is not directly affected by read length and is one of the first to build BWTs for long-read datasets. For a collection of m reads totaling N bases, this algorithm runs in $O(N * LCP_{avg} * \log(m))$ time where LCP_{avg} is the average Longest Common Prefix (LCP) for the suffixes in the implicit suffix array. Given an alphabet with σ symbols, the I/O volume for this algorithm is $O(N * LCP_{avg} * \log(m) * \log(\sigma))$ bits.

4.1 Related work

Many researchers have focused on how to accelerate the construction of BWTs for genomic datasets [Mantaci et al., 2005, Sirén, 2009, Bauer et al., 2011, Cox et al., 2012a, Bauer et al., 2013, Li, 2014]. Bauer et al. [2013] provide a comparison of several methods including their own: BCR, BCRext, and BCRext++. While there are slight differences between the three BCR algorithms, they all leverage the same general idea. Given a string collection with m strings (i.e. reads in a sequencing dataset) that are all k symbols long, one can imagine stacking the strings such that each string is a row and each column is a group of symbols that are all at the same index in the strings. The BCR algorithms then insert symbols in a “column-wise" manner [Bauer et al., 2013]. In other words, the algorithm inserts all symbols from the string collection at column index $(k-1)$, at column index $(k-2)$, ... at column index 1, and finally at column index 0. In total, it performs k insertion iterations and inserts exactly m symbols during each iteration for $O(mk)$ steps to calculate the BWT.

This algorithm works well for uniform, short-read datasets, like those generated by Illumina sequencers. However, for an alphabet, Σ, containing σ unique symbols, the I/O volume of the BCR algorithms is $O(mk^2 * \log(\sigma))$ meaning that “the length of the longest string in the collection dominates the I/O complexity" [Bauer et al., 2013]. In practice, they reported a 29-69% increase in run-time when the read lengths were doubled, but total number of bases stayed constant [Bauer et al., 2013]. In short, the BCR algorithms do not scale well as the length of reads grows, a property that is problematic for datasets that consist of long strings. For example, PacBio reads can be thousands of base pairs long and significantly vary in length [Eid et al., 2009]. Ropebwt2 is a recent adaptation of the BCR algorithms that uses B+ trees as the underlying storage container instead of arrays [Li, 2014]. This helps alleviate some of the I/O tradeoffs by removing the reallocation of arrays at the cost of $O(\log(N))$ insertions for a tree with N nodes. In practice, ropebwt2 is much faster than other implementations of BCR [Li, 2014]. However, like the other BCR algorithms, the performance of ropebwt2 decays as the length of reads in the dataset increases.

4.2 Approach

The main idea of the BWT merge algorithm is to divide-and-conquer over the entire string collection. Given a string collection, a BWT of each individual string can be calculated using any single-string construction algorithm. Then, the BWTs are repeatedly merged using the merge algorithm described in Chapter 3 [Holt and McMillan, 2014a]. Eventually, only one BWT remains that contains all of the strings from the string collection.

In addition to applying the divide-and-conquer strategy, we modified the merge algorithm itself to reduce the amount of work required for any particular merge. In general, the BWT merge algorithm acts like a most significant symbol radix sort on the implicit suffix array (see Section 3.3.3). In a most significant symbol radix sort, local sub-regions converge on the correct ordering of strings, and these local convergences eventually lead to a global convergence of the radix sort. Similarly, the interleave array of the merge algorithm has areas of local convergence that appear prior to global convergence. By tracking intervals of local convergence, the merge algorithm can reduce the amount of work necessary to reach global convergence by ignoring the intervals that have already converged.

4.3 Construction via merging

This BWT merge construction algorithm is based on the BWT merge algorithm that was previously described in Chapter 3. The merge algorithm merges two or more BWTs into a single BWT through an implicit most-significant-symbol-first radix sort on the suffixes represented by the BWT [Holt and McMillan, 2014a]. Recall that it takes advantage of the fact that the suffixes contained within each input BWT are already sorted. The algorithm finds an interleave of the input BWTs such that all of the combined suffixes will be in sorted order in the output BWT.

One can naïvely adapt the merging algorithm to constructing large BWTs by first computing the single-string BWT for each read string individually. Then, use the merge algorithm in one step to combine all of the single string BWTs into a single BWT. The main issues that arise using this approach are that the length of the largest LCP bounds the computation time, and the storage requirement for the interleave vector ($O(N * \log(m))$ bits for m reads) can become prohibitively large. Instead, there are two major changes made to the merge algorithm to support the construction of BWTs.

4.3.1 Tracking buckets

The first main change is to add bookkeeping to the merge algorithm to track which intervals of the interleave vector have already converged to the correct solution. The original merge algorithm relies on global convergence of the interleave to complete the merge [Holt and McMillan, 2014a]. However, the main insight of the first change is that local convergence of the interleave leads to global convergence. Intuitively, this happens because the merge is implicitly calculating a most-significant-symbol-first radix sort on the BWT's implicit suffix array. Once a given suffix has been sorted past its longest common prefix (LCP), it is in the correct, final position in the merged BWT.

Theorem 4.3.1. *Given a suffix string, R, with longest common prefix, LCP_R, the suffix will be in its correct, final position in the merged BWT after $(LCP_R + 1)$ iterations.*

Proof: Lemma 3.3.2 states that after i iterations, all suffixes are sorted up to their first i symbols. After iteration $(LCP_R + 1)$, the given suffix R must be in its correct position in the interleave because it does not share $(LCP_R + 1)$ symbols with any other suffix. If R is not in its correct position, then it violates the original assumption by implying that LCP_R is not the longest common prefix of R. ■

The goal of this first change is to identify which regions of the interleave actually need to be processed (regions which have not converged) and only process those regions of the interleave in the next iteration. Note that after an iteration i, the interleave can be broken into buckets corresponding to i-mers. Then, only buckets which changed in the previous iteration need to be scanned in the next iteration. These i-mer buckets are conceptually similar to the LCP-intervals used to cluster regions of the BWT that have an LCP of length i [Ohlebusch et al., 2010].

Given an alphabet, Σ, with σ unique symbols, one property of the merge is that a bucket representing some i-mer, α, will sort all suffixes starting with $c\alpha$ for every $c \in \Sigma$. This means a given bucket can create up to σ new buckets corresponding to $(i+1)$-mers. While iterating through bucket α, the algorithm updates the interleave for each possible new bucket $c\alpha$. If this update causes the interleave or the symbols represented by that interleave to change, then the new bucket $c\alpha$ needs to be processed in the next iteration. Otherwise, it can be ignored in the next iteration because those values were already processed and no changes were made from the current iteration. If the new bucket $c\alpha$ needs to be processed, then an offset into each input BWT and a total length for the bucket (for a two-way merge, this is a total of three integer values) are stored for the next iteration. Once an iteration is reached with no uncoverged buckets, the merge is complete.

Table 4.1 shows the initial state of the modified merge algorithm. For a merge with N total symbols, the initial state is always a single bucket corresponding to the empty string (“") with the bucket range $[0, N)$. This initial range is always processed and creates a new bucket for each 1-mer in the dataset. The interleave and list of buckets after iteration 1 is shown in Table 4.2. After iteration 1, there are three buckets corresponding to the 1-mers “\$", “A", and “C". Additionally, the interleave corresponding to each range was modified from the initial state to the end of iteration 1. For example, the range corresponding to “\$" ($[0, 2)$) went from $[0, 0]$ to $[0, 1]$, so the bucket is marked as changed.

Iteration 2 shows an example where not all buckets change from iteration 1. In Table 4.2, consider the bucket representing $\alpha =$“\$" at the end of iteration 1. This bucket corresponds to indices $[0, 2)$ of the current interleave. Those indices have interleave bits 0 and 1 with symbols ‘C’ and ‘A’ respectively, so the two possible new buckets generated from “\$" in iteration 2 (see Table 4.3) are “C\$" and “A\$". For “C\$", the algorithm writes a 0 bit to index 6 (this is from the normal merge

Initial State

Index	I_0	**BWT**	**Suffix**
0	0	C	$AAC
1	0	$	AAC$
2	0	A	AC$A
3	0	A	C$AA
4	1	A	$CAA
5	1	A	A$CA
6	1	C	AA$C
7	1	$	CAA$

i**-mer**	**Bucket Range**
“"	[0, 8)

Table 4.1: Bucket example - initial state. These tables show the initial state of the merge of two strings “AAC$" and “CAA$" with BWTs “C$AA" and “AAC$" respectively. The initial interleave, I_0, is assumed to be a concatenation of the two BWTs. The initial interleave, BWT, and implicit suffixes corresponding to the BWT are shown in the left table. The right table shows the initial buckets to be processed in iteration 1 of the merge algorithm. There is only one bucket representing the empty string. This initial bucket covers the entire merged BWT with range $[0, 8)$.

method). Since index 6 is already a 0 bit, the “C$" bucket does not need to be processed in iteration 3. For “A$", the algorithm writes a 1 bit to index 2. Since this causes a change, the “A$" bucket must be processed in iteration 3.

Tables 4.1-4.4 show a full example for this modified merge algorithm. In the example, the buckets are defined purely on the range in the merged BWT for brevity. In practice, the bucket for a two-way merge is actually defined by a tuple containing three values: two offsets (one into each input BWT) and a total length for the bucket. For example, for a merge of N total symbols, the initial tuple for the empty string will always be $(0, 0, N)$. During each iteration, the buckets are processed in lexicographical order based on the i-mer they represent (this is derived from the sum of the offset values).

The original merge algorithm calculates the FM-index on the fly while linearly traversing the BWTs. However, the buckets allow the merge algorithm to skip over section of the BWT, so the FM-index cannot be calculated on the fly anymore. Instead, a sampled FM-index for each BWT is calculated beforehand (this can be trivially done in $O(N)$ time), and the FM-index entries are accessed as needed for each bucket. As discussed in Section 2.3.3, the extra memory cost for storing an FM-index that is sampled every B values is $O(\frac{N\sigma}{B})$. Additionally, each lookup for a sampled FM-index is $O(B)$ instead of $O(1)$.

The original merge algorithm (Chapter 3) has a time complexity of $O(N * LCP_{max})$ where N

Iteration 1

Index	I_0	I_1	**BWT**	**Suffix**
0	0	0	C	\$AAC
1	0	1	A	\$CAA
2	0	0	\$	AAC\$
3	0	0	A	AC\$A
4	1	1	A	A\$CA
5	1	1	C	AA\$C
6	1	0	A	C\$AA
7	1	1	\$	CAA\$

i**-mer**	**Bucket Range**	**Changed**
\$	[0, 2)	Yes
A	[2, 6)	Yes
C	[6, 8)	Yes

Table 4.2: Bucket example - iteration 1. These tables show the conditions of the merge at the end of iteration 1. I_0 is the initial interleave and I_1 is the state of the interleave after the first iteration. In this first pass, only one bucket corresponding to the empty string and the full BWT was processed. Inside that range, there were three symbols, '\$', 'A', and 'C', that occurred 2, 4, and 2 times respectively. As a result, the algorithm creates three new buckets corresponding to the 1-mers "\$", "A", and "C" with the ranges shown above. Additionally, the interleave and corresponding BWT symbols for each bucket changed from iteration 1, so they will all be processed in iteration 2. Note that the algorithm stores additional information for each bucket corresponding to FM-index related values that are not shown here for brevity.

Iteration 2

Index	I_1	I_2	**MSBWT**	**Suffix**
0	0	0	C	\$AAC
1	1	1	A	\$CAA
2	0	1	A	A\$CA
3	0	0	\$	AAC\$
4	1	1	C	AA\$C
5	1	0	A	AC\$A
6	0	0	A	C\$AA
7	1	1	\$	CAA\$

i**-mer**	**Bucket Range**	**Changed**
\$A	[0, 1)	No
\$C	[1, 2)	No
A\$	[2, 3)	Yes
AA	[3, 5)	Yes
AC	[5, 6)	Yes
C\$	[6, 7)	No
CA	[7, 8)	No

Table 4.3: Bucket example - iteration 2. These tables show the conditions of the merge at the end of iteration 2. I_1 is the interleave at the end of iteration 1 and I_2 is the interleave at the end of iteration 2. In the second pass, the three buckets from iteration 1 are processed. The bucket correspond to "\$" is scanned and leads to the creation of two new buckets corresponding to "A\$" and "C\$". The "A\$" bucket corresponds to the range [2, 3). Note that after iteration 1 at index 2, the interleave and BWT were 0 and '\$'. In contrast, the values after iteration 2 at index 2 are 1 and 'A', indicating that a change has occurred. The second created bucket corresponding to "C\$" and range [6, 7) did not show any change from iteration 1 to 2 (interleave and BWT were 0 and 'A' in both). As a result, the "A\$" bucket will be processed in iteration 3 but the "C\$" bucket will not be. The full list of buckets generated is shown in the right table. Note that out of the seven 2-mers in this merge, only three of them will be processed in iteration 3.

Iteration 3

Index	I_2	I_3	MSBWT	Suffix
0	0	0	C	\$AAC
1	1	1	A	\$CAA
2	1	1	A	A\$CA
3	0	1	C	AA\$C
4	1	0	\$	AAC\$
5	0	0	A	AC\$A
6	0	0	A	C\$AA
7	1	1	\$	CAA\$

i-mer	Bucket Range	Changed
\$AA	[0, 1)	No
AA\$	[3, 4)	Yes
AAC	[4, 5)	Yes
CAA	[7, 8)	No

Table 4.4: Bucket example - iteration 3. These tables show the conditions of the merge at the end of iteration 3. I_2 is the interleave at the end of iteration 2 and I_3 is the interleave at the end of iteration 3. Note that the interleave at the end of iteration is correct and the suffixes are now correctly sorted. However, the algorithm has not yet detected convergence. At the end of iteration 3, two buckets have changed. When processed in iteration 4 (not shown), they will create two new buckets that will not change. Since no buckets changed at the end of iteration 4, the algorithm will save the BWT using the interleave and terminate.

is the total number of symbols in the merge and LCP_{max} is the maximum LCP between any two suffixes of the merge. With bucket tracking, the algorithm only checks unconverged i-mer buckets at each iteration. A unique suffix, α, and its associated bucket will converge once $i = LCP_\alpha + 1$, so the computation time for the tracked merge is $\Sigma_{x=1}^{N}(LCP_x + 1)$. This is equivalent to $O(N * LCP_{avg})$ where LCP_{avg} is the average LCP across all suffixes.

One problem with tracking this way is that the number of unconverged i-mer buckets can be very large and may lead to performance problems if the size of the structure for tracking buckets exceeds the amount of available memory. Given $\Sigma = (\$, A, C, G, N, T)$ and $\sigma = 6$, there are up to 6^i possible i-mer buckets in a given iteration. In our specific datasets, we found that the number of unconverged i-mer buckets followed this exponential growth curve until approximately iteration 13, decayed rapidly until about iteration 20, and then decayed steadily in subsequent iterations until convergence was achieved. After iteration 20, there were relatively few unconverged i-mers which actually needed to be processed by the merge algorithm. In other words, significant portions of the interleave were already converged to the correct answer after only 20 iterations.

To avoid the exponential memory growth of bucket tracking, the tracking method was further modified by delaying its start until iteration 20, so only later iterations ($i > 20$) track which $(i - 20)$-mer buckets are not converged. The intuition behind this change is that the number of unconverged $(i-20)$-mers at iteration i is always less than or equal to the number of unconverged i-mers at that same iteration. By delaying the tracking to a later iteration, the memory consumption peak is avoided. The selection of iteration 20 is a heuristic derived from observing our particular datasets. This optimal delay is likely related to the number of expected k-mer in the datasets. In other words, the optimal delay can be written as a function of genome length and sequencing error. In a genome of length G, the expected number of k-mers is $log_4(G)$, so after $log_4(G)$ iterations we expect the buckets to quickly converge to the correct solution.

This final algorithm performs 20 standard iterations over the entire interleave followed by $(LCP_{max} - 20)$ partial iterations (buckets are generated only for unconverged regions). The time complexity of this algorithm is therefore $N * (20 + LCP_{avg})$ or $O(N * LCP_{avg})$. The only additional memory cost is the tracking of unconverged i-mer buckets where $i > 20$. Assuming there is a constant decay in unconverged buckets after iteration 20 (as it is with our datasets), the additional memory cost is $O(s)$ where s is the number of shared 20-mers in the two input MSBWTs. In general, $s << N$ because there are usually duplicate 20-mers in the MSBWTs. In our presented results, the maximum size of this bucket tracker never exceeded 1.5 GB of memory.

4.3.2 Divide-and-conquer merge

If the original merge algorithm is naïvely used, the interleave vectors require $O(N * \log(m))$ bits of storage for m strings composed of a total of N bases. The merge algorithm iterates over the interleave vector several times in order to reach convergence. As a result, the algorithm needs to keep the interleave in main memory or risk a significant performance penalty as a result of disk I/O. Unfortunately, we found that using O($N * \log(m)$) bits of space was typically too large to fit even on machines with 32 GB of RAM. The is primarily due to the fact that there are often millions of strings in any given dataset, so the $\log(m)$ term becomes quite large.

To address this issue, the second major change made to the merge algorithm was to apply a divide-and-conquer approach in order to reduce the size of the interleave vector at any given time. At each level, pairs of BWTs are merged into larger BWTs at the next level. In other words, at each level the number of BWTs is cut in half but each BWT contains double the number of strings from the previous level. Using this approach, only two inputs are given to each merge, so the interleave vector requires only one bit per base. This reduced the maximum amount of memory needed for the interleaves to $O(N)$ instead of $O(N * \log(m))$. Furthermore, each merge can be computed independently by separate processes allowing for trivial multi-processing at low levels in the divide-and-conquer. After $O(\log(m))$ levels, one BWT contains every string from the original set. We allowed the BWT to perform merges of 256 and 2 BWTs simultaneously. Merging 256 BWTs requires using 1 byte per interleave value ($O(N)$ bytes) and merging 2 BWTs requires 1 bit per interleave value ($O(N)$ bits). Even if the interleave cannot be kept in main memory, it can be stored on disk and accessed in a non-random manner to compute the BWT. Figure 4.1 shows the theory behind which merge can be used as a function of the amount of available memory.

The divide-and-conquer approach changed both the time and memory requirements of the algorithm. As stated earlier, the memory cost was reduced from $O(N * \log(m))$ bits to $O(N)$ bits. In practice, we found the divide-and-conquer approach is much faster because the interleave vector can be kept entirely in memory even on memory limited machines. However, this comes at the cost of changing the time complexity from $O(N * LCP_{avg})$ to $O(N * LCP_{avg} * \log(m))$.

4.4 Results

4.4.1 Dataset description

We performed all tests on a single machine running Ubuntu 14.04 with 32 GB memory and an Intel Xeon E5-2620 6-core 2.00 GHz processor. We limited each test to at most four processors during the construction. The machine is connected to a 1 TB HDD for reading and writing any necessary input or output files.

We selected six datasets to use as test data for the construction algorithm. Two of the datasets are PacBio Circular Consensus Sequencing (CCS) and Continuous Long Read (CLR) reads for *E. coli* K12 MG1655[1]. We also used another PacBio CLR dataset for *D. melanogaster*[2]. In addition to PacBio datasets, we selected two Illumina datasets, SRR027520 (Human) and SRR065390 (*C. elegans*), that are datasets from the 1000 Genomes project [Consortium et al., 2010]. The final dataset is an African male dataset from the Sequence Read Archive ERA015743[3] used in previous studies of BWT construction [Bauer et al., 2011, 2013]. The ERA015743 is separated into multiple SRF files that we used to create paired-end FASTQ files. The first test is one FASTQ file, the

[1]https://github.com/PacificBiosciences/DevNet/wiki/Datasets
[2]http://blog.pacificbiosciences.com/2014/01/data-release-preliminary-de-novo.html
[3]ftp://ftp.sra.ebi.ac.uk/vol1/ERA015/ERA015743/srf/

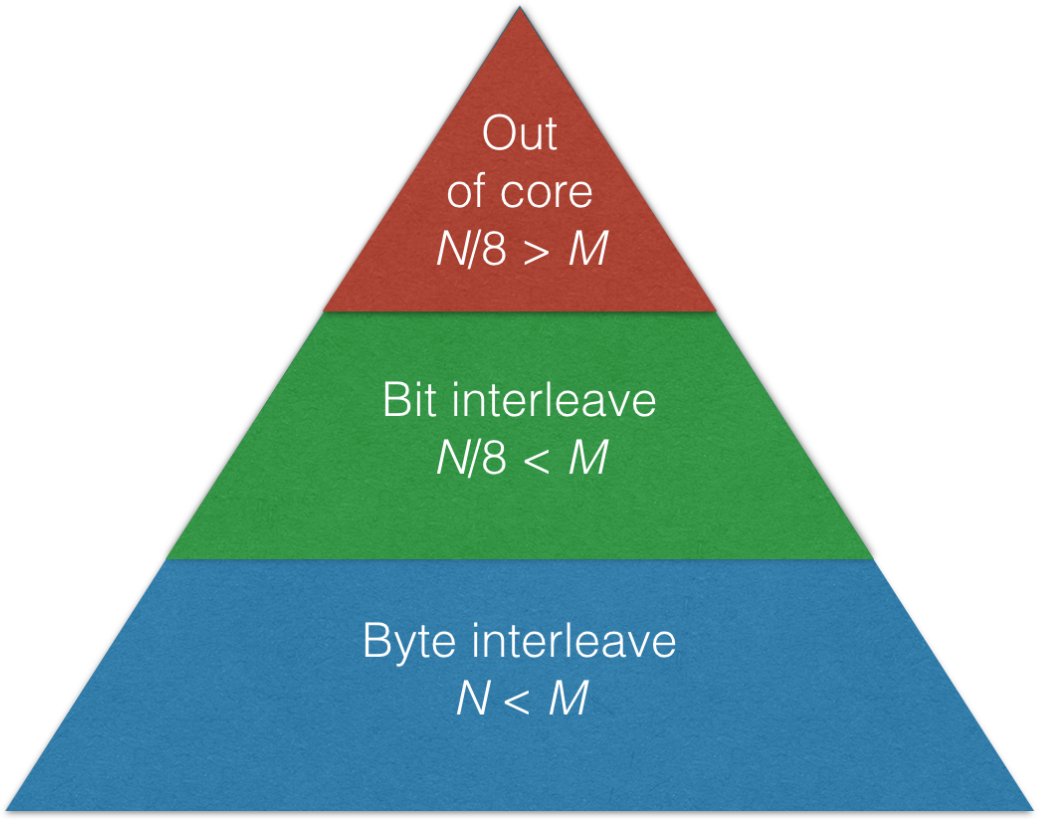


Figure 4.1: Divide-and-conquer merge. This figure visualizes the memory requirements for the divide-and-conquer merge as a function of the number of symbols being merged simultaneously, N, and the amount of memory, M, available on the machine. If $N < M$, then 256 BWTs can be merged simultaneously using an interleave that requires 1 byte per value (a total of N bytes). If $\frac{N}{8} < M$, then two BWTs can be merged using an interleave that requires 1 bit per value (a total of N bits). If $\frac{N}{8} > M$, then the interleave will not fit in main memory, and an out-of-core interleave must be used. The lowest levels of merging can use byte interleaves because there are relatively few symbols in the merge. Eventually, this has to transition to bit interleaves as the number of symbols increases. Finally, the highest levels of merging may require out-of-core allocation of the interleave because it is simply too large even with 1 bit per symbol.

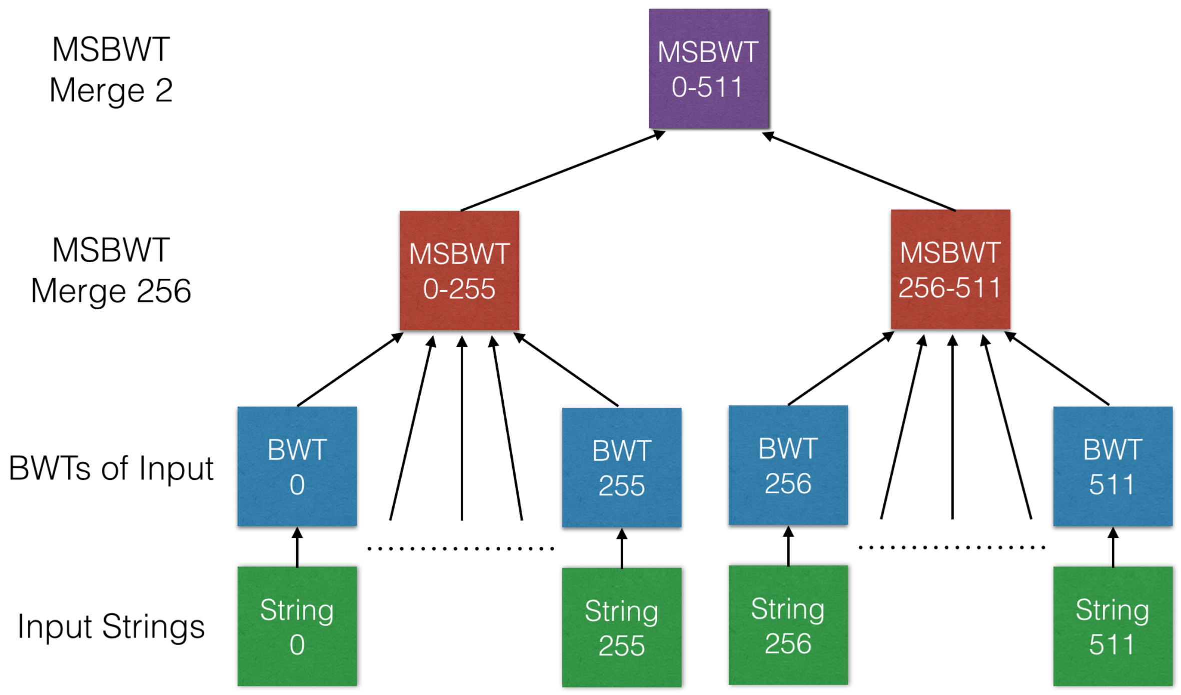


Figure 4.2: Divide-and-conquer merge implementation. This figure shows the layout of divide-and-conquer merges for a string collection containing 512 strings. First, each input string is transformed into a single-string BWT. Then, groups of 256 BWTs are merged using the original merge algorithm into multi-string BWTs containing 256 strings. In all subsequent levels, two BWTs are merged into one larger BWT using the delayed tracking algorithm. The merging terminates when only one BWT remains. In this example, only one two-way delayed tracking merge is necessary. Note that at each level, all BWT creations or BWT merges can be handled by separate processes in parallel.

second two files, and the third test is four files. All datasets were transformed into FASTQ format for testing.

4.4.2 Divide-and-conquer application

To build the merged BWT, the algorithm started by constructing the BWT for each string individually. Then, it merged those single-string BWTs into multi-string BWTs of 256 strings using the basic, unmodified BWT merge algorithm. These small merges required one byte per base in the interleave vector. Since only 256 strings were merged at the first stage, the size of the interleave vector for each merge was relatively small. After merging the strings into BWTs containing 256 strings, the algorithm used the two-way merge described in this chapter for subsequent merges. Figure 4.2 shows the layout of the merge implementation for a collection of 512 strings. All results presented in this chapter follow this divide-and-conquer approach to BWT construction.

4.4.3 Delayed tracking benefits

To demonstrate the benefits of delayed tracking, we ran the *E. coli* CLR dataset with three different types of convergence tracking: no tracking, full tracking, and delayed tracking. No tracking is just using the merge algorithm as described in Chapter 3 [Holt and McMillan, 2014a]. Full tracking monitors unconverged buckets from the first iteration such that it is tracking all unconverged i-mers at iteration i. Delayed tracking does not start tracking buckets until iteration 20 such that it is only tracking unconverged $(i-20)$-mers at iteration i. We plotted cumulative iteration times for the final merge in Figure 4.3. As expected, the cumulative time for no tracking grows linearly because each iteration requires a constant amount of work. The overhead from full tracking leads to longer early iterations followed by very short iterations later. Since delayed tracking does not start until iteration 20, the algorithm bypasses the peak quantity of unconverged buckets during early iterations while still benefiting from the speedups of tracking in later iterations.

A comparison of the quantity of unconverged buckets for both the full tracking and delayed tracking is shown in Figure 4.4. For this dataset, the peak quantity of unconverged buckets with full tracking is approximately 23 million. With delayed tracking, the peak quantity is significantly lower at approximately 25 thousand buckets. After peaking, the delayed tracking roughly follows the full tracking but with fewer buckets. The difference in buckets is caused by the fact that full tracking is monitoring all i-mer buckets whereas delayed tracking is monitoring $(i-20)$-mer buckets. As a result, some of the i-mer buckets are combined into one $(i-20)$-mer bucket, leading to overall fewer buckets to track with the delayed merging.

4.4.4 Full construction times

The timings and efficiencies for the merge-based construction algorithm on each dataset are summarized in Table 4.5. Both wall clock and efficiency were measured using Ubuntu's built-in `/usr/bin/time` command. The measured time includes reading FASTQ inputs, creating BWTs from each input, and performing all merges to the final result. Wall clock time is reported as microseconds per base to allow for easier comparison between datasets. For four processes, the maximum efficiency is 400% (each process at 100%).

These results demonstrate the effect of LCP_{avg} on the run time for the algorithm. Several factors influence LCP_{avg} including coverage, read length, and the error rate of the sequencing technology. The fastest results are for PacBio CLR data (long reads, high error rate). This is most likely because

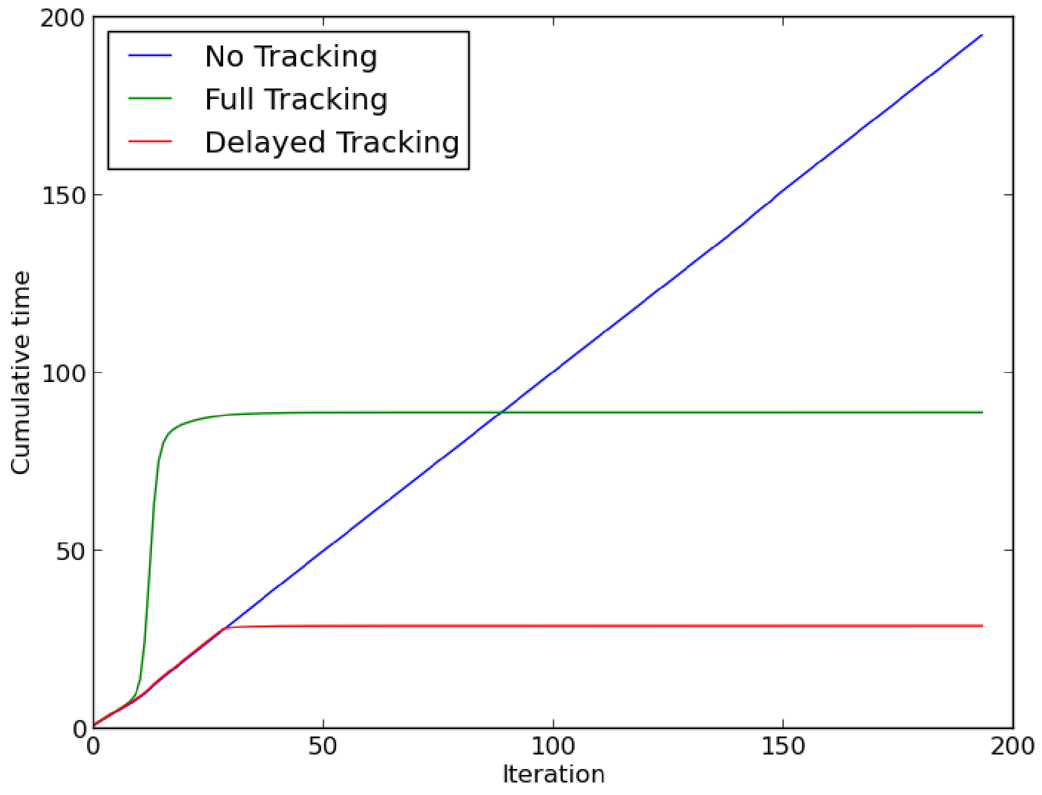


Figure 4.3: Tracking run-time comparison. This figure shows the time to execute the final merge of an *E. coli* CLR dataset with three different types of tracking: no tracking, full tracking, and delayed tracking. Along the x-axis is the iteration of the merge. The y-axis is the cumulative time to perform the iterations up to that point. No tracking is the basic merge algorithm as described in Chapter 3 [Holt and McMillan, 2014a]. As expected, it requires a constant amount of time for each iteration. Full tracking monitors all unconverged buckets from the first iteration such that at iteration i, the buckets represent i-mers. This causes slower early iterations when there are a large number of unconverged buckets followed by fast iterations when the number of buckets becomes small. Delayed tracking does not start tracking buckets until after 20 iterations, allowing it to avoid the overhead caused by tracking a large quantity of buckets while still benefiting from the speedup that full tracking achieves in later iterations.

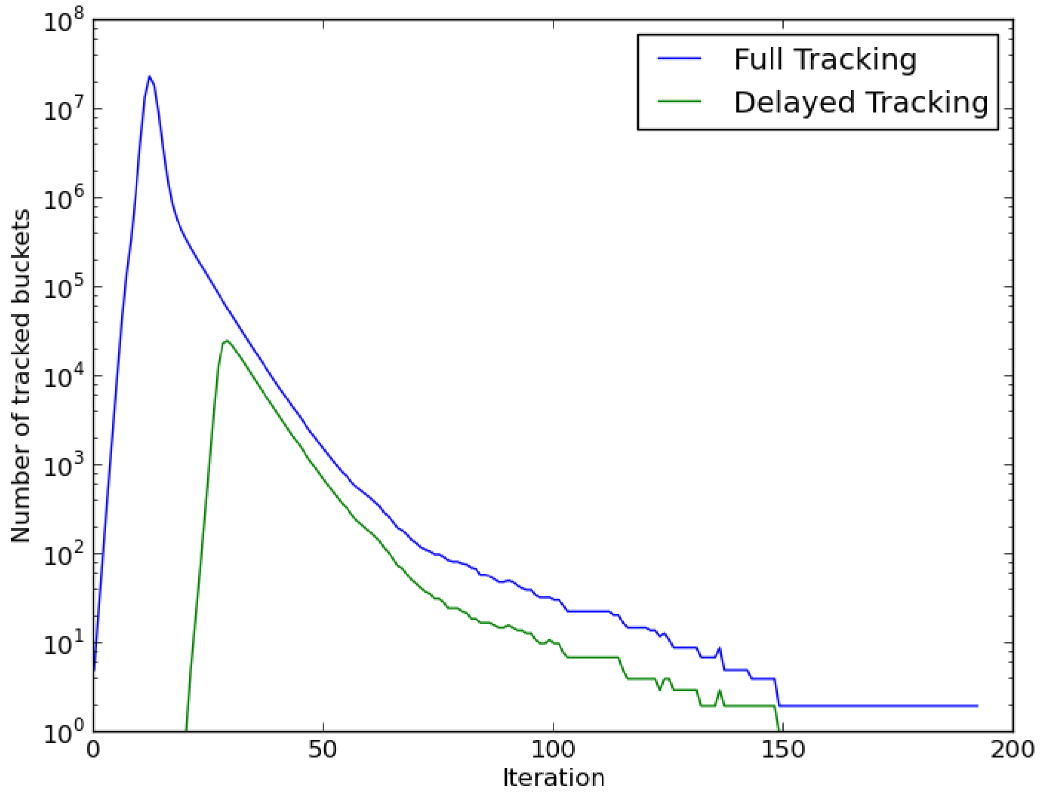


Figure 4.4: Tracking memory comparison. This is a plot of the number of tracked, unconverged buckets during each iteration of the final merge of an *E. coli* CLR dataset. The full tracking method tracks buckets from the very first iteration which leads to a peak quantity of 23 million buckets after iteration 13. The delayed tracking begins tracking at iteration 20 and has a significantly smaller peak quantity of 25 thousand buckets after iteration 30. Delayed tracking then follows the same decaying trend as full tracking until the merge is complete.

Dataset	Type	Reads	Bases	Avg. Len.	Wall clock	Efficiency
E. coli K12 MG1655	PacBio - CCS	231,629	217,871,193	940.6	5.26	252%
E. coli K12 MG1655	PacBio - CLR	50,765	98,213,822	1934.6	1.57	227%
D. melanogaster	PacBio - CLR	1,514,730	15,208,567,933	10,040.4	2.68	261%
SRR027520 - Human	Illumina	24,246,685	1,842,748,060	76	4.00	310%
SRR065390 - C. elegans	Illumina	33,808,546	3,380,854,600	100	4.52	292%
ERA015743 - Human 26M	Illumina	26,520,387	2,678,559,087	101	5.52	208%
ERA015743 - Human 53M	Illumina	53,040,774	5,357,118,174	101	3.95	303%
ERA015743 - Human 104M	Illumina	104,830,294	10,587,859,694	101	4.29	300%

Table 4.5: Merge-based construction performance. This table shows the time and efficiency of the merge-based construction algorithm for several datasets. The type of data, number of reads, number of bases, and average read length for each dataset are shown for comparison. For each dataset, the merge-based construction algorithm is timed using the built-in `/usr/bin/time` function. Wall clock time is reported as microseconds per base to allow for comparison between different data types. Each test is allowed four processes so the maximum achievable efficiency is 400%. In these results, most Illumina datasets require 4 or more microseconds per base. In contrast, the largest dataset (15 billion base *D. melanogaster* PacBio CLR) required only 2.68 microseconds per base despite having significantly longer reads and more total bases. For the PacBio CCS dataset, this algorithm achieved similar levels of performance as it did for the Illumina datasets even though the reads are approximately nine times longer. None of the tests achieved maximum efficiency with the best utilizing 310%.

CLR data has a relatively high sequencing error rate leading to a smaller average LCP. In contrast, Illumina datasets (short reads, low error rate) required more time per base despite having fewer total bases than the *D. melanogaster* PacBio CLR dataset. Finally, the one PacBio CCS dataset (long reads, low error rate) required about one microsecond per base more time than the Illumina datasets. This is due to the high LCP_{avg} resulting from the combination of long reads and low error rates.

Each test case fell short of achieving maximum efficiency. Some of this can be attributed to disk I/O, but the majority is inefficiency in the multi-processing. In the final stages of the construction, not every process is utilized since there are fewer merges than available processes (the final merge uses only one of the four processors). This leads to slower merges in the final stages since the work is no longer distributed evenly amongst all available processors.

4.5 Final notes

The merge-based BWT construction algorithm described is one of the earliest to directly address the problem of constructing the BWT from a string collection of long sequencing reads. While methods existed for uniform-length, short reads, they did not scale well as the read length increased. The asymptotic performance of the merge-based algorithm is not directly dependent on read length. Additionally, the strings do not need to be of uniform length for the algorithm to work correctly.

In these particular tests, a 20-iteration tracking delay was chosen heuristically. The delay tended to work well for these particular tests to avoid the peak memory caused by storing millions of i-mers in iteration i. However, a 20-iteration delay is likely not the optimal choice for all datasets. Datasets with a high number of repeated substrings may need to delay the tracking further in order to keep the list in memory. In contrast, datasets with a low number of repeated substrings may actually be able to begin tracking much earlier and converge on the correct solution faster. Furthermore, dynamically selecting the tracking method based on a fixed number of allowed buckets may enable the algorithm to select the optimal tracking delay while computing the BWT instead of a hard-coded delay.

As seen in the results, there are certain types of sequencing data that this algorithm performs better on. In particular, the long, noisy reads generated by some sequencers (such as PacBio CLR) are likely the best case scenario for the merge-based algorithm. The reasons for this are two fold: there is a low LCP_{avg} due to sequencing error and there are fewer reads with respect to the total number of sequenced bases. In contrast, datasets with a high LCP_{avg} and a high number of reads have the worst performance. As a result, this method is likely ill-suited for high-coverage sequencing runs that report billions of reads or long-read sequencers with low error rates.

Subsequently, additional modifications to the BCR algorithms led to the newer ropebwt2 implementation [Li, 2014]. While the performance of ropebwt2 decays as the read length increases, it outperforms both previous BCR algorithms and the merge-based BWT construction algorithm for most datasets. The next chapter introduces another BWT construction method that also works with long-read datasets but is less influenced by the error rate or the number of reads in the dataset.

CHAPTER 5

MULTI-STRING BWT CONSTRUCTION VIA INDUCED SORTING

Up to this point, three methods of constructing BWTs from a string collection have been introduced. The first method naïvely constructs the BWT from a suffix array which requires an expensive sort of all suffixes in the entire string collection (see Section 2.4). The second method is the “BCR" class of algorithms that build the BWT in a “column-wise" manner [Bauer et al., 2011, 2013]. This class of algorithms tends to work well for short, uniform length strings, but their performance falls off as the length of the strings increases. The third method is the merge-based construction algorithm from Chapter 4 [Holt and McMillan, 2014b]. While this method is one of the first to build BWTs of long reads, its performance decays as the number of strings in the collection grows and as the self-consistency (longest common substrings) of the strings increases.

In this chapter, we introduce a fourth method of multi-string BWT construction called MSBWT-IS that is based on the concept of induced sorting. The main idea of induced sorting is to compute a partial order of a subset of suffixes and to use that partial order to solve for a complete order of all suffixes. Induced sorting methods for both suffix arrays [Nong et al., 2009] and single-string BWTs [Okanohara and Sadakane, 2009] have been previously developed by other researchers. MSBWT-IS method is the natural extension of induced sorting to building BWTs of string collections. For a string collection with a total of N symbols, this algorithm requires $O(N)$ steps to compute the BWT. MSBWT-IS requires less CPU time than other algorithms for long-read sequencing datasets, and it is trivially adapted to constructing BWTs for any input alphabet.

5.1 Related work

In Section 4.1, we first introduced a class of algorithms called “BCR" [Bauer et al., 2011, 2013]. These algorithms all follow the same general approach to BWT construction. Given a string collection with m strings that are all k symbols long, the strings can be stacked such that each string is a row and each column is a group of symbols that are all at the same index in the strings. The BCR algorithms then insert symbols in a “column-wise" manner [Bauer et al., 2011, 2013], such that all symbols in a given column are inserted in one iteration of construction. With k insertion iterations that insert exactly m symbols, there are $O(mk)$ steps. However, the major drawback of this approach is the $O(mk^2)$ disk I/O [Bauer et al., 2013]. This is caused by performing full writes of the partial BWT to disk at the end of each iteration. Additionally, the algorithm requires pre-sorting the strings prior to construction. This class of algorithm tends to perform well on uniform, short-length strings such as reads from an Illumina sequencing run. However, these implementations of the BCR algorithm do not scale well as the length of the strings in the collection increases.

In Chapter 4, we described an alternate approach to BWT construction that repeatedly merges BWTs created from subsets of the entire string collection [Holt and McMillan, 2014a]. Given m strings with a combined length of N (note that $N = mk$ for m reads of uniform length k) and an average longest common prefix of LCP_{avg}, the run-time for this algorithm is $O(N * LCP_{avg} * \log(m))$. This algorithm tends to perform well when the length of shared substrings is relatively short and there are few strings in the collection. In particular, tests showed that this method worked well for long, noisy sequencing reads such as those generated by PacBio CLR sequencing. However, datasets with a large number of strings required many merges, and highly redundant datasets caused each individual merge to require more computation. As a result, this method does not scale well to large, high coverage sequencing datasets.

A recently developed algorithm, called “ropebwt2", is an adaptation of the BCR algorithm that uses in-memory B+-trees instead of arrays [Li, 2014]. The primary drawback of the other BCR algorithms was keeping the partial BWTs as arrays either in-memory or on-disk. As a result, each iteration of the algorithm required re-allocation of the array in order to make insertions. The “ropebwt2" algorithm instead keeps the partial BWTs in a run length encoded form inside a B+-tree. For a string collection with N total symbols, this reportedly requires $O(N * \log(N))$ steps to construct the BWT in the B+-tree format [Li, 2014]. However, in our results we identified that the length of strings in the collection still impacts performance as it does in previous versions of the BCR algorithm. However, the ropebwt2 implementation is much faster than previous BCR implementations [Li, 2014]. Additionally, this method outperforms the merge-based BWT construction algorithm even when given an ideal dataset for the merge algorithm (few reads and a low LCP_{avg}).

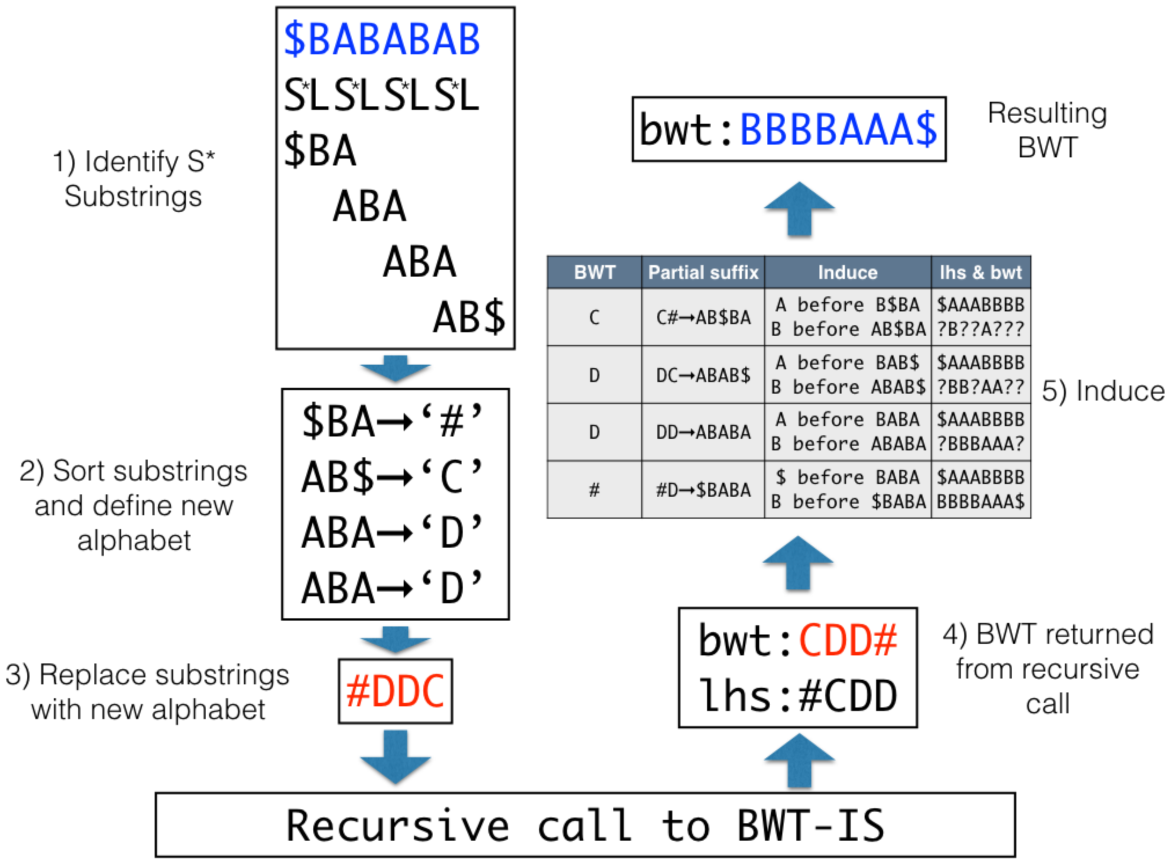
The publicly available implementation of “ropebwt2" can build the BWT in many different formats [Li, 2014]. In order to build the BWT in the format described in Chapter 2, the input strings must all be sorted lexicographically prior to running the algorithm. For large string collections, this can be a major drawback. The implementation can also build the BWT in a reverse-lexicographic order without pre-sorting the strings [Li, 2014]. Reverse-lexicographic order (RLO) reorders the strings in the collection such that they are stored in reverse lexicographic order with the primary benefit of increasing the length of runs in the BWT [Cox et al., 2012a]. Unfortunately, this method breaks the cyclic property of suffixes in the implicit suffix array. In other words, repeatedly finding predecessor symbols will likely lead to traversal of many strings instead of just one, so additional storage overhead is required to correctly traverse through the end-of-string characters of the strings. There is a third ordering of the strings called reverse-complement lexicographic ordering (RCLO) that encodes both forward and reverse-complemented strings in the BWT. When both the forward and reverse complement of every string in the collection are stored in reverse lexicographic order, every string forms a cycle with its reverse complement, alleviating the need for the additional overhead caused by RLO and the need for sorting the input prior to construction [Li, 2014]. A BWT that is RCLO-encoded stores twice as many symbols and any search finds both forward and reverse-complement sequence simultaneously. However, this method loses information about the original sequence since it cannot distinguish between forward and reverse-complemented strings without storing additional metadata for each string.

Two other algorithms, SA-IS [Nong et al., 2009] and BWT-IS [Okanohara and Sadakane, 2009], build the suffix array and BWT respectively for single strings using induced sorting. Given a single long string, induced sorting breaks the string into many substrings and replaces each substring with a new symbol from a new alphabet. This new alphabet preserves the lexicographic ordering of the substrings and each symbol in the new alphabet encodes at least two symbols from the previous alphabet. As a result, the new string is guaranteed to be at most half the size of the original at the cost of a larger alphabet. The algorithm is then recursively called on the new, shorter string until the BWT of the string can be trivially computed. When exiting the recursion, the BWT of the shorter string represents a partial ordering of the suffixes from the longer string. This partial ordering is then used to “induce" the full ordering and the BWT of the longer string. For a string of length M, BWT-IS requires $O(M)$ steps to compute the BWT of the string [Okanohara and Sadakane, 2009]. Our approach adapts the induced sorting method to build the BWT of a string collection instead of a single string by explicitly storing the substring alphabet at each step and modifying the recursion termination condition for a string collection.

5.2 Approach

MSBWT-IS is based on a single-string BWT construction algorithm called BWT-IS [Okanohara and Sadakane, 2009]. The main concept is to use *induced sorting* to recursively solve for the BWT. Initially, the BWT-IS algorithm splits a single long string, T_i, of length M into several much smaller "S* substrings". These substrings are then *implicitly* sorted (using an $O(M)$ method) and each is assigned a new integer symbol corresponding to its order in the implicit sort. Conceptually, these symbols represent a new alphabet that is not explicitly stored. Then, every S* substring is replaced with its new symbol to form a new, shorter string, T_{i+1}, composed of symbols from the new alphabet [Okanohara and Sadakane, 2009]. With each recursive call, the new string is guaranteed to be at most one half the size of the previous string [Okanohara and Sadakane, 2009]. Eventually, the string becomes small enough that the BWT can be trivially computed. The short BWT, B_{i+1}, represents a partial sort of the suffixes from T_i. Specifically, it is a sort of only the S* substrings from T_i. This partial sort is then used to "induce" the solution of the longer BWT, B_i. This step is repeated until the BWT for the original string is computed. Figure 5.1 shows an example of one level of the BWT-IS algorithm.

Our algorithm, MSBWT-IS, modifies this induced sorting algorithm to handle a collection of unsorted strings σ_i. In particular, the algorithm is adapted to *explicitly* sort the S* substrings of a string collection and save the newly created alphabet for use while exiting the recursion. In a linear pass over σ_i, it goes through each string and adds all S* substrings from all strings to an S* substring collection such that each S* substring is only represented once in the collection. In contrast, the BWT-IS algorithm does not remove duplicate strings during the sorting process. Then, these S* substrings are sorted against each other and assigned an integer value corresponding to its position in the sort. This new alphabet is explicitly stored for use during the inductive step. Finally, it constructs a new string collection, σ_{i+1}, by replacing each S* substring in each string with its corresponding symbol from the new alphabet. The algorithm recursively performs this set of operations until every string in the collection is one symbol long. Given that every string is one symbol long, the BWT of the collection can be trivially calculated with a one-pass radix sort of the symbols in the collection. Given a BWT, B_{i+1}, that is constructed from a higher level of recursion, the algorithm then induces the solution to B_i. Since B_{i+1} represents the lexicographic order of


$BABABAB
S*LS*LS*LS*L
$BA
ABA
ABA
AB$
1) Identify S* Substrings
$BA→‘#’
AB$→‘C’
ABA→‘D’
ABA→‘D’
2) Sort substrings and define new alphabet
#DDC
3) Replace substrings with new alphabet
Recursive call to BWT-IS
bwt:CDD#
lhs:#CDD
4) BWT returned from recursive call
5) Induce
bwt:BBBBAAA$
Resulting BWT

BWT	Partial suffix	Induce	lhs & bwt
C	C#→AB$BA	A before B$BA B before AB$BA	$AAABBBB ?B??A???
D	DC→ABAB$	A before BAB$ B before ABAB$	$AAABBBB ?BB?AA??
D	DD→ABABA	A before BABA B before ABABA	$AAABBBB ?BBBAAA?
#	#D→$BABA	$ before BABA B before $BABA	$AAABBBB BBBBAAA$

Figure 5.1: BWT-IS example. This figure shows one level of the BWT-IS induced sorting method on a single string “$BABABAB". First, the algorithm identifies S* substrings (see Section 5.3.1 for details). Second, the S* substrings are sorted and assigned a new symbol in a new alphabet such that the lexicographic ordering of the substrings is preserved. Note that one S* substring, “ABA", occurs twice and is assigned the same symbol, ‘D’, in the new alphabet. Third, the S* substrings are replaced with their new symbol in the new alphabet to create a new string. In this example, the new string is “#DDC". Fourth, this new, shorter string is passed into a recursive call of BWT-IS to compute the BWT. Note that the recursion eventually terminates because each recursive call generates a string that is at most half as long as the previous string. Eventually, computing the BWT of the string becomes trivial because it is relatively short. Finally, the resulting BWT, “CDD#" in this example, represents a partial ordering of the suffixes of “$BABABAB". The induce step uses this partial ordering to sort the remaining suffixes. For example, the smallest suffix in the shorter string begins with ‘#’ and is preceded by ‘C’ according to the BWT. This forms the partial suffix “C#" corresponding to partial suffix “AB$BA" in the original string. Thus, the symbols preceding partial suffixes “B$BA" and “AB$BA" are filled in the output BWT. This is repeated for every symbol from the shorter BWT until the entire resulting BWT for the long string is computed.

all S* substrings from B_i, the induce step needs to solve for the lexicographic order of all non-S* substrings in B_i. To do this, our algorithm pre-computes offsets into the BWT for each suffix of each S* substring in the alphabet such that the offset indicates the first suffix represented by the BWT that starts with the S* substring suffix. These offsets are calculated using an algorithm similar to the `Induce(...)` method of BWT-IS [Okanohara and Sadakane, 2009]. However, MSBWT-IS performs this `Induce(...)` subroutine on only the S* substrings in the alphabet (as opposed to the entire string as in BWT-IS), so it requires less space and time to compute the BWT. Finally, a linear traversal of B_{i+1} combined with the offsets allows the algorithm to induce the solution to B_i. A visual overview of this process is shown in Figure 5.2.

5.3 Construction via induced sorting

Given an alphabet Σ_i of length k_i, each symbol in the alphabet is represented as integers $[0..k_i-1]$. Additionally, assume that there are t_i *initializer/terminal* symbols that are the lexicographically smallest symbols in Σ_i. In other words, any symbol, s_i, such that $s_i < t_i$ is a initializer symbol in Σ_i. A string collection, σ_i, contains strings that are composed of symbols from Σ_i. Each string has exactly one terminal symbol that is always rotated to index 0 to become an initializer symbol.

We also define a series of properties for the initial string collection of the recursion. Let the initial string collection, σ_0, be composed of symbols from Σ_0, an alphabet with exactly one initializer symbol, $t_0 = 1$ (this single terminal symbol is commonly represented as '\$'). Additionally, let each string in this collection start with '\$', the only initializer symbol. Throughout our methods, we will be using an example with $\sigma_0 = \{$"\$TAGCT", "\$GAGCG"$\}$ and $\Sigma_0 = [\$, A, C, G, T]$ which can be represented by integer values $[0, 1, 2, 3, 4]$.

5.3.1 Computing the S* substring alphabet

The algorithm begins by classifying each symbol in each string in the collection using the same classification scheme of BWT-IS [Okanohara and Sadakane, 2009]. In any string T of length M, it assigns each symbol in T as being S-type or L-type. A symbol $T[x]$ is considered *S-type* if the suffix $T[x : M] < T[x + 1 : M]$. In other words, the suffix starting at index x lexicographically precedes the one starting at $(x + 1)$. Otherwise, it is considered *L-type.* Additionally, in any run of S-type symbols, the left-most S-type symbol is defined as *S*-type.* If $T[x]$ is S*-type, it is a local minimum in the string because $T[x : M] < T[x + 1 : M]$ and $T[x - 1 : M] > T[x : M]$. By the alphabet definition, all initializer symbols must be S*-type since they will always be the absolute minimum in

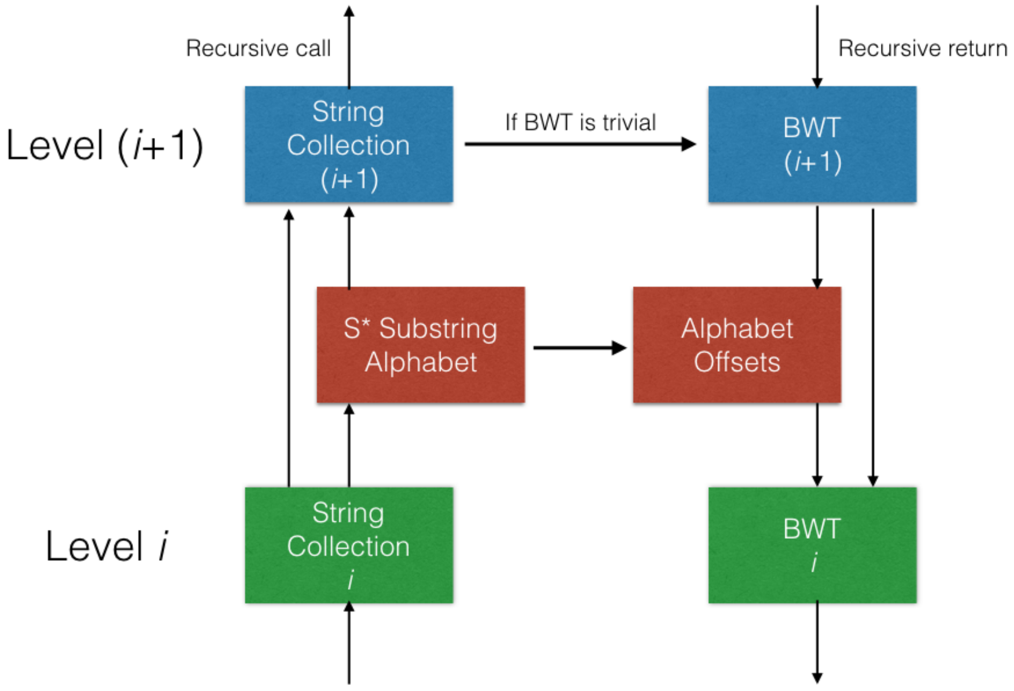


Figure 5.2: MSBWT-IS overview. This figure shows the outline for the induced sorting of MSBWT-IS. Starting at level i, the algorithm extract all S* substrings from the string collection. Then, it sorts the S* substrings and assigns each a new symbol in a new alphabet. Then, each S* substring from level i is replaced with its corresponding symbol from the new alphabet to create the new string collection at level $(i+1)$. If the calculation of the BWT of this new string collection is trivial, the algorithm will simply compute it. Otherwise, it will perform a recursive call on the new string collection. Once the BWT is obtained through either recursion or a trivial computation, all suffixes from the S* substrings are sorted against each other and used to assign an offset into the BWT at level i. Finally, these offsets and the BWT from level $(i+1)$ are used to induce the solution to the BWT at level i.

String Index	c (integer[c])	Type	S* Substring (integer[S*])
0	\$ (0)	S*	\$TA (041)
1	T (4)	L	
2	A (1)	S*	AGC (132)
3	G (3)	L	
4	C (2)	S*	CT\$ (240)
5	T (4)	L	
0	\$ (0)	S*	\$GA (031)
1	G (3)	L	
2	A (1)	S*	AGC (132)
3	G (3)	L	
4	C (2)	S*	CG\$ (230)
5	G (3)	L	

Table 5.1: S* substring example - level 0. This table shows the process for identifying S* substrings in string collection $\sigma_0 = \{$"\$TAGCT", "\$GAGCG"$\}$. All symbols that are local minima (based on integer[c]) are classified as S* type. The S* substrings are the strings from one S*-type up to and including the next S*-type symbol. All initializer symbols ('\$') are S*-type because an initializer symbol will always be an absolute minimum in the string.

any given string, so the first symbol in every string is S*-type. Table 5.1 shows how each symbol is labeled for the example collection containing two strings. Note that finding all S*-type symbols can be done for any string of length M in $O(M)$ steps [Okanohara and Sadakane, 2009]. To find the S*-type symbols in a string collection, the algorithm simply applies the same method on each string one at a time until all S*-type symbols are known. For a string collection of combined string length N_i, the algorithm can find the S* substrings in the entire collection in $O(N_i)$ steps.

Let an *S* substring* be defined as the substring from one S*-type symbol up to and including the next S*-type symbol [Okanohara and Sadakane, 2009]. Thus, the S* substrings for "\$TAGCT" are "\$TA", "AGC", and "CT\$" (see Table 5.1). In our algorithm, whenever an S* substring is found, it is explicitly stored in a collection. For our implementation, we used a hash map as the collection, so all S* substrings can be identified and stored in the collection in $O(N_i)$ steps.

Once all S* substrings are in the collection, the collection is sorted and each S* substring is assigned an integer symbol corresponding to its order in the sort. Thus, each S* substring has a one-to-one correspondence with a symbol in a new alphabet. For example, in Table 5.1, there are five S* substrings with sorted order ["\$GA", "\$TA", "AGC", "CG\$", "CT\$"] that corresponds to symbols [0, 1, 2, 3, 4] in Σ_1. It's also worth noting that at this point, Σ_1 has five total symbols ($k_1 = 5$), two of which correspond to initializer symbols ($t_1 = 2$). This is trivially found by counting

String Index	c	Type	S* Substring
0	1	S*	1241
1	2	S	
2	4	L	
0	0	S*	0230
1	2	S	
2	3	L	

Table 5.2: S* substring example - level 1. This table shows the process for identifying S* substrings in $\sigma_1 = \{124, 023\}$. All symbols that are local minima are classified as S* type. The S* substrings are the strings from one S*-type up to and including the next S*-type symbol. In this case, each string is entirely contained within one S* substring each.

the number of S* substrings in the collection that begin with an initializer symbol. In this example, both "$GA" and "$TA" start with '$', so '0' and '1' must be initializer symbols in Σ_1.

Using the new alphabet Σ_{i+1} and the input string collection σ_i, the algorithm then creates a new string collection σ_{i+1}. For each string T in σ_i, it replaces each S* substring in T with its corresponding value in the S* substring collection. For example, in Table 5.1, the string "$TAGCT" is composed of three S* substrings: "$TA", "AGC", and "CT$". Using the sorted alphabet, these S* substrings correspond to 1, 2, and 4 from Σ_1, so string "124" is added to σ_1. Similarly, the second string "$GAGCG" is added as "023" to σ_1.

5.3.2 Recursion

At each level of recursion, a new alphabet and string collection are computed. For each string T of length M, the new string in the collection is guaranteed to be at most $\frac{M}{2}$ symbols long [Okanohara and Sadakane, 2009]. Thus, for a string collection of combined string length N_i, it is known that $N_{i+1} \leq \frac{N_i}{2}$. In contrast, the change in alphabet size is entirely data dependent. Datasets that are highly repetitive tend to have smaller alphabets because the S* substrings are repeated. In our experience, the alphabet size tends to grow rapidly at early stages to compensate for the shortening of the strings. As the strings become shorter, the alphabet size tends to shrink simply because there are fewer symbols in the entire collection.

In BWT-IS, the recursion terminates when every symbol in the single string is unique [Okanohara and Sadakane, 2009]. However, with a string collection, this terminating condition is not sufficient as identical strings can exist in the collection, meaning it may be impossible to ever reach the unique symbol state. Instead, the terminating condition for the recursion is when all symbols in the alphabet are initializer symbols. Since a string can only have one initializer symbol, this condition implies that each string is only one symbol long. This termination condition is guaranteed to be reached eventually because the strings are always shortened with each level of recursion. Additionally, for a given string collection of m one-symbol-long strings, construction of a BWT can be accomplished in $O(m)$ steps using a one pass radix sort.

In our example, we previously solved for an alphabet Σ_1 with five symbols and string collection $\sigma_1 = \{124, 023\}$. After extracting and sorting all S* substrings from Σ_1, we get another alphabet Σ_2 with two symbols ($k_2 = 2$) that are both initializer symbols ($t_2 = 2$) and a string collection $\sigma_2 = \{1, 0\}$ (see Table 5.2 for the S* substrings). Since each string is only one symbol long, the predecessor symbol for any symbol in the BWT is itself, thus the BWT can be trivially constructed by simply sorting the symbols with a one pass radix sort. In our example, radix sorting σ_2 will get its BWT, $B_2 = [0, 1]$.

5.3.3 Inducing the BWT

Given a BWT for level $(i + 1)$, B_{i+1}, all that remains is to induce the BWT B_i. First, note that each symbol in B_{i+1} corresponds to an S* substring at level i. Thus, each symbol in B_{i+1} provides information about the ordering of all suffixes from the corresponding S* substring at level i. In our example, $B_2[0] = 0$ indicating that the smallest suffix in σ_2 is preceded by a '0'. This '0' corresponds to S* substring "0230" in σ_1. The algorithm then uses this information to induce the location of three symbols in B_1. Namely, the first suffix in B_1 starting with "30" is preceded by "02", starting with "230" is preceded by "0", and starting with "0230" is preceded by "3" (derived from B_2 and Σ_1). In other words, the single symbol from B_2 informs multiple symbols in B_1. However, the algorithm still needs to know where suffixes starting with "30", "230", and "0230" are located as offsets into the BWT (and implicit suffix array). In other words, it needs to know an offset value into B_i for each suffix of each S* substring from σ_i.

To compute the offsets, the algorithm needs to know both the lexicographic order of the S* substring suffixes and the number of times each S* substring suffix occurs in the entire string collection. Given the sorted order of the S* substring suffixes, the offset for a suffix is the sum of the frequencies of all suffixes that lexicographically precede it. For example, the S* substring suffixes of σ_1 are shown in sorted order in Table 5.3. In this example, each suffix occurs once, so the offsets start at 0 and increase by 1 for each offset.

S* substring suffix	Frequency	Offset into B_1
0230	1	0
1241	1	1
230	1	2
241	1	3
30	1	4
41	1	5

Table 5.3: S* substring suffixes - level 1. This table shows the sorted list of S* substring suffixes for $\sigma_1 = \{124, 023\}$. Note that all suffixes of length 1 are excluded because they are represented elsewhere in the suffix list. In this particular example, each suffix occurs once. The offsets are calculated by starting with 0 for the first S* substring suffix and adding the frequency to get the next offset. Since each suffix occurs once, the offsets are only increasing by one each time.

While the sorted order of the S* substring suffixes can be explicitly computed with a sort subroutine, we instead use a modified version of the `Induce(...)` subroutine of Okanohara and Sadakane [2009]. To summarize, the original `Induce(...)` subroutine computes the sorted order of *all* S* substring suffixes in the whole input string. In general, it does this by first solving for the sorted order of the L-type suffixes and then by solving for the reverse sorted order of all the S-type suffixes. The two sorts can then be trivially combined for a complete ordering of all S* substring suffixes. This algorithm runs in $O(M)$ steps for a string of length M.

Instead of computing the order of *every* S* substring suffix in the string collection, our method only computes the order of the S* substring suffixes in the stored alphabet. In other words, duplicated S* substring suffixes are removed prior to performing the sort. Then, it uses the computed order and the S* substring suffix frequencies to calculate offsets into the BWT. In short, MSBWT-IS only sorts k_i suffixes where k_i is the combined length of all S* substrings instead of M suffixes as in BWT-IS. This requires less memory than the original `Induce(...)` method because there are no repeated S* substrings in the computation. Additionally, since the alphabet already allows for the possibility of multiple terminal symbols, this adaptation enables the algorithm to run on string collections instead of just a single string.

The final step is to linearly traverse the BWT B_{i+1} and fill in B_i using the computed offsets. For each symbol c in B_{i+1}, the algorithm gets the corresponding S* substring in σ_1. Then, the offset for each S* substring suffix in the S* substring is extracted. Every time an offset is used, it is then incremented by one so that the next time the S* substring suffix is used, it is pointing to the next entry. Given the offset value, the character stored at the offset is the symbol preceding the S*

Index, x	$B_2[x]$	B_1
init	–	[?, ?, ?, ?, ?, ?]
0	0	[3, ?, 0, ?, 2, ?]
1	1	[3, 4, 0, 1, 2, 2]

Table 5.4: Induced BWT - level 1. This table shows the state of BWT B_1 as the values for the BWT B_2 are filled in. During each iteration x, the S* substring corresponding to symbol $B_2[x]$ and the offsets for each S* substring suffix are extracted and stored. Then, the corresponding predecessor symbol for each offset is written to output BWT B_1. Initially, the entire BWT is unknown. The first symbol encountered in B_2 is '0', corresponding to S* substring "0230" in σ_1. This S* substring has three suffixes ["30", "230", "0230"] corresponding to offsets [4, 2, 0] in Table 5.3. Thus, index 4 gets set to '2' (the symbol preceding "30"), index 2 gets set to '0', and index 0 gets set to '3'. This process is repeated for $B_2[1] = 1$ to finish solving for B_1.

substring suffix, which can be computed from the stored alphabet. In our example, the computed BWT $B_2 = [0, 1]$. The first symbol is $B_2[0] = 0$, corresponding to the S* substring "0230". This S* substring has three suffixes ["30", "230", "0230"] corresponding to offsets $[4, 2, 0]$. Thus, index 4 gets set to '2' (the symbol preceding "30"), index 2 gets set to '0', and index 0 gets set to '3'. We show the partial result in row 0 of Table 5.4. We then repeat the process for $B_2[1] = 1$ for S* substring "1231" to retrieve offsets [5, 3, 1] and symbols ['2', '1', '4']. The final result is shown in row 1 of Table 5.4.

At this point, the BWT for $\sigma_1 = [124, 023]$ has been solved as $B_1 = [3, 4, 0, 1, 2, 2]$. The final step is to exit the recursion to solve for $B_0 = BWT(\sigma_0)$. The algorithm uses the same method as before but with the alphabets from levels 0 and 1. The S* substring suffixes, frequencies, and offsets are shown in Table 5.5. Then each symbol from B_1 is processed one at a time, showing the state of B_0 after each symbol in Table 5.6.

5.3.4 Asymptotic performance

The algorithm can be summarized as a three step process. The initial step requires two linear passes over a string collection σ_i of total size N_i, which can be done in $O(N_i)$ steps. While performing these linear passes, the S* substrings are stored in a collection to create Σ_{i+1} of size k_{i+1}. These substrings are then sorted in $O(k_{i+1} * \log(k_{i+1}))$ steps prior to performing the second linear pass. The second step is to recursively compute the BWT for a string collection that is at most half the size of the current collection, $N_{i+1} \leq \frac{N_i}{2}$. The third step is to calculate the offsets into the BWT for each S* substring suffix. Using the modified `Induce(...)` method, this requires $O(k_{i+1})$ steps (note that $k_{i+1} \leq N_i$). Finally, a linear pass using B_{i+1} and the offsets allows the output BWT B_i to be

S* substring suffix	Frequency	Offset into B_0
\$GA	1	0
\$TA	1	1
AGC	2	2
CG\$	1	4
CT\$	1	5
G\$	1	6
GA	1	7
GC	2	8
T\$	1	10
TA	1	11

Table 5.5: S* substring suffixes - level 0. This table shows the sorted list of S* substring suffixes for $\sigma_0 = \{\$TAGCT, \$GAGCG\}$. Note that all suffixes of length 1 are excluded because they are represented elsewhere in the suffix list. The offset for a suffix is calculated as the sum of all frequencies of S* substring suffixes that lexicographically precede it. Most suffixes in this example occur once, but the S* substring suffixes “AGC" and “GC" both occur twice, once in each string.

Index, x	$B_1[x]$	B_0
init	–	[?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?]
0	3	[?, ?, ?, ?, G, ?, C, ?, ?, ?, ?, ?]
1	4	[?, ?, ?, ?, G, G, C, ?, ?, ?, C, ?]
2	0	[G, ?, ?, ?, G, G, C, \$, ?, ?, C, ?]
3	1	[G, T, ?, ?, G, G, C, \$, ?, ?, C, \$]
4	2	[G, T, G, ?, G, G, C, \$, A, ?, C, \$]
5	2	[G, T, G, T, G, G, C, \$, A, A, C, \$]

Table 5.6: Induced BWT - level 0. This table shows how each symbol in B_1 is used to induce multiple symbols in B_0. During each iteration x, the S* substring corresponding to symbol $B_1[x]$ and the offsets for each S* substring suffix are extracted and stored. Then, the corresponding predecessor symbol for each offset is written to output BWT B_0. Initially, the entire BWT is unknown. The first symbol encountered in B_1 is ‘3’, corresponding to S* substring “CG\$" in σ_1. This S* substring has two suffixes [“G\$", “CG\$"] corresponding to offsets $[6, 4]$ in Table 5.5. Thus, index 6 gets set to ‘C’ (the symbol preceding “G\$") and index 4 gets set to ‘G’ (the symbol preceding “CG\$"). This process is repeated for $x = [1..5]$ to finish solving for B_0.

written in $O(N_i)$ steps.

In summary, the total cost for level i is $T(N_i) = N_i + k_{i+1} * \log(k_{i+1}) + T(\frac{N_i}{2})$. However, the modifed `Induce(...)` function can actually be used to perform an implicit sort of the S* substrings. This subroutine requires $O(N_i)$ steps in the worst case (every substring is unique), leading to a total runtime of $T(N_i) = N_i + T(\frac{N_i}{2})$ which is $O(N_i)$. Note that in practice, we found the time to perform the explicit sort of the S* substrings (using a built-in C++ `std::sort`) was much faster than the implicit sort calculation performed by the modified `Induce(...)` function because k_{i+1} tends to be much smaller than N_i for the test datasets.

5.4 Results

5.4.1 BWT construction tools

Our implementation of MSBWT-IS is in C++ and is publicly available[1]. It currently does not rely on any external libraries. We compiled the program using g++ with options `g++ -O3 -std=c++11`.

We compare MSBWT-IS to the ropebwt2 tool first mentioned in Section 5.1 [Li, 2014]. Recall that this method is a variant of the BCR algorithm that performs “column-wise" insertions. While the algorithm is reported as $O(N * \log(N))$ for a string collection with N symbols, the BCR class of algorithm performs worse as the length of reads increases simply because the number of “columns" increases with it. The ropebwt2 implementation tackles this issue by keeping the partial BWT in memory as a B+-tree, allowing for faster modification of the structure and reducing the cost from repeatedly allocation of space for the BWT arrays. In general, this method is quite fast, outperforming both earlier BCR implementations and the merge-based construction algorithm of Chapter 4. Ropebwt2 is implemented in C and was compiled using the instructions available online.

For ropebwt2, we actually tested two versions of their output. The first version requires a pre-sort of the strings in the collection, but builds a BWT that is functionally identical to the one produced by MSBWT-IS. In our results, this version is labeled as `ropebwt2 -LR + sort` to reflect the options and that the cost of running the sort is included. The second version does not require a pre-sort, but it stores both the forward and reverse-complement of all strings in the collection. This allows

[1]https://github.com/holtjma/msbwt-is

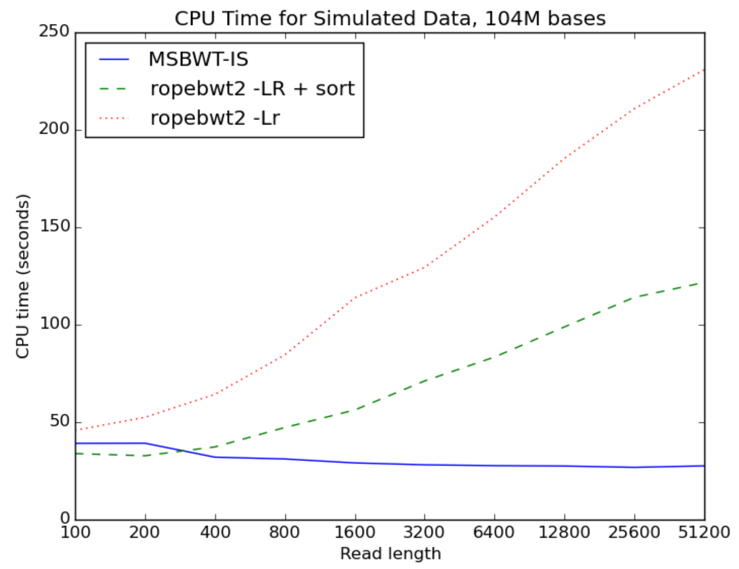
downstream analyses to perform single queries to retrieve both forward and reverse counts as a single value. However, the forward and reverse-complements are joined together in the implicit suffix array such that it is impossible to tell which string was the original without additional information. Additionally, this BWT can require twice as much space because it is storing twice as many strings. The second version is labeled as `ropebwt2 -Lr`. In general, both versions of the ropebwt2 algorithm are expected to get slower as the length of the strings increases due to its "column-wise" insertion. Additionally, the added time caused by sorting the strings is expected to increase with both the number and length of the strings in the collection. The ropebwt2 tool also calculates the BWT in parallel. In our experience, it utilizes approximately 2-3 threads on average using this parallelization.

All tests were run on a machine running Ubuntu 14.04 with 32 GB memory and an Intel Xeon E5-2620 6-core 2.00 GHz processor. The machine is connected to a 1 TB HDD for reading and writing any necessary input or output files. For measuring performance, we used the built-in `/usr/bin/time` function to extract real time, user time, memory usage, and CPU utilization. For performing the pre-sort required by ropebwt2, we used the built-in `/usr/bin/sort` function.

5.4.2 Simulated datasets

For simulated datasets, we tested the effect of read length on each method. To do this, we generated random "genome" strings and then sampled reads from those strings at approximately 100x coverage with a 1% error rate. To maintain a constant number of input bases, the read length was varied through repeated doublings while simultaneously halving the number of reads. Our first, smaller batch of datasets started with 2^{20} 100-basepair reads and went to 2^{11} 51200-basepair reads, so each dataset had approximately 104 megabases of simulated data. Our second, larger batch of datasets started with 2^{24} 100-basepair reads and went to 2^{15} 51200-basepair reads, so each dataset had approximately 1.7 gigabases of simulated data. For each dataset, we ran MSBWT-IS and both versions of ropebwt2.

The CPU and wall clock times for the smaller batch of datasets are shown in Figure 5.3. As expected, the CPU time required by ropebwt2 grows with the read length. Since the `ropebwt2 -Lr` version encodes both forward and reverse-complement strings, it requires approximately twice as much CPU time as `ropebwt2 -LR`. In contrast to both ropebwt2 types, the CPU time required by the MSBWT-IS algorithm is actually decaying slightly as the read length increases because there are fewer end-of-string symbols ('\$') to encode. As a result, MSBWT-IS requires far less CPU time for



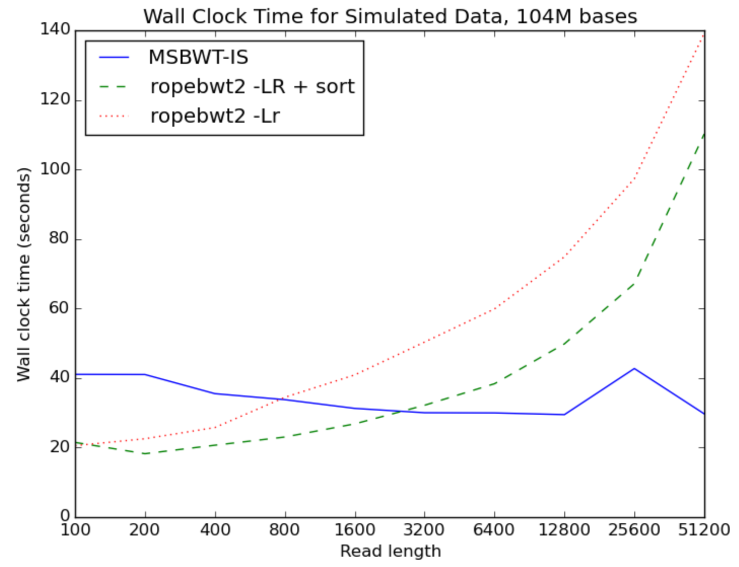


Figure 5.3: Performance - small simulated datasets. These figures show the CPU time (top) and wall clock time (bottom) as measured by `/usr/bin/time` for the three different methods of BWT construction for long reads. Datasets were generated by first randomly generating a “genome" string and then sampling reads from that string at approximately 100x coverage with a 1% mismatch rate. Multiple datasets were generated by varying the read length while keeping a constant number of genomic bases across all datasets (approximately 104 mega-bases per dataset). Each consecutive dataset has half the number of reads but double the read length. In general, the CPU time for MSBWT-IS decays slightly as the read length increases. This is caused by a reduction in total symbols because there are fewer end-of-string characters as the number of reads decreases. In contrast, both versions of ropebwt2 require increasing CPU time as the length of the reads increases. This performance pattern is reflected in the wall clock time. However, the ropebwt2 wall clock times are shifted downward relative to the MSBWT-IS times due to parallelization. Finally, the `ropebwt2 -Lr` option is almost always slower than `ropebwt2 -LR` because it is encoding twice as many symbols (both forward and reverse complement strings).

long reads than both versions of ropebwt2. The CPU usage of all three methods is reflected in the wall clock time. The wall clock time of MSBWT-IS is roughly the same as the CPU time because it is not parallelized. In contrast, the wall clock times of ropebwt2 tend to be 2-3x faster due to multi-processing.

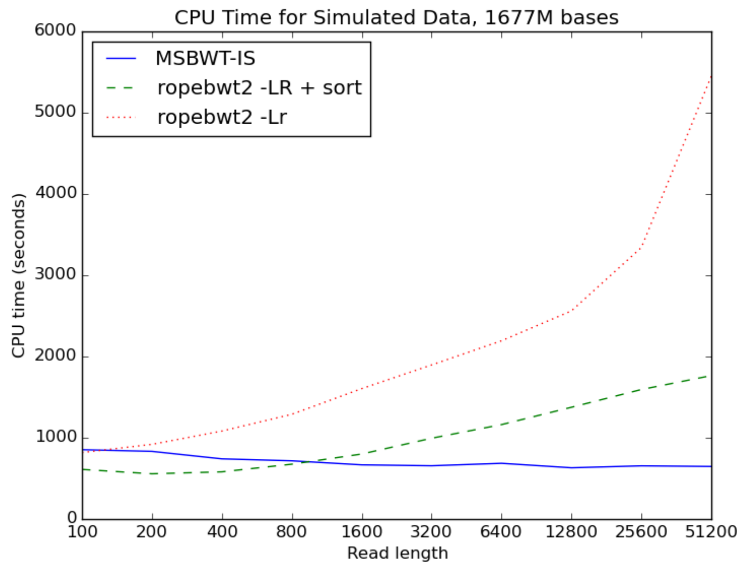
The CPU and wall clock times for the larger batch of datasets are shown in Figure 5.4. In general, the same trends from the small datasets are evident in the large datasets. The CPU time of MSBWT-IS is decaying slightly as read length increases while both versions of ropebwt2 require more CPU time. Additionally, the ropebwt2 wall clock times are roughly 2-3x less than their CPU times due to multi-processing. Given these initial tests, the expectation is for MSBWT-IS to require less CPU time than ropebwt2 when the read length is relatively long.

5.4.3 Long-read datasets

The next tests were performed on several publicly available PacBio datasets. In general, PacBio reads are thousands of basepairs long but have a relatively high error rate. The selected datasets include three *E. coli* K12 MG1655 datasets[2,3], one *P. falciparum* 3d7 dataset[4], one *N. crassa* OR74A dataset[5], one *S. cerevisiae* dataset[6], one *C. elegans* dataset[7], and one *A. thaliana* P5C3 dataset[8].

Table 5.7 shows the CPU and wall clock time for all tested long-read datasets for both MSBWT-IS and the two methods of ropebwt2. In all tests, MSBWT-IS required less CPU time than both versions of ropebwt2. Additionally, `ropebwt2 -Lr` used approximately double the amount of CPU time as `ropebwt2 -LR + sort` because it encodes both the forward and reverse-complement sequences. In general, `ropebwt2 -LR` required less wall clock time than MSBWT-IS due to parallelization. However, MSBWT-IS required less wall clock time than `ropebwt2 -LR` for the two test cases with the longest average read lengths (*C. elegans* and *Arabidopsis* P5C3). Additionally, these two datasets had large differences in CPU time, again suggesting that MSBWT-IS is less affected by long reads than ropebwt2.

[2]https://github.com/PacificBiosciences/DevNet/wiki/Datasets
[3]https://github.com/PacificBiosciences/DevNet/wiki/E-coli-K12-MG1655-Resequencing
[4]https://figshare.com/articles/Plasmodium_3D7_Genome_Assembly_With_PacBio_Data_and_CLEAR_/712587
[5]https://github.com/PacificBiosciences/DevNet/wiki/Neurospora-Crassa-(Fungus)-Genome,-Epigenome,-and-Transcriptome
[6]https://github.com/PacificBiosciences/DevNet/wiki/Saccharomyces-cerevisiae-W303-Assembly-Contigs
[7]https://github.com/PacificBiosciences/DevNet/wiki/C.-elegans-data-set
[8]https://github.com/PacificBiosciences/DevNet/wiki/Arabidopsis-P5C3



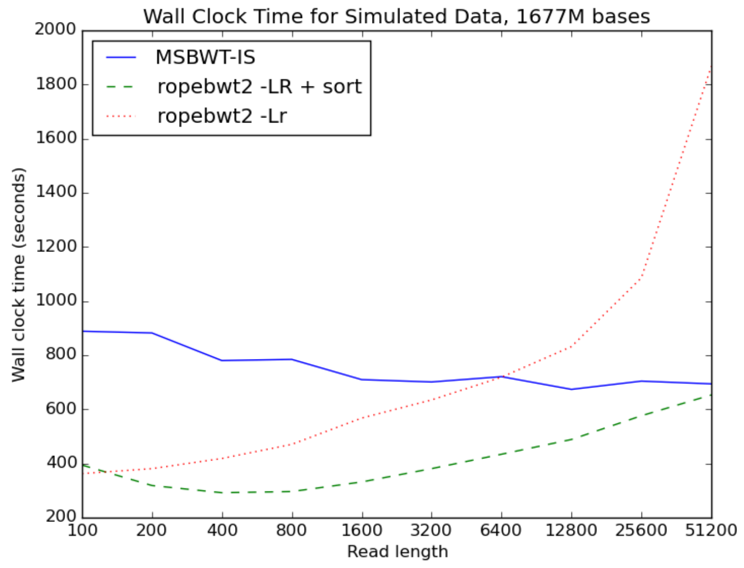


Figure 5.4: Performance - large simulated datasets. These figures show the CPU time (top) and wall clock time (bottom) as measured by `/usr/bin/time` for the three different methods of BWT construction for long reads. Datasets were generated by first randomly generating a “genome" string and then sampling reads from that string at approximately 100x coverage with a 1% mismatch rate. Multiple datasets were generated by varying the read length while keeping a constant number of genomic bases across all datasets (approximately 1677 mega-bases per dataset). Each consecutive dataset has half the number of reads but double the read length. The same patterns from the small datasets (see Figure 5.3) persist in these larger simulations. Both CPU and wall clock time required by both versions of ropebwt2 is growing with read length whereas they are decaying slightly with MSBWT-IS.

Dataset	Bases	Average length	MSBWT-IS	ropebwt2 -LR + sort	ropebwt2 -Lr
E. coli MG1655 CLR	98213822	1934	71.48 (87.54)	151.44 (65.39)	316.97 (119.84)
E. coli MG1655 CCS	217871193	940	63.48 (67.68)	237.48 (93.5)	452.25 (160.5)
E. coli k12 re-seq.	436994771	5278	323.51 (349.63)	392.89 (152.51)	1048.34 (327.48)
P. falciparum 3d7	630045976	2595	437.29 (455.77)	485.0 (197.36)	1422.0 (503.53)
N. crassa OR74A	981884113	5581	813.35 (863.34)	922.52 (345.35)	2008.75 (677.26)
S. cerevisiae W303	1307390784	6030	1025.7 (1091.48)	1153.83 (467.61)	3658.49 (1379.77)
C. elegans	2586974111	13127	844.81 (911.96)	2138.94 (921.28)	4333.18 (1653.7)
A. thaliana P5C3	2692706178	14609	791.62 (823.13)	2205.07 (948.18)	4379.72 (1649.87)

Table 5.7: Long-read dataset performance. This table shows the performance of MSBWT-IS and the two versions of ropebwt2 on long-read sequencing datasets ordered by their total number of bases. For each method, the CPU and wall clock time was measured using `/usr/bin/time`. The CPU time in seconds is shown for each test with the wall clock time in seconds in parentheses. In all test cases, MSBWT-IS requires less CPU time than `ropebwt2 -LR`. However, `ropebwt2 -LR` tend to beat MSBWT-IS in wall clock time because the implementation is parallelized. However, for the two datasets with the longest reads, MSBWT-IS requires less than 40% of the CPU time of `ropebwt2 -LR` and uses less wall clock time despite ropebwt2's parallelization. `ropebwt2 -Lr` uses more CPU time than the others because it is encoding twice as many symbols.

Dataset	Average length	MSBWT-IS	ropebwt2 -LR + sort	ropebwt2 -Lr
Mouse transcriptome b38	1707	75.01 (96.47)	132.35 (60.87)	260.06 (102.06)
Human genome b38	7053376	1439.61 (1642.67)	2971.04 (3096.37)	8467.9 (8660.0)
Mouse genome b38	123888075	2059.48 (2206.4)	3023.82 (3128.13)	6817.27 (6965.0)

Table 5.8: Genome dataset performance. This table shows the performance of MSBWT-IS and the two version of ropebwt2 on genome sized datasets. For each method, the CPU and wall clock time was measured using `/usr/bin/time`. The CPU time in seconds is shown for each test with the wall clock time in seconds in parentheses. In all three test cases, MSBWT-IS requires less CPU time than both versions of ropebwt2. Additionally, ropebwt2 does not parallelize when there are few strings as in the genome datasets. As a result, MSBWT-IS requires less wall clock time than ropebwt2 for both genome datasets.

5.4.4 Genome and transcriptome datasets

The last set of tests were run on genome sized datasets. Specifically, we ran tests on the mouse transcriptome, mouse genome, and human genome. In contrast to sequencing datasets, these datasets are expected to have less redundancy and the strings tend to be millions of basepairs long. As expected, MSBWT-IS outperforms both versions of ropebwt2 in CPU time. Additionally, since there are few strings in the genome datasets, ropebwt2 does not parallelize the computation. As a result, MSBWT-IS requires less wall clock time than ropebwt2 for the genome datasets.

5.5 Final notes

MSBWT-IS is an induced sorting based algorithm intended to compute the BWT of a string collection. The asymptotic run time of the algorithm for a dataset with N_i bases is $O(N_i)$. In our current implementation, the run time is instead represented recursively as $T(N_i) = N_i + k_{i+1} * \log(k_{i+1}) + T(\frac{N_i}{2})$ because we explicit sort the S* substrings, indicating that the run-time is affected by the total number of bases in the dataset and the number of different S* substrings in the dataset. In our experience, N_i is the driving term of this function. However, our tests show some evidence that k_{i+1} (the size of the next alphabet) can affect run-time. For example, the two *E. coli* datasets represent two different types of PacBio sequencing. CLR has approximately a 15% error rate, so k_{i+1} is relatively large. In contrast, CCS has approximately a 1% error rate, so k_{i+1} is relatively small. For these small datasets, this difference has a major impact on performance. In our MSBWT-IS tests, the *E. coli* CCS dataset required less CPU time than the *E. coli* CLR dataset despite the fact that this particular CCS dataset had twice the number of bases as the CLR dataset, indicating that the k_{i+1} term is impacting performance.

While MSBWT-IS tends to require less CPU time than ropebwt2, it does not always outperform it in wall clock time. This is primarily due to a lack of parallelization in the implementation of the algorithm. Fortunately, many parts of the algorithm can be adapted for parallelization. There are already C++ implementations of hash maps and sorting algorithms that operate in parallel with some amount of blocking when necessary. The main issue with parallelization is the calculation of the BWT while exiting the recursion. While sections of the BWT B_{i+1} can be linearly traversed in parallel, it is not obvious how to pre-calculate offsets for each individual section such that there are no collisions in the writing of B_i. Additionally, further optimization may be possible by exploring alternative methods for storing the S* substring collection or making more adaptations to the `Induce(...)` subroutine of Okanohara and Sadakane [2009].

MSBWT-IS tends to perform better on long strings than other BWT construction algorithms. In particular, BCR class implementations such ropebwt2 do not scale well with read length. Parallelization helps to improve their performance in practice, but fails to outperform MSBWT-IS when the reads become sufficiently long. In addition to working on strings of variable length, MSBWT-IS is not restricted to any particular type of string. In our test cases, we restricted the algorithm to an initial genomic alphabet with six symbols, one of which was the end-of-string symbol. However, adapting the implementation to allow for larger initial alphabets is trivial since the implementation already has to handle different alphabets at higher levels of recursion. Thus MSBWT-IS is a general algorithm for constructing BWTs of string collections with any type of alphabet.

CHAPTER 6

LONG READ CORRECTION VIA FM-INDEX QUERIES

While there are many sequencing technologies, most sequencing datasets can be categorized as either a short-read dataset or a long-read dataset. The reads in a short-read dataset (i.e. Illumina) are usually hundreds of basepairs long, of uniform lengths, and have a relatively low error rate (1%). In constrast, the reads in a long-read dataset (i.e. PacBio, nanopore) are usually thousands of basepairs, of variable length, and have a relatively high error rate (15%). Despite the high error rate, long-read DNA-seq datasets are useful because they provide evidence for contiguous sequence that spans low complexity or repeat regions of the genome. These repeat regions are too difficult to assemble through with short reads because the length of the region is usually longer than the length of the read, leading to ambiguities in the assembly. As a result, many researchers have looked into correcting the sequencing errors in long reads prior to their use in downstream analysis [Chin et al., 2013, Au et al., 2012, Koren et al., 2012, Salmela and Rivals, 2014].

In this chapter, we introduce a new method called FM-index Long Read Corrector (FMLRC) that corrects long reads using the BWT and FM-index of a short-read sequencing dataset as an implicit de Bruijn graph [Bruijn, 1946]. Since the FM-index can search for substrings of any length, our method is not constrained to a single fixed k-mer size, so it represents all possible de Bruijn graphs for the short-read sequencing data up to the read length. Additionally, the BWT is a lossless encoding of the short reads, allowing any pruning threshold to be dynamically adjusted without needing to reconstruct an entire de Bruijn graph. Our method is unique in that it applies both a short k-mer and long K-mer de Bruijn graph to the correction process, allowing for the correction algorithm to correct through low complexity regions up to the size of the long K-mer.

6.1 Related work

Long read correction algorithms can be broadly classified as self-correcting algorithms or hybrid correction algorithms. Self-correcting algorithms correct long reads using only the long read sequencing. Self-correcting algorithms, like HGAP [Chin et al., 2013], align the long reads to each other and generate a consensus sequence to perform the correction. In order to generate an accurate consensus, this method requires high-coverage sequencing of the long reads to overcome the high error rate. Unfortunately, the cost per basepair sequenced is relatively high for long-read sequencing technologies.

Hybrid correction algorithms instead use short-read sequencing to correct the long reads. The two main benefits of hybrid correction are that the short-read sequencing has fewer sequencing errors and the cost per basepair sequenced is lower than long-read sequencing technologies. Some example hybrid correction algorithms are LSC [Au et al., 2012], PacBioToCA [Koren et al., 2012], and LoRDEC [Salmela and Rivals, 2014]. Of these three methods, LoRDEC is the most recent, achieving comparable or better results than LSC and PacBioToCA while using less computational resources [Salmela and Rivals, 2014].

LoRDEC is a method for correcting long reads individually using a low-error, short-read sequencing dataset [Salmela and Rivals, 2014]. LoRDEC first generates a de Bruijn graph composed of k-mers (k length substrings) from the short reads [Salmela and Rivals, 2014]. Then, the graph is pruned such that any low-frequency k-mers (specified by a user-defined threshold) are removed from the graph. Long reads are then compared against this graph and broken into regions that are labeled as either *solid* or *weak*. All k-mers from the long read that are within solid regions are contained in the pruned short-read de Bruijn graph. All k-mers within weak regions are not in the de Bruijn graph. In general, the assumption of LoRDEC is that weak regions are caused by errors in sequencing and should be replaced with the closest series of solid, overlapping k-mers from the de Bruijn graph. When weak *inner* regions are identified in a long read, the flanking solid k-mers are used as endpoints for finding a *bridge* (or path) in the de Bruijn graph to connect the two solid regions. In the event of multiple supported bridges, the one with the closest edit distance to the original sequence is chosen. The *head* and *tail* of the graph are symmetrical special cases where there is only one flanking solid region. In either case, LoRDEC searches for the best extension of the single solid region with the smallest edit distance to the weak head or tail sequence. Since each long read is corrected individually (no comparison to other long reads is performed), LoRDEC trivially parallelizes by distributing the reads amongst all available processors.

LoRDEC has been shown to be one of the fastest and most accurate methods for performing long read correction using a short-read sequencing dataset [Salmela and Rivals, 2014]. However, during the de Bruijn graph construction, the user must select a fixed, short k-mer size and a fixed threshold for pruning. These options are often chosen heuristically and changing them requires re-computing the entire de Bruijn graph prior to re-running the correction algorithm. Additionally, when k-mers are relatively short, fixed-size de Bruijn graph methods often fail to correct regions of low complexity or repeated sequence that are longer than k basepairs. In the de Bruijn graph, low complexity sequences tend to look like "hairballs" of interconnected nodes where there are too many possible paths to explore. When LoRDEC enters a low complexity region, it will usually fail to find a path because it reaches a self-imposed limit on the graph exploration [Salmela and Rivals, 2014].

FMRC is a method for self-correcting short-read datasets using the BWT and FM-index [Greenstein et al., 2015]. First, FMRC builds the BWT and FM-index of the short-read dataset. Then, it corrects each short-read individually and in parallel. The k-mer frequency for each k-mer in the read is counted using the FM-index. If a low-frequency, "untrusted" k-mer is discovered, it greedily looks for an alternate "trusted" k-mer [Greenstein et al., 2015]. If an alternate k-mer is found, it replaces it in the read and re-calculates the k-mer frequencies. This is done repeatedly until all k-mers are considered "trusted" or until no more changes can be made. While this method works fairly well for short reads, the greedy approach does not scale well to long reads because of the repeated k-mer frequency recalculation and the increased error rate in the reads (there are more k-mers to modify). FMRC does allow for the k-mer size to be adjusted by the user without needing to recalculate the BWT and FM-index, but it uses a fixed, user-defined threshold for marking k-mers as "trusted".

6.2 Approach

The FM-index Long Read Corrector (FMLRC) is a hybrid correction method that uses the BWT and FM-index of a short-read sequencing dataset to correct a long-read sequencing dataset. FMLRC can be broken down into a two-pass algorithm. The first pass traverses an implicit k-mer de Bruijn graph constructed from the short reads and corrects the long reads using that graph. The second pass is computationally identical to the first pass, but it uses a longer K-mer de Bruijn graph for its correction method.

Prior to either pass, FMLRC builds a BWT and FM-index of the short reads to be used as an implicit de Bruijn graph [Bruijn, 1946]. Note that these data structures are used in both passes of the algorithm because they are not fixed to a particular k-mer length or threshold. Each long read is individually broken into *solid* and *weak* regions. *Solid* regions are composed of k-mers supported by the implicit de Bruijn graph at a threshold and *weak* regions are not supported at that threshold. Given two solid regions with a single weak region in between, the algorithm searches for solid *bridges* in the de Bruijn graph that connect the solid regions and span the weak regions. If no bridges are found, no change is made to the long read. If one or more bridges are found, they are aligned to the original weak region, and the one with the smallest edit distance is chosen to replace it. Weak regions at the beginning or end of the long read are treated similarly, but with a single-ended assembly as opposed to a bridging method. This first pass at correction is almost identical to the method described by LoRDEC [Salmela and Rivals, 2014]. In our results, we demonstrate the performance using our short k-mer pass and using LoRDEC as a pre-process instead.

The key difference in FMLRC is a second pass over each long read using a longer K-mer. The main reason FMLRC can use a long K-mer is because the BWT and FM-index implicitly represents all de Bruijn graphs for the short-read sequencing dataset. The identification of solid and weak regions is repeated as before, but with the long K and a different pruning threshold. Finally, weak regions are replaced with solid bridges or paths from the K-mer de Bruijn graph.

6.3 Correction of long read sequences using BWTs

The key difference in FMLRC is that the correction is performed using both a short k-mer and a long K-mer. Our method assumes that a BWT and FM-index of the short reads has been constructed. Since a BWT and FM-index is not restricted to any particular k-mer length, it acts as an implicit un-pruned de Bruijn graph for both k and K such that the frequency of any k-mer or K-mer in the dataset can be looked up in respectively $O(k)$ or $O(K)$ steps (see Section 2.3.2).

For our method, the two passes are programmatically identical with the value of k or K passed in as a parameter. For brevity, we describe the FMLRC correction using parameter k noting that replacing k with K describes the second pass of our method. Additionally, all thresholds and parameters are described as functions of k and the implicit k-mer de Bruijn graph.

6.3.1 The BWT and implicit De Bruijn graph

As described in Section 2.4.2, the BWT and FM-index can be used to implicitly represent all de Bruijn graphs for a string collection. When a node in any of the graphs is accessed, it performs a k-mer search in $O(k)$ steps using the `getRange(...)` function of Section 2.3.2. This informs FMLRC of the k-mer frequency in the dataset. FMLRC then compares the k-mer frequency to a threshold and classifies as *weak* if it is below the threshold and *strong* if it is above the threshold.

In this regard, the BWT and FM-index are advantageous data structures because values such as k-mer size and the threshold distinguishing solid and weak can be dynamically adjusted before or during the correction process without needing to explicitly recompute the full de Bruijn graph. Our method constructs the BWT once for short read sequencing data and uses the property of the BWT de Bruijn graph to allow for any value of k, K, or the pruning threshold t.

6.3.2 Identification of weak regions

Given a long read and a BWT of the short reads, the first step is to identify weak regions of the long read. To do this, our method uses a hybrid threshold that is a function of k-mer length and k-mer frequency in surrounding solid k-mers. First, the method retrieves k-mer counts for every k-mer in the long read. For most long reads, the majority of k-mer counts are at or near zero due to the high error rate from the sequencing technology. Second, every k-mer with a count that is less than a user-defined, absolute minimum threshold, T, is removed from the list of counts. Next, the median, m, of the remaining counts (that are all $\geq T$) is calculated. A second user-defined parameter, F, is the fraction of the median m that is required for a path to be consider solid. Finally, the solid threshold t for a particular read is calculated as $t = \max(T, F * m)$. In other words, the method calculates a dynamic threshold based on k-mer counts from that particular read. However, if that dynamic threshold is below an absolute, user-defined minimum, the absolute minimum is used as the threshold instead. For low-coverage short-read datasets, it is often the case that $t = T$ because $F * m < T$. For high-coverage short-read datasets, this dynamic threshold alleviates the need to select a fixed threshold beforehand, and it instead uses counts from the implicit de Bruijn graph to derive an expected count for k-mers in the read. Given the read-specific threshold t, weak regions are identified as contiguous weak k-mers (the frequency of each k-mer is $< t$). Similarly, solid regions are contiguous solid k-mers (the frequency of each k-mer is $\geq t$).

6.3.3 Correction of weak regions

Weak regions can be flanked by zero, one, or two solid regions. If a weak region has no flanking solid regions, the entire read is one large weak region with no solid k-mers to initialize a traversal of the de Bruijn graph. As a result, these reads are simply written to the output with no changes.

If a weak region has one flanking solid region, then it is either the head or tail weak region for the read. In either case, the solid k-mer closest to the weak region is used as the initializer k-mer for traversing the de Bruijn graph. The de Bruijn graph is explored from this start k-mer using a depth-first traversal and an expected path length based on the distance to the end of the read. Additionally, our method enforces a limit on the amount of branching that is allowed to reduce computation time. This branch limit is usually only reached if the algorithm is traversing a complex section of the de Bruijn graph where many interconnected nodes lead to an exponential number of graph traversals. If no paths are found or the branch limit is reached, then no change is made to the read. If one or more paths are found, then they are each aligned to the original weak head/tail region and the one with the smallest edit distance replaces the original.

Finally, the most common case is a weak region flanked by two solid regions. In this case, an initial starting k-mer is selected from the first solid region and a target k-mer is selected from the second. The de Bruijn graph is explored in a depth-first traversal from the starting k-mer. Thus, the algorithm attempts to find a path in the de Bruijn graph that bridges the two solid regions and spans the weak region. Each bridge is extended so long as it is shorter than an expected length that is estimated from the distance between the two solid k-mers in the long read. Whenever a bridge is found that ends with the target k-mer, it is added to a list of discovered bridges. Similar to the head and tail traversal, the bridge traversals also have a branch limit to reduce the amount of computation. If no bridges are found or the branch limit is reached, then no change is made to the weak region. If one or more bridges are found, they are each aligned to the original weak region and the one with the smallest edit distance is used to replace the original.

6.3.4 Differences in the short and long passes

FMLRC is a two pass algorithm using first a short k-mer followed by a second longer K-mer. In general, the short k-mer pass does the majority of the correction for the method. While the k-mers are described as “short", they are still long enough to uniquely identify most regions of the genome. If a region can be uniquely identified using short k-mers, then correction is often a relatively easy process and a longer K-mer is not required. This is why many other k-mer based methods are able to correct the majority of errors in long reads.

In our method, the short k-mer pass is similar to other previously described methods, specifically LoRDEC [Salmela and Rivals, 2014]. The main differences are that we dynamically choose the pruning threshold t and the branch limit based on k/K. Additionally, there are some cases where we use different starting and target k-mers in the de Bruijn graph traversal. One disadvantage of the previously described methods is that they are restricted to a single k value associated with their de Bruijn graph. Additionally, some de Bruijn graphs are limited to short k-mers because of the underlying implementation. However, these other correction methods can be used in place of the first k-mer based pass of the algorithm. Thus, the full pipeline could be described as a short k-mer correction method (such as LoRDEC) followed by our long K-mer correction pass. In our results, we test FMLRC as both a two-pass method and as a one-pass long K-mer method following a different short k-mer correction method.

To the best of our knowledge, the long K-mer pass is unique to our method. To provide some intuition behind why it improves the results, we focus on the differences in de Bruijn graphs representing the same data but with two distinct k-mer lengths. In general, two distinct paths will be merged in a k-mer de Bruijn graph if they share a pattern that is at least k long. This is because the nodes along that shared region will be identical. At the ends of the shared region, there will be two paths emerging representing the differences at the edge of the shared regions. For example, Figure 6.1 shows an example de Bruijn graph using short 3-mers to represent two strings.

When the same sequences are viewed through a longer K-mer de Bruijn graph, the number of merged, ambiguous paths strictly decreases because an increasing amount of similarity is required for the paths to become merged in the graph. To demonstrate this idea, Figure 6.2 shows the 5-mer de Bruijn graph for the same two strings from Figure 6.1. Note that while traversal of the graph for 3-mers was relatively difficult due to branching and cycles, the 5-mer graph is easily traversed because it is two disjoint linear paths. In practice, the short k-mer is still long enough to uniquely identify most areas of the genome. However, genomic characteristics such as low-complexity sequence, gene families, or repeat regions are difficult to traverse using short k-mers. Thus, our method uses the larger K-mer to bridge weak regions composed of repeated or low-complexity sequences that are too difficult to traverse using a small k-mer.

Since our method uses two passes with different sizes of k and K, it allows for less branching when k is small and more branching when K is large. As we just described, a small k-mer de Bruijn graph will have more branches and may require more computation to do a full depth-first traversal in repeat regions. To avoid this, we place more restrictions on the short k-mer graph traversals such that it primarily fixes the “easy" errors caused by sequencing. As a result, the “harder" graph traversals are pushed off to the long K-mer de Bruijn graph. The longer K-mer de Bruijn graph is less likely to have many branches and more likely to have a single pass connecting a start and target

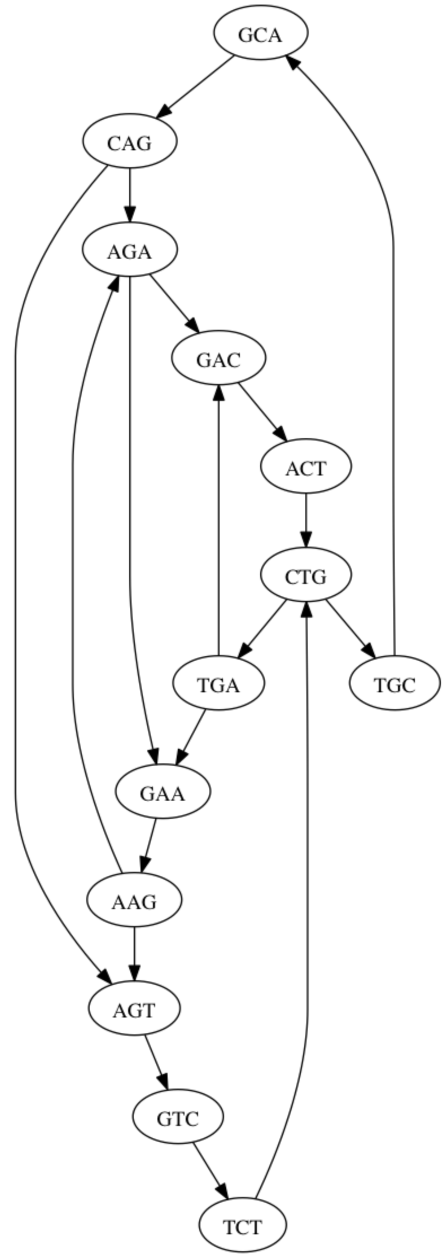


Figure 6.1: Example k-mer graph. This figure shows the 3-mer de Bruijn graph for two strings: “CAGACTGCAG" and “CAGTCTGAAG". Every 3-mer from the two strings is a node in the graph. An edge indicates that the 2-mer suffix of the source 3-mer is the same as the 2-mer prefix of the destination 3-mer. In this graph, the two distinct strings are not obvious because the k-mer is relatively small. The graph contains many cycles, and the start and end k-mers for each string are obfuscated because of the structure. Algorithms that traverse this graph would need to explore every path in hopes of finding a bridge connecting two specific k-mers. This can lead to long computations or failure to find bridges if a branching limit is imposed on the traversal method.

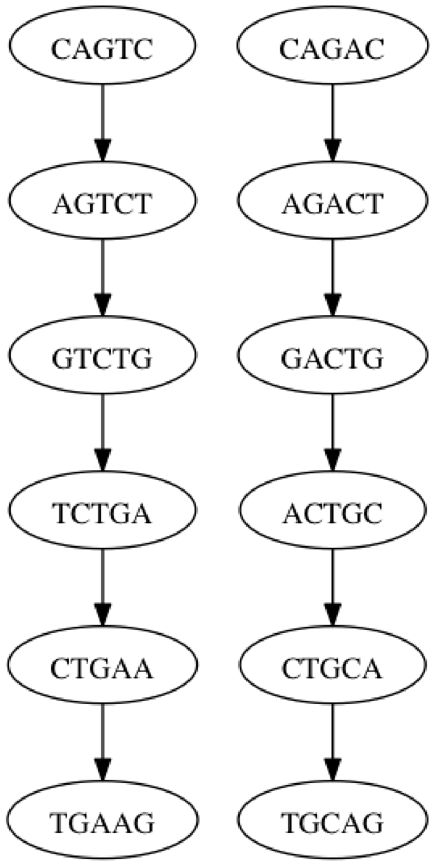


Figure 6.2: Example K-mer graph. This figure shows the 5-mer de Bruijn graph for two strings: "CAGACTGCAG" and "CAGTCTGAAG". Every 5-mer from the two strings is a node in the graph. An edge indicates that the 4-mer suffix of the source 5-mer is the same as the 4-mer prefix of the destination 5-mer. In contrast to the 3-mer graph of Figure 6.1, the two distinct paths in this 5-mer graph are obvious because there are no shared 5-mers between the two strings. Algorithms that traverse this graph would find connecting K-mers to be a comparatively simple operation because there is no branching or cycles in the graph.

K-mer. In order to enable this approach, our method calculates a branch limit that scales linearly with the selected k/K value for the pass.

6.4 Results

6.4.1 Correction and alignment tools

For all test cases, we used FMLRC v0.1.2 which is a publicly available C++ program[1]. The implementation assumes that a BWT of the short-read dataset is constructed as a pre-process, and that it is in the run-length encoded format of the *msbwt* package[2] (see Appendix A). FMLRC includes two FM-index implementations. The default FM-index implementation is a highly sampled FM-index that enables fast k-mer lookups at the cost of a larger memory footprint. The second FM-index implementation is a traditional sampled FM-index that allows users to set the sampling rate, leading to longer computations with a smaller memory footprint. The two FM-index implementations produce identical corrected read results.

We compare FMLRC to the LoRDEC v0.6 hybrid correction method [Salmela and Rivals, 2014] first described in Section 6.1. Recall that this method corrects reads using a short k-mer, pruned de Bruijn graph. For all tests, we ran LoRDEC with the options `-k 21 -s 5` indicating a k-mer of length 21 and that any k-mer with frequency less than 5 is pruned from the de Bruijn graph. We also compared to a combined approach where LoRDEC is used for the short k-mer correction and FMLRC is used for the long K-mer correction.

After running each correction method, we aligned the resulting FASTA file to the corresponding reference genome for the organism using BLASR v2.0.0 [Chaisson and Tesler, 2012]. For BLASR, the only extra parameter we used was `-hitPolicy randombest` to force the alignment to keep only the best mapping for each read. Note that BLASR will also split reads into multiple subreads if it detects multiple subreads that best align to different genomic locations. All corrections and alignments were executed on the Killdevil computing cluster located at University of North Carolina at Chapel Hill.

[1]https://github.com/holtjma/fmlrc
[2]https://github.com/holtjma/msbwt

Dataset	**Illumina**			**PacBio**		
	Average Length	**Median Length**	**Total Bases**	**Average Length**	**Median Length**	**Total Bases**
E. coli k12	100	100	2842864800	5279	4578	436994771
P. falciparum 3d7	135	135	10549043505	2596	2262	630045976
S. cerevisiae W303	300	300	15113671800	6030	5112	1307390784
D. melanogaster	75	75	5204863650	9310	8005	15744118797
A. thaliana P5C3	213	150	11034657110	3068	1528	11529818309

Table 6.1: Data summary. Summary of the data used for the experiments in this chapter.

6.4.2 Data description

We tested the correction algorithms on five publicly available PacBio datasets. The PacBio datasets were downloaded for one *E. coli* K12 MG1655 datasets[3], one *P. falciparum* 3d7 dataset[4], one *S. cerevisiae* dataset[5], one *A. thaliana* P5C3 dataset[6], and one *D. melanogaster* dataset[7]. For each dataset, we also downloaded a publicly available short-read sequencing dataset (many from GenBank [Benson et al., 2008]): a publicly available dataset[8] for *E. coli*, SRR1503358 for *P. falciparum*, a publicly available dataset[9] for *S. cerevisiae*, SRR1652473 for *A. thaliana*, and SRR1104304 for *D. melanogaster*. Note that the short-read sequencing datasets are from the same strain as its long-read sequencing dataset, but they are not necessarily from the same sample. However, this does not impact the results since we only compared hybrid correction methods.

6.4.3 FMLRC parameter selection

FMLRC allows for four main parameters to be defined by the user. The first two are the T and F parameters described in Section 6.3.2. T is the absolute minimum frequency required for a k-mer to be considered solid in the de Bruijn graph. F is the fraction of the median counts required for a k-mer to be considered solid in the de Bruijn graph. For all test cases, we heuristically chose $T = 5$ and $F = .10$.

The second two parameters are the choice of k and K for the short and long correction passes. If a k size is too long, it will be difficult to find due to the increased likelihood of an error in the k-mer, but if it is too short, the k-mer may occur in multiple contexts and lead to complex graph traversals.

[3]https://github.com/PacificBiosciences/DevNet/wiki/E-coli-K12-MG1655-Resequencing
[4]https://figshare.com/articles/Plasmodium_3D7_Genome_Assembly_With_PacBio_Data_and_CLEAR_/712587
[5]https://github.com/PacificBiosciences/DevNet/wiki/Saccharomyces-cerevisiae-W303-Assembly-Contigs
[6]https://github.com/PacificBiosciences/DevNet/wiki/Arabidopsis-P5C3
[7]https://github.com/PacificBiosciences/DevNet/wiki/Drosophila-sequence-and-assembly
[8]http://spades.bioinf.spbau.ru/spades_test_datasets/ecoli_mc/
[9]http://labshare.cshl.edu/shares/schatzlab/www-data/nanocorr/2015.07.07

Matching Bases						
	K					
k	–	49	59	69	79	89
17	357689915	364261034	365911374	366756300	366397703	364348529
19	360879691	371679728	372242631	372053470	371238554	369385323
21	364324790	373222105	**373389445**	373122774	372396464	370679046
23	365894380	373163803	373345702	373138050	372519218	371088777
25	366929792	373126530	373247984	373037934	372565125	371123308

Edit Distance						
	K					
k	–	49	59	69	79	89
17	16988141	7368593	6323839	5722348	5509356	6120257
19	8012854	4341121	4145344	4071010	4091550	4446711
21	6341729	3685543	3596552	3577941	3626889	3952585
23	5940344	3523463	**3439920**	3443619	3536317	3886421
25	5977754	3532665	3455707	3490784	3629262	4041437

Percent Matching						
	K					
k	–	49	59	69	79	89
17	96.2561%	98.3792%	98.5830%	98.7011%	98.7337%	98.5771%
19	98.1107%	99.0143%	99.0575%	99.0710%	99.0633%	98.9746%
21	98.5083%	99.1579%	99.1812%	99.1854%	99.1742%	99.0935%
23	98.6225%	99.1967%	99.2180%	99.2169%	99.1959%	99.1129%
25	98.6340%	99.1991%	**99.2193%**	99.2109%	99.1798%	99.0845%

Table 6.2: Choosing k and K. This table shows the result of running FMLRC using many different values for k and K for an *E. coli* dataset. The test cases with $K = -$ indicate that no second pass of correction using the long K-mer was performed, so those test cases use a single pass short k-mer only. After correcting the reads, we aligned the results using BLASR and gathered statistics on the alignments. Matching bases indicates the number of matching bases across all mappings. Edit distance indicates the total number of mismatches, insertions, and deletions across all mappings. Percent matching is calculated as the total number of matching bases divided by the total number of bases that aligned. For each statistic, the best result is bolded in the above table. To summarize, a $K = 59$ is the column containing the test cases with the largest matching bases, lowest edit distance, and highest percent matching. Additionally, all tested values of K for a long K-mer pass improves the results over a single k-mer pass.

To gain some insight into what values of k and K are optimal, we ran multiple tests using the *E. coli* K12 MG1655 dataset. We allowed $k = [17, 19, 21, 23, 25]$ and $K = [-, 49, 59, 69, 79, 89]$, leading to a total of 30 test cases. The test cases with $K = -$ indicate that no second K-mer pass was performed (it is only using a one-pass, short k-mer for correction). For each test case, we ran FMLRC, aligned the corrected reads to the reference genome, and then gathered statistics on the resulting alignment. We counted the the total number of bases that matched the reference genome and the total edit distance of the mappings. We also calculated the percentage of bases that matched the reference genome by taking the total number of matching bases and dividing by the total number of bases in the corrected reads. The results of this experiment are shown in Table 6.2.

For each statistic measured, the best result used $K = 59$, but three different values for k. In all of our tests, performing a second pass with the long K-mer always improved all three statistics. Additionally, these results show that using a K-mer that is too large can have a negative impact on the performance of the correction. If the size of the K-mer relative to the read is too large, then the number of nodes in the de Bruijn graph will be reduced leading to difficulties during graph traversal. In our tests, while $K = 59$ is usually the best, $K = 49$ or $K = 69$ generally have similar performances. In contrast, the difference in edit distance from 69 to 79 and from 79 to 89 is more dramatic, indicating that there may be too few 79-mers and 89-mers with enough coverage to correct the reads. It is worth noting that the "best" k and K is likely data-dependent because differences in coverage, sequencing quality, and sequencing content will impact the ability of FMLRC to find solid k-mers and perform corrections.

6.4.4 Alignment-based results

For each dataset, we selected k and K based on the length and coverage of reads in the short-read sequencing datasets. For all test cases, we used $k = 21$ for both LoRDEC and FMLRC. For K, we used 59 for the three organisms with smaller genomes ($<$ 25 million basepairs) and 69 for the two datasets with larger genomes ($>$ 100 million basepairs). We tested LoRDEC(k), LoRDEC(k) combined with FMLRC(K), and FMLRC(k, K) correction methods. For each corrected read collection, we aligned the reads using BLASR [Chaisson and Tesler, 2012] to the corresponding reference genome for the organism. We also aligned the uncorrected sequence data to the reference genome for comparison. Finally, we gathered the following statistics on each alignment: number of bases that match the reference, edit distance of the read mappings to the reference genome, and the percentage of matching bases out of the total number of aligned bases. A summary of the results is shown in Table 6.3.

In all test cases, either LoRDEC(k)+FMLRC(K) or FMLRC(k, K) had the lowest edit distance and highest percent matching. With the exception of the *D. melanogaster* test cases, LoRDEC(k)+FMLRC(K) or FMLRC(k, K) also had the most matching bases. With the exception of *D. melanogaster*, the second pass of FMLRC(K) strictly improved the results of LoRDEC(k) by increasing the number of matching bases, decreasing the edit distance of aligned subreads, and increasing the percent matching. This suggests that performing the second large K-mer pass is improving the results.

Dataset	**Method (k)**	**Matching Bases**	**Edit Distance**	**Percent Matching**
E. coli k12	Uncorrected	336443541	54379832	89.54
	LoRDEC (21)	345874813	11273571	97.16
	LoRDEC (21) + FMLRC (59)	370965658	3947279	99.07
	FMLRC(21, 59)	**373389445**	**3596552**	**99.18**
P. falciparum 3d7	Uncorrected	52606809	13240402	82.05
	LoRDEC (21)	54026517	11107774	84.89
	LoRDEC (21) + FMLRC (59)	68239267	**8203978**	90.64
	FMLRC(21, 59)	**72308728**	8582093	**90.87**
S. cerevisiae W303	Uncorrected	1036918138	166258621	89.93
	LoRDEC (21)	1093308817	48999753	96.55
	LoRDEC (21) + FMLRC (59)	**1123474843**	24007596	98.35
	FMLRC(21, 59)	1121693505	**22675840**	**98.56**
D. melanogaster	Uncorrected	12634631139	1954342890	90.06
	LoRDEC (21)	**13129572976**	777750172	95.86
	LoRDEC (21) + FMLRC (59)	13055410406	**755524218**	**95.98**
	FMLRC(21, 59)	12877419670	963162596	95.04
A. thaliana P5C3	Uncorrected	15365943113	17587675791	87.37
	LoRDEC (21)	16446004357	1856138182	92.44
	LoRDEC (21) + FMLRC (59)	16943619873	**1646252130**	**93.50**
	FMLRC(21, 59)	**17187480187**	1816781841	93.09

Table 6.3: Long read correction results. This table shows the alignment statistics for the long read correction methods. We gathered statistics on the uncorrected reads, reads corrected using only LoRDEC, reads corrected using a combination of LoRDEC and FMLRC, and reads corrected using only FMLRC. For each set of reads, we aligned the results using BLASR to the corresponding reference genome for the dataset. For each alignment, we counted the total number of mapped bases that match the reference genome, the total edit distance of all mapped reads, and the percentage of matching bases out of the total number of mapped bases. In all test cases, either LoRDEC+FMLRC or FMLRC had the lowest edit distance and highest percent matching. Additionally, with the exception of the *D. melanogaster* tests, LoRDEC+FMLRC or FMLRC had the most matching bases as well.

6.4.5 Performance

We selected the three smallest datasets for performance tests on a single machine. These tests were run on a machine running Ubuntu 14.04 with 32 GB memory and an Intel Xeon E5-2620 6-core 2.00 GHz processor. The machine is connected to a 1 TB HDD for reading and writing any necessary input or output files. For measuring performance, we used the built-in `/usr/bin/time` function to extract real time, user time, memory usage, and CPU utilization. Each correction method was allowed eight processes for computing the results.

We collected real time, CPU time, and maximum memory consumption for LoRDEC(k), FMLRC(K), FMLRC(k, K), and FMLRC-mem(k, K). FMLRC(K) is a single pass using only a long K-mer that is run on the result of LoRDEC(k). FMLRC(k, K) is the default implementation of the two-pass algorithm described in this chapter. FMLRC-mem(k, K) is a second implementation of FMLRC that is designed to have a smaller memory footprint than the default FMLRC, but at the cost of longer run-time. Note that FMLRC and FMLRC-mem produce identical results. For FMLRC-mem, we used extra parameters `fmlrc -i -F 8` to enable the alternate indexing scheme. The results of these tests are shown in Table 6.4.

In all test cases, FMLRC(k, K) requires much less CPU time than LoRDEC(k) but much more memory. FMLRC-mem(k, K) requires more CPU than either FMLRC(k, K) or LoRDEC(K), but also has a much smaller memory footprint than FMLRC(k, K). FMLRC(K) requires less wall clock and CPU time than any other test case because it is only executing a single K-mer correction pass. Note that the maximum memory usage of FMLRC(k, K) and FMLRC(K) are essentially identical because the vast majority of the memory is used for storing the short-read BWT and FM-index structures and those structures do not change with the size of k or K.

6.5 Final notes

In this chapter, we introduced FMLRC, a long read correction method that uses the BWT and FM-index as an implicit de Bruijn graph. The method uses two passes to perform the correction: one with a relatively short k-mer and one with a longer K-mer. In each pass, the method searches for weak, unsupported regions in the reads and uses the implicit de Bruijn graph to identify alternate solid paths to correct the reads. We compared the results of FMLRC to a similar algorithm LoRDEC for a variety of PacBio datasets. Additionally, we tested the combination of LoRDEC for the short k-mer and FMLRC for the long K-mer pass to see how FMLRC improves the results of

Dataset	Method (k)	Real time (h:mm:ss)	CPU time (seconds)	Max Memory (GB)
E. coli k12	LoRDEC (21)	1:10:41	32169.27	1.250
	FMLRC(59)	10:40	4996.62	3.851
	FMLRC(21, 59)	32:58	15685.06	3.851
	FMLRC-mem(21, 59)	5:04:07	145666.26	1.046
P. falciparum 3d7	LoRDEC (21)	1:29:09	38706.87	2.288
	FMLRC(59)	29:51	13972.41	13.412
	FMLRC(21, 59)	1:22:39	39300.30	13.412
	FMLRC-mem(21, 59)	7:08:23	205122.77	2.768
S. cerevisiae W303	LoRDEC (21)	5:58:32	162536.71	2.442
	FMLRC(59)	2:36:31	74303.81	19.874
	FMLRC(21, 59)	3:27:08	98394.55	19.593
	FMLRC-mem(21, 59)	26:02:58	748559.43	4.698

Table 6.4: Long read correction performance. This table shows the real time, CPU time, and maximum memory usage of the correction methods on three datasets that were run on a single machine and measured using `/usr/bin/time`. Each method was allowed eight processes for correction. LoRDEC was run once using a single k-mer. FMLRC was run once using a single long K-mer. FMLRC was run twice using a short k-mer and a long K-mer, but with different underlying FM-index implementations. The default implementation is designed to use less CPU time at the cost of a higher memory footprint. The second implementation is labeled with "FMLRC-mem" because it is designed to use less memory at the cost of longer run-times. FMLRC requires less CPU usage than LoRDEC but much more memory. FMLRC-mem tends to require more CPU than either FMLRC or LoRDEC, but also has a much lower memory footprint when compared to FMLRC.

LoRDEC. In all test cases, we showed that using a long K-mer pass improved the results of LoRDEC. Additionally, there were some test cases where FMLRC(k, K) outperformed the combination of LoRDEC(k)+FMLRC(K).

We introduced two different implementations of FMRLC that use different underlying indexing structures. The default implementation requires far less CPU and wall clock time than LoRDEC at the cost of a larger memory footprint. The second implementation is slower but requires less memory. One area for future research is changing the indexing structure of FMLRC either by improving the existing implementations or using an entirely new indexing structure that may be faster and/or more memory efficient. Additionally, there is some evidence that the short k-mer pass of FMLRC does not correct as well as LoRDEC. This is likely caused by our use of a simpler, less comprehensive bridging algorithm. Future work will seek to enhance the performance of this bridging algorithm to improve the result.

Another area of future work is to further explore the choice of k and K in FMLRC. In our results, we chose the values for k and K heuristically based on tests run on a single dataset. However, the optimal k and K for a particular dataset likely depends on several data-dependent factors such as genome size, coverage of the short-read sequencing, and error rate of both short and long-read sequencing datasets. Additionally, the optimal k and K may vary from read to read within a single dataset due to genomic events like duplicated sequence. In such cases, a dynamically chosen k or K value may lead to better results than selecting two static values.

CHAPTER 7

BWT-BASED WEB TOOLS

In Chapter 4 and 5, we described algorithms that efficiently construct BWTs for short and long read sequencing datasets. The BWT data structure stores the sequencing data in a compressed format, and the FM-index enables fast k-mer searches into a BWT. For genomic data, this means there are methods to calculate k-mer frequencies and extract specific reads from the BWT. Despite the utility of the BWT, there are few interfaces for accessing the stored information. Most of the BWT-based tools available require extensive knowledge of BWTs through an API or are command line tools that provide limited and specific functionality. In either case, the results are usually difficult to interpret without additional processing. Additionally, we are unaware of any visualization tools that retrieve individual raw sequence data from the BWT.

In this chapter, we discuss several web tools that bridge this gap through web-based visualizations. The primary purpose of these tools is to provide access to and search in the raw sequence data in order to identify and highlight biologically significant patterns. The tools hide the details of BWTs and FM-index based queries while still enabling the users to access and explore the data in a way that other sequence analysis approaches (such as alignment) do not allow.

7.1 Related work

There are many online and offline tools specifically for visualizing sequencing data. Many of these tools are require the data to be pre-processed prior to visualization. Tools like the Integrative Genomic Viewer (IGV) [Thorvaldsdóttir et al., 2013] or Savant [Fiume et al., 2010] visualize reads that have been aligned to a reference sequence. Once aligned, the user can navigate to regions of the alignment to see read mappings, pileup counts (the number of reads that mapped to a particular location), sequencing errors/genomic variants, etc. As a result, these tools help highlight features of the alignment that are not easily noticed through the SAM alignment format.

Tablet [Milne et al., 2010] offers similar navigational visualization, but for sequence assembly formats instead of alignments. The tool visualizes the assembly by displaying the reads against the assembled contigs. Note that this is conceptually similar to displaying the aligned reads against a reference genome.

For both the alignment and assembly based methods, the data is indexed into a new format such that any accesses are performed by knowing the location in the new format (i.e. chromosome and position in an alignment). As a result, any data that did not get indexed (i.e. did not align) is simply not included in the visualization. Depending on the accuracy of the pre-processing method, this can lead to large amounts of data that are simply lost to the user. Additionally, genomic variants that exist in the data but not in the reference genome can be more difficult to align. In the presence of multiple alleles, the allele included in the reference will be easier to align, and alleles not included in the reference will be biased against due to the difficulty of aligning it.

While not a sequencing dataset visualizer, BLAT [Kent, 2002] is worth mentioning because of its sequence-centric approach to visualizing a sequence alignment. The actual BLAT tool was created for alignment, but there is a web tool associated with BLAT for querying long sequences in a reference genome. The tool returns a list of candidate alignment locations that allow for some error in the alignment. Once a candidate is selected, the tool will then visualize the alignment along with some auxiliary information for the user. While BLAT does not visualize sequencing data like IGV or Tablet, it does provide a fast, sequence-centric approach for searching one or more reference genomes.

7.2 Approach

The BWT web tools described in this chapter take a sequence-centric approach to accessing the sequencing data. The BWTs and FM-indices of sequencing datasets are pre-computed and stored on a web server. Since the data is stored as a BWT, there is no loss of sequenced reads like in alignment-based approachs. Instead, any biases come from the selection of the query sequence used during a lookup with the FM-index.

When a user requests information from a web tool, the request is sent to the server. Even when compressed, the BWTs can be very large on the server's local disk. To return results quickly, it memory maps the BWT and FM-index such that only pages necessary for the query are loaded into memory. Once the query has been processed, the server returns any relevant information the client needs to display the results. The information is then visualized using a combination of HTML for static interfaces and Javascript for dynamic interfaces.

There are four classes of tool discussed below. The first class retrieves all reads with a specific k-mer and then displays that data on screen. The second class searches for batches of k-mers and primarily retrieves k-mer frequency information. The third class incorporates annotated reference genome information to generate k-mer pileups for specific genes or sections of the reference genome. The fourth class visualizes the data as a collapsed de Bruijn graph, enabling users to perform targeted, user-guided assemblies.

7.3 Data description

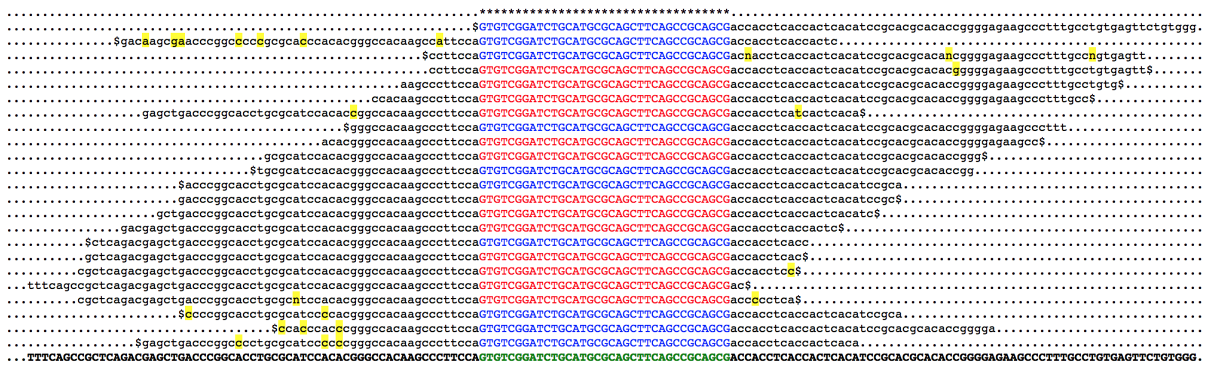
Throughout this chapter, we will make reference to many mouse sequencing datasets generated by other studies. Each set of raw sequence data was used to generate a BWT and FM-index to be accessed by the web tools. Mice are diploid organisms, meaning they have two copies of each chromosome. Many of these datasets are inbred mice, meaning that the two copies are nearly identical. Because they are inbred, we generally expect only one version of each allele (genetic variation such as a single nucleotide polymorphism, insertion, or deletion) to occur within each dataset. Some of the sequencing datasets are of F1 hybrid mice resulting from a cross between two inbred mice. As a result, we expect to see two different alleles for these mice in areas where the the two parental inbreds differ from each other. For this chapter, we primarily use mouse DNA-seq datasets from Keane et al. [2011] and mouse RNA-seq datasets from Crowley et al. [2015].

7.4 k-mer based query tools

The BWT and FM-index allow for arbitrary length k-mers to be queried in $O(k)$ steps. Additionally, a string can be retrieved from the BWT in $O(L)$ steps where L is the length of the read being retrieved. The first two web tools are based on using these two functions to search for a user provided k-mer and then retrieve all reads containing that k-mer.

With either k-mer tool, the user selects one or more datasets and provides the k-mer to search for. The server then opens the BWT for each dataset and performs the k-mer search. This returns a range of indices in the BWT that can then be used to retrieve each read containing the particular k-mer. Once all of the reads for a dataset are extracted, they are then sent back to the client for visualization.

The first tool simply displays the results of the k-mer query for the user. In Figure 7.1, an example query on an inbred mouse DNA-seq dataset is shown. Each retrieved read is shown as a row in the



```
..................................................................***********************************..................................................................
.................................................................$GTGTCGGATCTGCATGCGCAGCTTCAGCCGCAGCGaccacctcaccactcacatccgcacgcacaccggggagaagccctttgcctgtgagttctgtggg.
...............$gacaagcgaacccggcccccgcgcacccacacgggccacaagccattccaGTGTCGGATCTGCATGCGCAGCTTCAGCCGCAGCGaccacctcaccactc...................................................
..........................................................$ccttccaGTGTCGGATCTGCATGCGCAGCTTCAGCCGCAGCGacnacctcaccactcacatccgcacgcacancggggagaagccctttgccngtgagtt........
...........................................................ccttccaGTGTCGGATCTGCATGCGCAGCTTCAGCCGCAGCGaccacctcaccactcacatccgcacgcacacggggagaagccctttgcctgtgagtt$.......
.......................................................aagcccttccaGTGTCGGATCTGCATGCGCAGCTTCAGCCGCAGCGaccacctcaccactcacatccgcacgcacaccggggagaagccctttgcctgtg$...........
...................................................ccacaagcccttccaGTGTCGGATCTGCATGCGCAGCTTCAGCCGCAGCGaccacctcaccactcacatccgcacgcacaccggggagaagccctttgcc$...............
...................gagctgacccggcacctgcgcatccacaccggccacaagcccttccaGTGTCGGATCTGCATGCGCAGCTTCAGCCGCAGCGaccacctcatcactcaca$...............................................
...............................................$gggccacaagcccttccaGTGTCGGATCTGCATGCGCAGCTTCAGCCGCAGCGaccacctcaccactcacatccgcacgcacaccggggagaagcccttt...................
............................................acacgggccacaagcccttccaGTGTCGGATCTGCATGCGCAGCTTCAGCCGCAGCGaccacctcaccactcacatccgcacgcacaccggggagaagcc$......................
....................................gcgcatccacacgggccacaagcccttccaGTGTCGGATCTGCATGCGCAGCTTCAGCCGCAGCGaccacctcaccactcacatccgcacgcacaccggg$..............................
..................................$tgcgcatccacacgggccacaagcccttccaGTGTCGGATCTGCATGCGCAGCTTCAGCCGCAGCGaccacctcaccactcacatccgcacgcacaccgg................................
........................$acccggcacctgcgcatccacacgggccacaagcccttccaGTGTCGGATCTGCATGCGCAGCTTCAGCCGCAGCGaccacctcaccactcacatccgca..........................................
........................gacccggcacctgcgcatccacacgggccacaagcccttccaGTGTCGGATCTGCATGCGCAGCTTCAGCCGCAGCGaccacctcaccactcacatccgc$..........................................
.....................gctgacccggcacctgcgcatccacacgggccacaagcccttccaGTGTCGGATCTGCATGCGCAGCTTCAGCCGCAGCGaccacctcaccactcacatc$.............................................
................gacgagctgacccggcacctgcgcatccacacgggccacaagcccttccaGTGTCGGATCTGCATGCGCAGCTTCAGCCGCAGCGaccacctcaccactc$..................................................
...........$ctcagacgagctgacccggcacctgcgcatccacacgggccacaagcccttccaGTGTCGGATCTGCATGCGCAGCTTCAGCCGCAGCGaccacctcacc.......................................................
...........gctcagacgagctgacccggcacctgcgcatccacacgggccacaagcccttccaGTGTCGGATCTGCATGCGCAGCTTCAGCCGCAGCGaccacctcac$.......................................................
..........cgctcagacgagctgacccggcacctgcgcatccacacgggccacaagcccttccaGTGTCGGATCTGCATGCGCAGCTTCAGCCGCAGCGaccacctcc$........................................................
...tttcagccgctcagacgagctgacccggcacctgcgcatccacacgggccacaagcccttccaGTGTCGGATCTGCATGCGCAGCTTCAGCCGCAGCGac$...............................................................
..........cgctcagacgagctgacccggcacctgcgcntccacacgggccacaagcccttccaGTGTCGGATCTGCATGCGCAGCTTCAGCCGCAGCGaccccctca$........................................................
........................$ccccggcacctgcgcatccccacgggccacaagcccttccaGTGTCGGATCTGCATGCGCAGCTTCAGCCGCAGCGaccacctcaccactcacatccgca..........................................
.....................................$ccacccacccgggccacaagcccttccaGTGTCGGATCTGCATGCGCAGCTTCAGCCGCAGCGaccacctcaccactcacatccgcacgcacaccgggga.............................
..................$gagctgacccggcccctgcgcatccccccgggccacaagcccttccaGTGTCGGATCTGCATGCGCAGCTTCAGCCGCAGCGaccacctcaccactcaca................................................
...TTTCAGCCGCTCAGACGAGCTGACCCGGCACCTGCGCATCCACACGGGCCACAAGCCCTTCCAGTGTCGGATCTGCATGCGCAGCTTCAGCCGCAGCGACCACCTCACCACTCACATCCGCACGCACACCGGGGAGAAGCCCTTTGCCTGTGAGTTCTGTGGG.
```

Figure 7.1: K-mer lookup tool. This is a visualization of all reads with a particular k-mer in an inbred mouse DNA-seq dataset. Every read is shown as a row with the exact matching k-mer shown in either red or blue. A red k-mer indicates that the k-mer was found exactly on a read. A blue k-mer indicates that the reverse complement sequence of the k-mer was found exactly on the read, so the entire read was reverse-complemented prior to displaying it. The reads are aligned such that the shared exact matching k-mers are in the same position horizontally. At the bottom is a bold sequence with the shared k-mer in green that represents the consensus sequence of all the reads. A base in this consensus is chosen programmatically by selecting the base with the highest frequency in the column. Any bases that do not match the consensus are highlighted in yellow. In this way, random errors show up as sporadic highlighted bases. In contrast, heterozygosity shows up as consistent highlighted bases in a column.

visualization. These reads have been horizontally shifted such that the shared k-mer from every read is aligned. If the k-mer is red, it indicates that the read contained the k-mer exactly on the forward version of read. If the k-mer is blue, it indicates that the read contained the reverse-complement k-mer, so the entire read was reverse-complemented for visualization. At the bottom of the display is a bolded sequence representing the consensus sequence of all retrieved reads. The consensus is generated by selecting the most prevalent base at each position. Bases from individual reads that do not match the consensus sequence are highlighted in yellow for easier identification by the user. Using this approach, random sequencing errors appears as sporadic highlighted bases whereas homologous sequences appear as consistent highlighted bases in a column in which consensus and alternate states are typically found at similar frequencies. Finally, the interface allows the user to select any contiguous sequence present in the results and then perform the query on that new k-mer.

The second tool is an adaptation of the first modified to aid in identifying the different sequence versions or alleles sharing a k-mer query. Figure 7.2 shows a single k-mer query where there is evidence of multiple alleles in the bases both before and after the query. This suggests that the chosen query is part of two distinct sequences that both occur within the inbred organism.

```
..................................................***************************************************..................................................
..................................................TTTCCACTACAGATGGTTGTGAGCCACCATGTGGTTGCTGGGAATTGAACTcaggacctctggaagagcagtcagtgctcttaaccactgagccatctct$
.................................................aTTTCCACTACAGATGGTTGTGAGCCACCATGTGGTTGCTGGGAATTGAACTcaggacctctggaagagcagtcagtgctcttaaccactgagccatctc$.
...............................................$gaTTTCCACTACAGATGGTTGTGAGCCACCATGTGGTTGCTGGGAATTGAACTcaggacctctggaagagcagtcagtgctcttaaccactgagccatct...
...............................................$gaTTTCCACTACAGATGGTTGTGAGCCACCATGTGGTTGCTGGGAATTGAACTcaggacctctggaagagcagtcagtgctcttaaccactgagccatct...
................................................gaTTTCCACTACAGATGGTTGTGAGCCACCATGTGGTTGCTGGGAATTGAACTcaggacctctggaagagcagtcagtgctcttaaccactgagccatct$..
...............................................agaTTTCCACTACAGATGGTTGTGAGCCACCATGTGGTTGCTGGGAATTGAACTcaggacctctggaagagcagtcagtgctcttaaccactgagccatc$...
...............................................agaTTTCCACTACAGATGGTTGTGAGCCACCATGTGGTTGCTGGGAATTGAACTcaggacctctggaagagcagtcagtgctcttaaccactgagccatc$...
.............................................$cagaTTTCCACTACAGATGGTTGTGAGCCACCATGTGGTTGCTGGGAATTGAACTcaggacctctggaagagcagtcagtgctcttaaccactgagccat.....
...........................................$catcagaTTTCCACTACAGATGGTTGTGAGCCACCATGTGGTTGCTGGGAATTGAACTcaggacctctggaagagcagtcagtgctcttaaccactgagc.........
...........................................catcagaTTTCCACTACAGATGGTTGTGAGCCACCATGTGGTTGCTGGGAATTGAACTcaggacctctggaagagcagtcagtgctcttaaccactgagc$........
........................................gggcatcagaTTTCCACTACAGATGGTTGTGAGCCACCATGTGGTTGCTGGGAATTGAACTcaggacctctggaagagcagtcagtgctcttaaccactg$..........
......................................gagggcatcagaTTTCCACTACAGATGGTTGTGAGCCACCATGTGGTTGCTGGGAATTGAACTcaggacctctggaagagcagtcagtgctcttaaccac$............
...............................$ccagaagagggcatcagaTTTCCACTACAGATGGTTGTGAGCCACCATGTGGTTGCTGGGAATTGAACTcaggacctctggaagagcagtcagtgctctt...................
.............................acaccagaagagggcatcagaTTTCCACTACAGATGGTTGTGAGCCACCATGTGGTTGCTGGGAATTGAACTcaggacctctggaagagcagtcagtgct$.....................
.......................tcagacacaccagaagagggcatcagaTTTCCACTACAGATGGTTGTGAGCCACCATGTGGTTGCTGGGAATTGAACTcaggacctctggaagagcagtc$...........................
......................ttcagacacaccagaagagggcatcagaTTTCCACTACAGATGGTTGTGAGCCACCATGTGGTTGCTGGGAATTGAACTcaggacctctggaagagcagt$............................
.....................cttcagacacaccagaagagggcatcagaTTTCCACTACAGATGGTTGTGAGCCACCATGTGGTTGCTGGGAATTGAACTcaggacctctggaagagcag$.............................
...................gacttcagacacaccagaagagggcatcagaTTTCCACTACAGATGGTTGTGAGCCACCATGTGGTTGCTGGGAATTGAACTcaggacctctggaagagc$..............................
...................gacttcagacacaccagaagagggcatcagaTTTCCACTACAGATGGTTGTGAGCCACCATGTGGTTGCTGGGAATTGAACTcaggacctctggaagagc$..............................
................actgacttcagacacaccagaagagggcatcagaTTTCCACTACAGATGGTTGTGAGCCACCATGTGGTTGCTGGGAATTGAACTcaggacctctggaag$..................................
..............atactgacttcagacacaccagaagagggcatcagaTTTCCACTACAGATGGTTGTGAGCCACCATGTGGTTGCTGGGAATTGAACTcaggacctctgga$....................................
............$tatactgacttcagacacaccagaagagggcatcagaTTTCCACTACAGATGGTTGTGAGCCACCATGTGGTTGCTGGGAATTGAACTcaggacctctgg......................................
.............tatactgacttcagacacaccagaagagggcatcagaTTTCCACTACAGATGGTTGTGAGCCACCATGTGGTTGCTGGGAATTGAACTcaggacctctgg$.....................................
.............tatactgacttcagacacaccagaagagggcatcagaTTTCCACTACAGATGGTTGTGAGCCACCATGTGGTTGCTGGGAATTGAACTcaggacctctgg$.....................................
.............tatactgacttcagacacaccagaagagggcatcagaTTTCCACTACAGATGGTTGTGAGCCACCATGTGGTTGCTGGGAATTGAACTcaggacctctgg$.....................................
.............tatactgacttcagacacaccagaagagggcatcagaTTTCCACTACAGATGGTTGTGAGCCACCATGTGGTTGCTGGGAATTGAACTcaggacctctgg$.....................................
.............tatactgacttcagacacaccagaagagggcatcagaTTTCCACTACAGATGGTTGTGAGCCACCATGTGGTTGCTGGGAATTGAACTcaggacctctgg$.....................................
.........$gagtatactgacttcagacacaccagaagagggcatcagaTTTCCACTACAGATGGTTGTGAGCCACCATGTGGTTGCTGGGAATTGAACTcaggacctc.........................................
.........tgagtatactgacttcagacacaccagaagagggcatcagaTTTCCACTACAGATGGTTGTGAGCCACCATGTGGTTGCTGGGAATTGAACTcaggacct$.........................................
.........tgagtatactgacttcagacacaccagaagagggcatcagaTTTCCACTACAGATGGTTGTGAGCCACCATGTGGTTGCTGGGAATTGAACTcaggacct$.........................................
.......$gtgagtatactgacttcagacacaccagaagagggcatcagaTTTCCACTACAGATGGTTGTGAGCCACCATGTGGTTGCTGGGAATTGAACTcaggacc...........................................
.....$atgtgagtatactgacttcagacacaccagaagagggcatcagaTTTCCACTACAGATGGTTGTGAGCCACCATGTGGTTGCTGGGAATTGAACTcagga.............................................
.....$atgtgagtatactgacttcagacacaccagaagagggcatcagaTTTCCACTACAGATGGTTGTGAGCCACCATGTGGTTGCTGGGAATTGAACTcagga.............................................
.....tatgtgagtatactgacttcagacacaccagaagagggcatcagaTTTCCACTACAGATGGTTGTGAGCCACCATGTGGTTGCTGGGAATTGAACTcagg$.............................................
...$gtatgtgagtatactgacttcagacacaccagaagagggcatcagaTTTCCACTACAGATGGTTGTGAGCCACCATGTGGTTGCTGGGAATTGAACTcag...............................................
...tgtatgtgagtatactgacttcagacacaccagaagagggcatcagaTTTCCACTACAGATGGTTGTGAGCCACCATGTGGTTGCTGGGAATTGAACTca$...............................................
.................................................gTTTCCACTACAGATGGTTGTGAGCCACCATGTGGTTGCTGGGAATTGAACTcaggacctctacaagagcagtcagtgctctgaatctctgagccatctc$.
.................................................gTTTCCACTACAGATGGTTGTGAGCCACCATGTGGTTGCTGGGAATTGAACTcaggacctctacaagagcagtcagtgctctgaatctctgagccatctc$.
.................................................gTTTCCACTACAGATGGTTGTGAGCCACCATGTGGTTGCTGGGAATTGAACTcaggacctctacaagagcagtcagtgctctgaatctctgagccatctc$.
........................................$gggtataggTTTCCACTACAGATGGTTGTGAGCCACCATGTGGTTGCTGGGAATTGAACTcaggacctctacaagagcagtcagtgctctgaatctctga..........
.......................................aagggtataggTTTCCACTACAGATGGTTGTGAGCCACCATGTGGTTGCTGGGAATTGAACTcaggacctctacaagagcagtcagtgctctgaatctct$...........
..............................ccaccagaaaagggtataggTTTCCACTACAGATGGTTGTGAGCCACCATGTGGTTGCTGGGAATTGAACTcaggacctctacaagagcagtcagtgctc$....................
..........................agacccaccagaaaagggtataggTTTCCACTACAGATGGTTGTGAGCCACCATGTGGTTGCTGGGAATTGAACTcaggacctctacaagagcagtcagt$........................
.....................$cttcagacccaccagaaaagggtataggTTTCCACTACAGATGGTTGTGAGCCACCATGTGGTTGCTGGGAATTGAACTcaggacctctacaagagcagt.............................
....................ctcttcagacccaccagaaaagggtataggTTTCCACTACAGATGGTTGTGAGCCACCATGTGGTTGCTGGGAATTGAACTcaggacctctacaagagca$..............................
............$tgctgctctcttcagacccaccagaaaagggtataggTTTCCACTACAGATGGTTGTGAGCCACCATGTGGTTGCTGGGAATTGAACTcaggacctctac......................................
...........actgctgctctcttcagacccaccagaaaagggtataggTTTCCACTACAGATGGTTGTGAGCCACCATGTGGTTGCTGGGAATTGAACTcaggacctct$.......................................
.........acactgctgctctcttcagacccaccagaaaagggtataggTTTCCACTACAGATGGTTGTGAGCCACCATGTGGTTGCTGGGAATTGAACTcaggacct$.........................................
....$gagtacactgctgctctcttcagacccaccagaaaagggtataggTTTCCACTACAGATGGTTGTGAGCCACCATGTGGTTGCTGGGAATTGAACTcagg..............................................
....tgagtacactgctgctctcttcagacccaccagaaaagggtataggTTTCCACTACAGATGGTTGTGAGCCACCATGTGGTTGCTGGGAATTGAACTcag$..............................................
..$gtgagtacactgctgctctcttcagacccaccagaaaagggtataggTTTCCACTACAGATGGTTGTGAGCCACCATGTGGTTGCTGGGAATTGAACTca................................................
............................$gacccccagaaagggtataggTTTCCACTACAGATGGTTGTGAGCCACCATGTGGTTGCTGGGAATTGAACTcaggacctctacaagagcagtcagtgc.......................
...................tctcttcagacccaccagaaaatggtataggTTTCCACTACAGATGGTTGTGAGCCACCATGTGGTTGCTGGGAATTGAACTcaggacctctacaagaga$...............................
...TTTATGTGAGTATACTGACTTCAGACACACCAGAAGAGGGCATCAGATTTCCACTACAGATGGTTGTGAGCCACCATGTGGTTGCTGGGAATTGAACTCAGGACCTCTGGAAGAGCAGTCAGTGCTCTTAACCACTGAGCCATCTCT.
```

Figure 7.2: *K*-mer lookup tool with multiple alleles. This is a visualization of all reads with a particular *k*-mer in an inbred mouse DNA-seq dataset. Every read is shown as a row with the exact matching *k*-mer shown in either red or blue. A red *k*-mer indicates that the *k*-mer was found exactly on a read. A blue *k*-mer indicates that the reverse complement sequence of the *k*-mer was found exactly on the read, so the entire read was reverse-complemented prior to displaying it. The reads are aligned such that the shared exact matching *k*-mers are in the same position horizontally. At the bottom is a bold sequence with the shared *k*-mer in green that represents the consensus sequence of all the reads. A base in this consensus is chosen programmatically by selecting the base with the highest frequency in the column. Any bases that do not match the consensus are highlighted in yellow. In this way, random errors show up as sporadic highlighted bases. In contrast, heterozygosity shows up as consistent highlighted bases in a column. In this example, there are many consistently highlighted bases in several reads near the bottom of the visualization, suggesting that this *k*-mer occurs in two distinct sequences from the inbred organism.

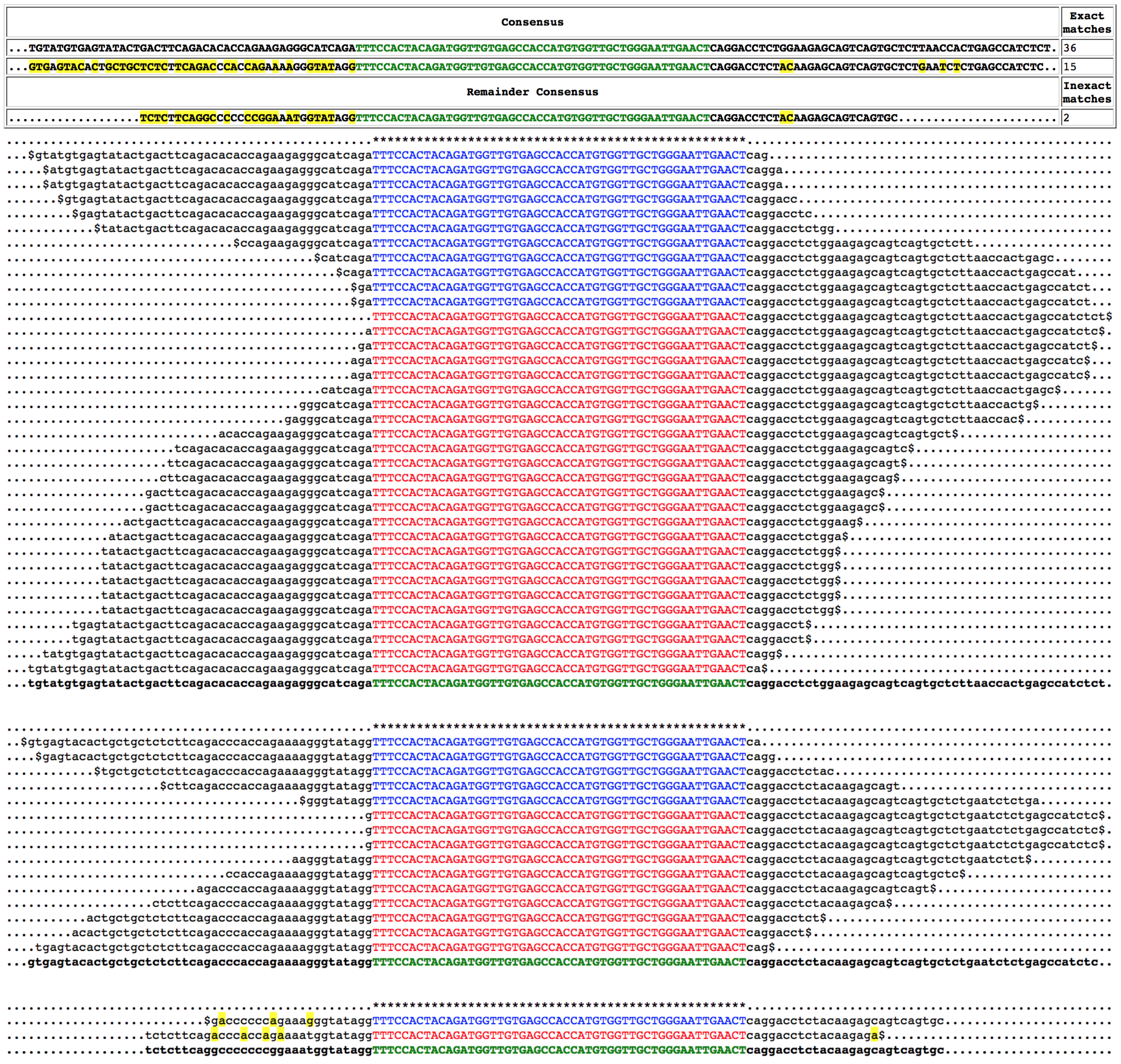
Figure 7.3 demonstrates the output of the allele-finding tool on the same dataset and k-mer query from Figure 7.2. After retrieving the reads, pairs of reads are scored against each other based on sequence overlap and number of matching bases. The closest pairs are then repeatedly merged into larger clusters such that all reads in a cluster exactly match their consensus. In other words, there is no disagreement on what the consensus should be for a particular cluster of reads. All clusters that have few reads are then merged into a remainder cluster where the reads vote on the consensus but are not required to match it. At the top of the visualization, a summary of the consensus sequences are displayed along with the number of reads in the corresponding cluster. Differences in each consensus from the largest cluster's consensus are highlighted in yellow. Below the summary, the reads of each cluster are displayed using the same visualization style of the original k-mer search.

These two tools enable arbitrary k-mer searches into multiple datasets and present the results to the user. By highlighting differences in the consensus sequence, the tools enable identification of random sequencing errors and/or consistent inconsistencies that may indicate genomic variation. Additionally, the allele-based tool helps the user to identify these alleles by clustering the reads prior to visualization. In both tools, the user has to know something about the sequence content of the datasets before performing the search. In particular, the choice of k-mer query is important because it requires the sequence to exactly match the query. As a result, variants may result in difficulties finding reads for a dataset.

In order to account for these off-target variants, there is another available tool that allow for k-mer searches with a user-specified edit distance, e. When an edit distance based query is requested, the server uses a branch-and-bound search routine to find any k-mers that are within e base changes, insertions, or deletions from the target sequence. Since this is a branch-and-bound search with an alphabet of length 4, the time to find all the k-mers with a particular edit distance $O(4^e * k)$. Once identified, the tool returns the k-mers and the reads corresponding to the k-mer. As with the first tool, these reads are then visually organized and a consensus sequence is generated for each edited k-mer.

7.5 Batch query tools

The k-mer based visualizations are useful for accessing raw sequence data surrounding a specific k-mer. However, there are often cases where the actual read context is not required. Instead, the user may only be concerned with the k-mer frequencies associated with a batch of queries. Typically,



Consensus	Exact matches
...TGTATGTGAGTATACTGACTTCAGACACACCAGAAGAGGGCATCAGATTTCCACTACAGATGGTTGTGAGCCACCATGTGGTTGCTGGGAATTGAACTCAGGACCTCTGGAAGAGCAGTCAGTGCTCTTAACCACTGAGCCATCTCT.	36
...GTGAGTACACTGCTGCTCTCTTCAGACCCACCAGAAAAGGGTATAGGTTTCCACTACAGATGGTTGTGAGCCACCATGTGGTTGCTGGGAATTGAACTCAGGACCTCTACAAGAGCAGTCAGTGCTCTGAATCTCTGAGCCATCTC..	15
Remainder Consensus	**Inexact matches**
...................TCTCTTCAGGCCCCCCCGGAAATGGTATAGGTTTCCACTACAGATGGTTGTGAGCCACCATGTGGTTGCTGGGAATTGAACTCAGGACCTCTACAAGAGCAGTCAGTGC.......................	2

```
..................................................***************************************************..................................................
...$gtatgtgagtatactgacttcagacacaccagaagagggcatcagaTTTCCACTACAGATGGTTGTGAGCCACCATGTGGTTGCTGGGAATTGAACTcag...............................................
.....$atgtgagtatactgacttcagacacaccagaagagggcatcagaTTTCCACTACAGATGGTTGTGAGCCACCATGTGGTTGCTGGGAATTGAACTcagga.............................................
.....$atgtgagtatactgacttcagacacaccagaagagggcatcagaTTTCCACTACAGATGGTTGTGAGCCACCATGTGGTTGCTGGGAATTGAACTcagga.............................................
.......$gtgagtatactgacttcagacacaccagaagagggcatcagaTTTCCACTACAGATGGTTGTGAGCCACCATGTGGTTGCTGGGAATTGAACTcaggacc...........................................
.........$gagtatactgacttcagacacaccagaagagggcatcagaTTTCCACTACAGATGGTTGTGAGCCACCATGTGGTTGCTGGGAATTGAACTcaggacctc.........................................
............$tatactgacttcagacacaccagaagagggcatcagaTTTCCACTACAGATGGTTGTGAGCCACCATGTGGTTGCTGGGAATTGAACTcaggacctctgg......................................
...............................$ccagaagagggcatcagaTTTCCACTACAGATGGTTGTGAGCCACCATGTGGTTGCTGGGAATTGAACTcaggacctctggaagagcagtcagtgctctt...................
..........................................$catcagaTTTCCACTACAGATGGTTGTGAGCCACCATGTGGTTGCTGGGAATTGAACTcaggacctctggaagagcagtcagtgctcttaaccactgagc........
.............................................$cagaTTTCCACTACAGATGGTTGTGAGCCACCATGTGGTTGCTGGGAATTGAACTcaggacctctggaagagcagtcagtgctcttaaccactgagccat.....
...............................................$gaTTTCCACTACAGATGGTTGTGAGCCACCATGTGGTTGCTGGGAATTGAACTcaggacctctggaagagcagtcagtgctcttaaccactgagccatct...
...............................................$gaTTTCCACTACAGATGGTTGTGAGCCACCATGTGGTTGCTGGGAATTGAACTcaggacctctggaagagcagtcagtgctcttaaccactgagccatct...
..................................................TTTCCACTACAGATGGTTGTGAGCCACCATGTGGTTGCTGGGAATTGAACTcaggacctctggaagagcagtcagtgctcttaaccactgagccatctct$
.................................................aTTTCCACTACAGATGGTTGTGAGCCACCATGTGGTTGCTGGGAATTGAACTcaggacctctggaagagcagtcagtgctcttaaccactgagccatctc$.
................................................gaTTTCCACTACAGATGGTTGTGAGCCACCATGTGGTTGCTGGGAATTGAACTcaggacctctggaagagcagtcagtgctcttaaccactgagccatct$..
...............................................agaTTTCCACTACAGATGGTTGTGAGCCACCATGTGGTTGCTGGGAATTGAACTcaggacctctggaagagcagtcagtgctcttaaccactgagccatc$...
...............................................agaTTTCCACTACAGATGGTTGTGAGCCACCATGTGGTTGCTGGGAATTGAACTcaggacctctggaagagcagtcagtgctcttaaccactgagccatc$...
...........................................catcagaTTTCCACTACAGATGGTTGTGAGCCACCATGTGGTTGCTGGGAATTGAACTcaggacctctggaagagcagtcagtgctcttaaccactgagc$.......
........................................gggcatcagaTTTCCACTACAGATGGTTGTGAGCCACCATGTGGTTGCTGGGAATTGAACTcaggacctctggaagagcagtcagtgctcttaaccactg$..........
......................................gagggcatcagaTTTCCACTACAGATGGTTGTGAGCCACCATGTGGTTGCTGGGAATTGAACTcaggacctctggaagagcagtcagtgctcttaaccac$............
.............................acaccagaagagggcatcagaTTTCCACTACAGATGGTTGTGAGCCACCATGTGGTTGCTGGGAATTGAACTcaggacctctggaagagcagtcagtgct$.....................
.......................tcagacacaccagaagagggcatcagaTTTCCACTACAGATGGTTGTGAGCCACCATGTGGTTGCTGGGAATTGAACTcaggacctctggaagagcagtc$...........................
......................ttcagacacaccagaagagggcatcagaTTTCCACTACAGATGGTTGTGAGCCACCATGTGGTTGCTGGGAATTGAACTcaggacctctggaagagcagt$............................
.....................cttcagacacaccagaagagggcatcagaTTTCCACTACAGATGGTTGTGAGCCACCATGTGGTTGCTGGGAATTGAACTcaggacctctggaagagcag$.............................
...................gacttcagacacaccagaagagggcatcagaTTTCCACTACAGATGGTTGTGAGCCACCATGTGGTTGCTGGGAATTGAACTcaggacctctggaagagc$...............................
...................gacttcagacacaccagaagagggcatcagaTTTCCACTACAGATGGTTGTGAGCCACCATGTGGTTGCTGGGAATTGAACTcaggacctctggaagagc$...............................
................actgacttcagacacaccagaagagggcatcagaTTTCCACTACAGATGGTTGTGAGCCACCATGTGGTTGCTGGGAATTGAACTcaggacctctggaag$..................................
..............atactgacttcagacacaccagaagagggcatcagaTTTCCACTACAGATGGTTGTGAGCCACCATGTGGTTGCTGGGAATTGAACTcaggacctctgga$....................................
.............tatactgacttcagacacaccagaagagggcatcagaTTTCCACTACAGATGGTTGTGAGCCACCATGTGGTTGCTGGGAATTGAACTcaggacctctgg$.....................................
.............tatactgacttcagacacaccagaagagggcatcagaTTTCCACTACAGATGGTTGTGAGCCACCATGTGGTTGCTGGGAATTGAACTcaggacctctgg$.....................................
.............tatactgacttcagacacaccagaagagggcatcagaTTTCCACTACAGATGGTTGTGAGCCACCATGTGGTTGCTGGGAATTGAACTcaggacctctgg$.....................................
.............tatactgacttcagacacaccagaagagggcatcagaTTTCCACTACAGATGGTTGTGAGCCACCATGTGGTTGCTGGGAATTGAACTcaggacctctgg$.....................................
.............tatactgacttcagacacaccagaagagggcatcagaTTTCCACTACAGATGGTTGTGAGCCACCATGTGGTTGCTGGGAATTGAACTcaggacctctgg$.....................................
.........tgagtatactgacttcagacacaccagaagagggcatcagaTTTCCACTACAGATGGTTGTGAGCCACCATGTGGTTGCTGGGAATTGAACTcaggacct$.........................................
.........tgagtatactgacttcagacacaccagaagagggcatcagaTTTCCACTACAGATGGTTGTGAGCCACCATGTGGTTGCTGGGAATTGAACTcaggacct$.........................................
.....tatgtgagtatactgacttcagacacaccagaagagggcatcagaTTTCCACTACAGATGGTTGTGAGCCACCATGTGGTTGCTGGGAATTGAACTcagg$.............................................
...tgtatgtgagtatactgacttcagacacaccagaagagggcatcagaTTTCCACTACAGATGGTTGTGAGCCACCATGTGGTTGCTGGGAATTGAACTca$...............................................
...tgtatgtgagtatactgacttcagacacaccagaagagggcatcagaTTTCCACTACAGATGGTTGTGAGCCACCATGTGGTTGCTGGGAATTGAACTcaggacctctggaagagcagtcagtgctcttaaccactgagccatctct.

..................................................***************************************************..................................................
..$gtgagtacactgctgctctcttcagacccaccagaaaagggtataggTTTCCACTACAGATGGTTGTGAGCCACCATGTGGTTGCTGGGAATTGAACTca................................................
....$gagtacactgctgctctcttcagacccaccagaaaagggtataggTTTCCACTACAGATGGTTGTGAGCCACCATGTGGTTGCTGGGAATTGAACTcagg..............................................
............$tgctgctctcttcagacccaccagaaaagggtataggTTTCCACTACAGATGGTTGTGAGCCACCATGTGGTTGCTGGGAATTGAACTcaggacctctac......................................
.....................$cttcagacccaccagaaaagggtataggTTTCCACTACAGATGGTTGTGAGCCACCATGTGGTTGCTGGGAATTGAACTcaggacctctacaagagcagt.............................
........................................$gggtataggTTTCCACTACAGATGGTTGTGAGCCACCATGTGGTTGCTGGGAATTGAACTcaggacctctacaagagcagtcagtgctctgaatctctga..........
.................................................gTTTCCACTACAGATGGTTGTGAGCCACCATGTGGTTGCTGGGAATTGAACTcaggacctctacaagagcagtcagtgctctgaatctctgagccatctc$.
.................................................gTTTCCACTACAGATGGTTGTGAGCCACCATGTGGTTGCTGGGAATTGAACTcaggacctctacaagagcagtcagtgctctgaatctctgagccatctc$.
.................................................gTTTCCACTACAGATGGTTGTGAGCCACCATGTGGTTGCTGGGAATTGAACTcaggacctctacaagagcagtcagtgctctgaatctctgagccatctc$.
.......................................aagggtataggTTTCCACTACAGATGGTTGTGAGCCACCATGTGGTTGCTGGGAATTGAACTcaggacctctacaagagcagtcagtgctctgaatctct$...........
..............................ccaccagaaaagggtataggTTTCCACTACAGATGGTTGTGAGCCACCATGTGGTTGCTGGGAATTGAACTcaggacctctacaagagcagtcagtgctc$....................
..........................agacccaccagaaaagggtataggTTTCCACTACAGATGGTTGTGAGCCACCATGTGGTTGCTGGGAATTGAACTcaggacctctacaagagcagtcagt$........................
....................ctcttcagacccaccagaaaagggtataggTTTCCACTACAGATGGTTGTGAGCCACCATGTGGTTGCTGGGAATTGAACTcaggacctctacaagagca$..............................
...........actgctgctctcttcagacccaccagaaaagggtataggTTTCCACTACAGATGGTTGTGAGCCACCATGTGGTTGCTGGGAATTGAACTcaggacctct$.......................................
.........acactgctgctctcttcagacccaccagaaaagggtataggTTTCCACTACAGATGGTTGTGAGCCACCATGTGGTTGCTGGGAATTGAACTcaggacct$.........................................
....tgagtacactgctgctctcttcagacccaccagaaaagggtataggTTTCCACTACAGATGGTTGTGAGCCACCATGTGGTTGCTGGGAATTGAACTcag$..............................................
...gtgagtacactgctgctctcttcagacccaccagaaaagggtataggTTTCCACTACAGATGGTTGTGAGCCACCATGTGGTTGCTGGGAATTGAACTcaggacctctacaagagcagtcagtgctctgaatctctgagccatctc..

..................................................***************************************************..................................................
............................$gacccccagaaagggtataggTTTCCACTACAGATGGTTGTGAGCCACCATGTGGTTGCTGGGAATTGAACTcaggacctctacaagagcagtcagtgc.......................
...................tctcttcagacccaccagaaaatggtataggTTTCCACTACAGATGGTTGTGAGCCACCATGTGGTTGCTGGGAATTGAACTcaggacctctacaagaga$...............................
...................tctcttcaggcccccccggaaatggtataggTTTCCACTACAGATGGTTGTGAGCCACCATGTGGTTGCTGGGAATTGAACTcaggacctctacaagagcagtcagtgc.......................
```

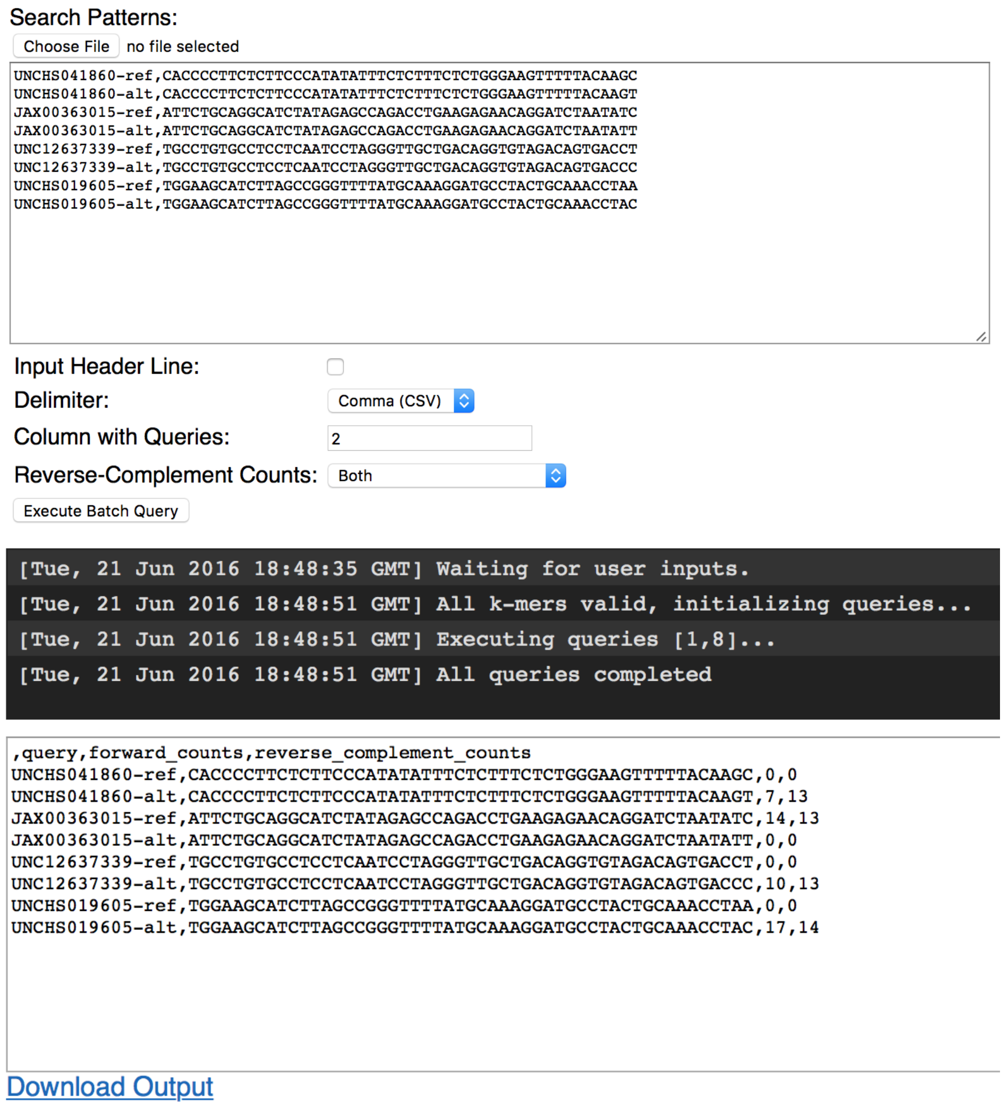
Figure 7.3: *K*-mer allele tool. This is a visualization of the allele based visualization of the same *k*-mer from Figure 7.2 in an inbred mouse DNA-seq dataset. After retrieving all reads with an exact matching *k*-mer, each entire read is compared against each other read and exact matching clusters are formed from the reads. Each cluster has a consensus sequence that every read in the cluster matches. Any reads that don't exactly match a cluster with at least five reads are allocated to the remainder consensus for the query. All consensus sequences and the remained consensus are shown as a summary table at the top of the visualization. Differences in the consensus sequences are highlighted in yellow. Additionally, each consensus has a numerical value indicating the number of reads that exactly match that consensus. Below the summary, each cluster of reads is shown with its matching consensus in the same visualization style of Figure 7.2.

these queries are pre-annotated and represent biologically relevant sequences such as single nucleotide polymorphisms, insertions, deletions, splice junctions, or copy number.

The batch query tools are designed to provide easy methods input a batch of k-mers and query all of them simultaneously through the web interface. The first version of this tool allows users to select a single dataset to query. Users then upload a CSV file (or manually type the queries into the web client), specify which column of the CSV file contains the k-mers, and tell the client to execute the queries. The client then requests k-mer counts from the server for each of the k-mers provided by the user. The output is another CSV file where all columns from the input are preserved and two new columns are added corresponding to forward and reverse-complement k-mer frequencies. The user can then copy this information or download the output to a CSV file for analysis. An example input and output for this tool is shown in Figure 7.4. As a final note, this tool also has a version that allows for querying k-mers within an edit distance. Any discovered k-mers within the specified edit distance are returned as extra columns in the output.

The second version of this tool allows for multiple datasets to be queried and saved to the output file. After selecting which datasets to query, the users again upload a CSV file, specify a column with labels and a column with queries, and tell the client to execute the queries. For each selected dataset, the client then requests k-mer counts from the server. Then, the output is formatted as a new CSV table where columns are the queries and rows are the datasets. For each dataset and k-mer, the corresponding cell stores the frequency of the k-mer in that dataset. An example output for this tool is shown in Figure 7.5.

In contrast to the k-mer based visualizations, these tools are better suited to gathering count statistics for many different k-mer patterns rather than to exploring the sequencing surrounding a particular pattern. In particular, this class of tool enables users to gather k-mer frequency information in multiple datasets without needing to download the datasets and count them using a locally installed program. Most use cases for these tools involve first identifying k-mer patterns that specifically target a biologically meaningful sequence. For example, the pattern may uniquely identify a particular splice junction or genomic variant within an organism. In either case, the sequence must be known beforehand and off-target variants may affect the results.


Search Patterns:

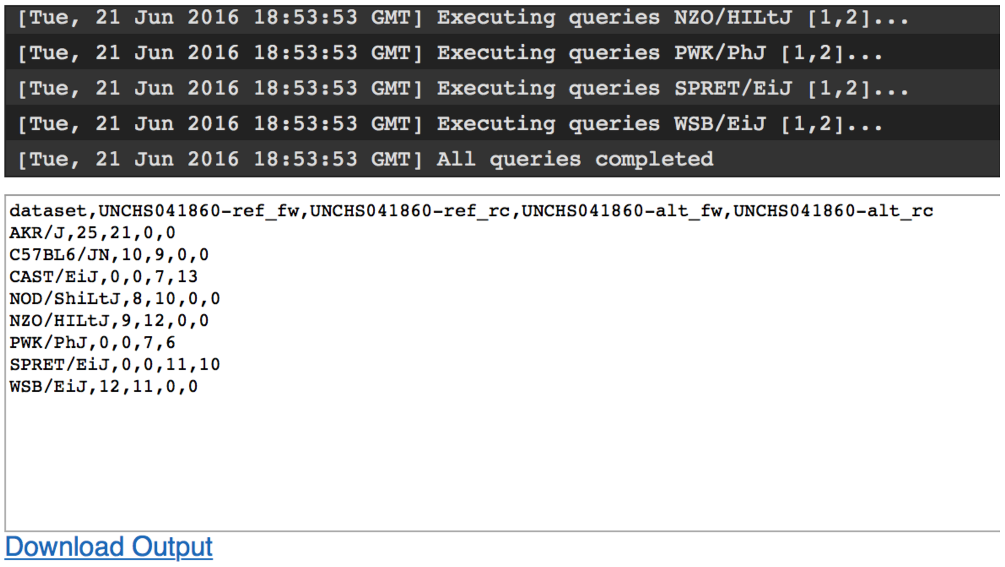
```
UNCHS041860-ref,CACCCCTTCTCTTCCCATATATTTCTCTTTCTCTGGGAAGTTTTTACAAGC
UNCHS041860-alt,CACCCCTTCTCTTCCCATATATTTCTCTTTCTCTGGGAAGTTTTTACAAGT
JAX00363015-ref,ATTCTGCAGGCATCTATAGAGCCAGACCTGAAGAGAACAGGATCTAATATC
JAX00363015-alt,ATTCTGCAGGCATCTATAGAGCCAGACCTGAAGAGAACAGGATCTAATATT
UNC12637339-ref,TGCCTGTGCCTCCTCAATCCTAGGGTTGCTGACAGGTGTAGACAGTGACCT
UNC12637339-alt,TGCCTGTGCCTCCTCAATCCTAGGGTTGCTGACAGGTGTAGACAGTGACCC
UNCHS019605-ref,TGGAAGCATCTTAGCCGGGTTTTATGCAAAGGATGCCTACTGCAAACCTAA
UNCHS019605-alt,TGGAAGCATCTTAGCCGGGTTTTATGCAAAGGATGCCTACTGCAAACCTAC
```

Input Header Line: ☐
Delimiter: Comma (CSV)
Column with Queries: 2
Reverse-Complement Counts: Both

```
[Tue, 21 Jun 2016 18:48:35 GMT] Waiting for user inputs.
[Tue, 21 Jun 2016 18:48:51 GMT] All k-mers valid, initializing queries...
[Tue, 21 Jun 2016 18:48:51 GMT] Executing queries [1,8]...
[Tue, 21 Jun 2016 18:48:51 GMT] All queries completed
```

```
,query,forward_counts,reverse_complement_counts
UNCHS041860-ref,CACCCCTTCTCTTCCCATATATTTCTCTTTCTCTGGGAAGTTTTTACAAGC,0,0
UNCHS041860-alt,CACCCCTTCTCTTCCCATATATTTCTCTTTCTCTGGGAAGTTTTTACAAGT,7,13
JAX00363015-ref,ATTCTGCAGGCATCTATAGAGCCAGACCTGAAGAGAACAGGATCTAATATC,14,13
JAX00363015-alt,ATTCTGCAGGCATCTATAGAGCCAGACCTGAAGAGAACAGGATCTAATATT,0,0
UNC12637339-ref,TGCCTGTGCCTCCTCAATCCTAGGGTTGCTGACAGGTGTAGACAGTGACCT,0,0
UNC12637339-alt,TGCCTGTGCCTCCTCAATCCTAGGGTTGCTGACAGGTGTAGACAGTGACCC,10,13
UNCHS019605-ref,TGGAAGCATCTTAGCCGGGTTTTATGCAAAGGATGCCTACTGCAAACCTAA,0,0
UNCHS019605-alt,TGGAAGCATCTTAGCCGGGTTTTATGCAAAGGATGCCTACTGCAAACCTAC,17,14
```

Figure 7.4: Mass query tool. These images are screenshots of the mass query tool when run on a small set of short k-mers. The top image shows a CSV input for a selection of k-mer queries where the input has an identifier, a k-mer query, and some metadata associated with the query. The user informs the client that a header line is present and that the k-mer queries are in second column of the input. The client then requests the specific k-mer counts from the server. In the output, the original CSV input is copied and two new columns are added corresponding to the forward and reverse-complement counts for the k-mer queries. This output can then be copied or downloaded in CSV format for more analysis. In this example, the chosen sample is an inbred organism, so only one version (either “ref" or “alt") of each allele has counts greater than zero in the output.



```
[Tue, 21 Jun 2016 18:53:53 GMT] Executing queries NZO/HILtJ [1,2]...
[Tue, 21 Jun 2016 18:53:53 GMT] Executing queries PWK/PhJ [1,2]...
[Tue, 21 Jun 2016 18:53:53 GMT] Executing queries SPRET/EiJ [1,2]...
[Tue, 21 Jun 2016 18:53:53 GMT] Executing queries WSB/EiJ [1,2]...
[Tue, 21 Jun 2016 18:53:53 GMT] All queries completed
```

```
dataset,UNCHS041860-ref_fw,UNCHS041860-ref_rc,UNCHS041860-alt_fw,UNCHS041860-alt_rc
AKR/J,25,21,0,0
C57BL6/JN,10,9,0,0
CAST/EiJ,0,0,7,13
NOD/ShiLtJ,8,10,0,0
NZO/HILtJ,9,12,0,0
PWK/PhJ,0,0,7,6
SPRET/EiJ,0,0,11,10
WSB/EiJ,12,11,0,0
```

Figure 7.5: Batch query tool. This image is a screenshot of the batch query tool when run on a two probe k-mers for multiple datasets. For this example, eight different datasets were selected by the user. The client requests k-mer counts for each dataset and outputs the result in a new CSV file format where columns represent k-mer queries and rows are the datasets being queried. The resulting CSV file can be copied or downloaded for future analysis. In this example, all eight datasets are inbred datasets, so only one of the two alleles has counts greater than zero for each dataset.

7.6 Reference based tools

In all the tools presented thus far, the user is required to provide k-mers relevant to the tool. Instead of specifying a particular query k-mer beforehand, the reference based tool requires that users specify a particular region from a reference genome. The tool will then automatically perform k-mer queries to search for in the BWT for a particular dataset. As a result, the server must have access to some auxiliary information to generate the queries. First, it must have access to a reference genome to pull k-mer queries from. Secondly, if lookups are done using gene name, there must be a database of gene annotations that can translate a particular gene ID into coordinates in the reference genome. Our particular tool uses the mouse reference genome build 37 and a list of gene annotations from BioMart [Smedley et al., 2015].

Given a region of the genome, the server splits the whole region into 40-mers that overlap by 20 bases. In general, a 40-mer is long enough to uniquely identify most genomic sequences in the reference genome and overlapping by 20 allows the tool to cover each base twice. If a gene ID is entered, the server first looks in a gene annotation database for the exons boundaries and uses the region from the start of the first exon to the end of the last exon. Each k-mer in the region is then queried against the user-selected dataset to retrieve counts for both the forward and reverse-complement of the k-mer. The output is a plot where the x-axis is the genomic coordinates of the region and the y-axis is the k-mer count for that region. Additionally, the specific k-mers from the region are displayed below the plot. Figure 7.6 shows an example plot generated from a mouse RNA-seq dataset for the region corresponding to gene *Egr3*.

One of the issues with using a reference genome is that there are often many differences between the reference and the given organism's genome. As a result, k-mer counts may drop to zero due to a variant in the genome (and the sequencing reads) that does not match the k-mer from the reference. To help combat this reference bias, the reference-based tool allows for minor corrections to the sequence to better match the sequencing dataset. In particular, when two overlapping, low count k-mers are identified, the tool hypothesizes that the shared overlapping area contains a variant that is not included in the reference genome. It then attempts to perform a small *de novo* assembly that is conceptually similar to the correction algorithms described in Section 6.3.3. If an alternate path exists to span the low count region, it is replaced and the new counts are report in the output along with the corrections made to the path. In Figure 7.7, the top plot shows the counts from an uncorrected reference genome for gene *Igf2* in a mouse DNA-seq dataset, and the bottom plot shows the counts for the same region after modifying the reference genome to better match the sequencing dataset. While there are still some areas with drops in the k-mer counts, many of the sudden drops have been corrected leading to a smoother graph overall.

This reference based tool is similar to the pileup counts produced by alignment based visualization tools like IGV [Thorvaldsdóttir et al., 2013]. However, reads can be double counted if the same k-mer occurs multiple times in the reference or read. Additionally, while the k-mer from a read must exactly match the reference query for it to be counted, there is no requirement that the remaining bases in the read match the reference at all. For sufficiently large k-mers, the k-mer frequencies from the BWT queries are expected to have a linear relationship with the pileup counts from an alignment. However, genomic characteristics such as duplications, gene families, or low complexity queries can lead to inconsistent k-mer frequencies because the query is not unique to a single genomic position. These inconsistencies may appear as sudden spikes for low complexity queries or as elevated plateaus in the event of duplication.

For example, Figure 7.8 shows the reference k-mer counts for a WSB/EiJ DNA-seq mouse

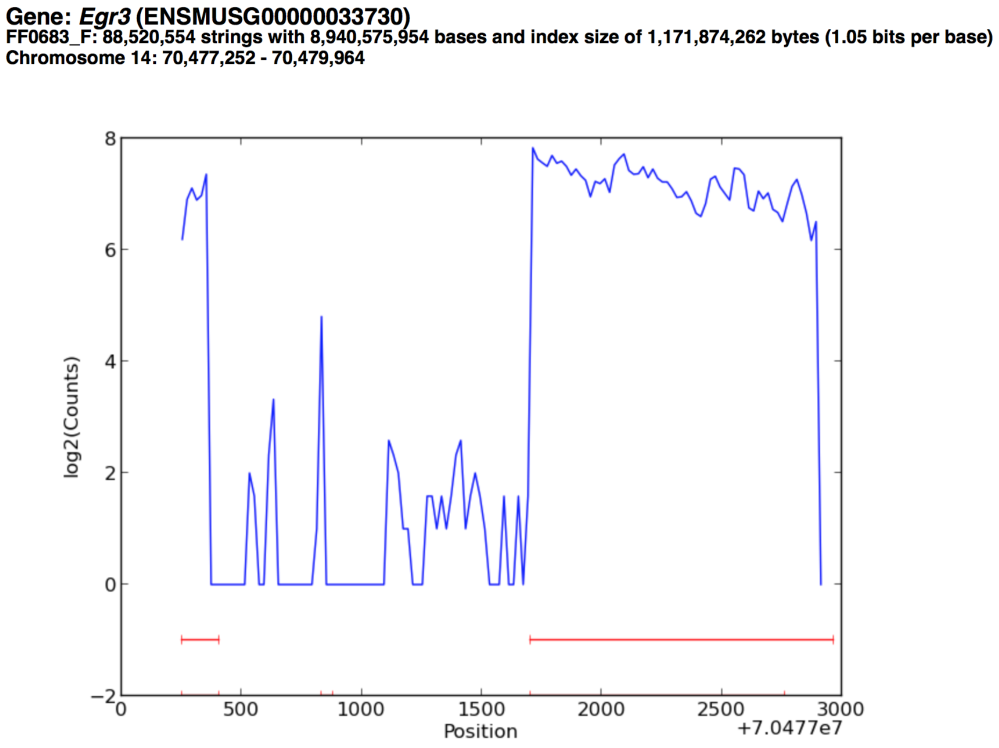
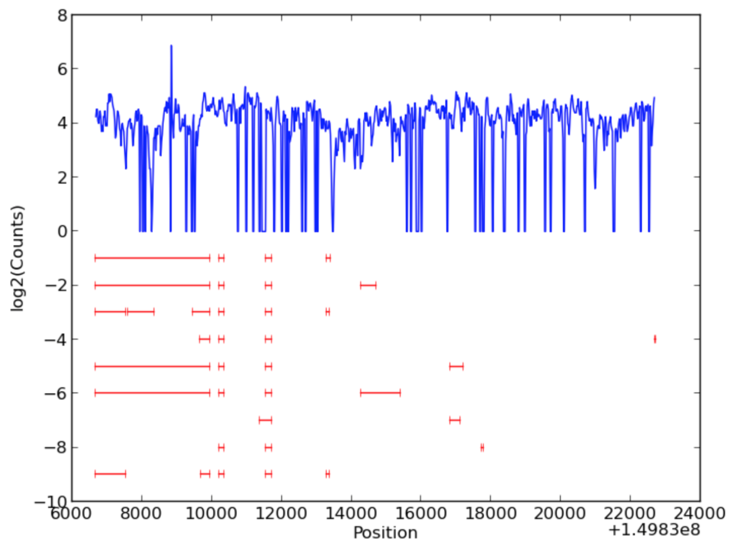


Figure 7.6: Reference based tool. This figure shows the plot generated by the reference based web tool for gene *Egr3* when run on a mouse RNA-seq dataset. The server uses a gene annotation database to extract all transcripts and exons corresponding to the gene. The coordinates of the first and last exon defined the boundaries of the region to query. Then, the server splits the region into 40-mers that are overlapping by 20 bases. Each 40-mer is queried against the BWT to retrieve counts. Finally, the counts are plotted such that the x-axis represents genomic coordinates and the y-axis represents the 40-mer count. Below each plot are the annotated transcripts from the database. Each transcript is shown as a series of red bars where each bar corresponds to an exon. In this example, 40-mers that are within an exon region have a count of approximately 150. In contrast, 40-mers that are within intron regions have much lower counts that are more likely noise from sequencing. In general, this is because intronic regions should not be captured by RNA sequencing, so the expectation is for those 40-mers to be absent from the RNA-seq dataset.

Gene: *Igf2* (ENSMUSG00000048583)
CASTdna: 1,156,409,988 strings with 116,797,408,788 bases and index size of 19,384,515,269 bytes (1.33 bits per base)
Chromosome 7: 149,836,673 - 149,852,721



Gene: *Igf2* (ENSMUSG00000048583)
CASTdna: 1,156,409,988 strings with 116,797,408,788 bases and index size of 19,384,515,269 bytes (1.33 bits per base)
Chromosome 7: 149,836,673 - 149,852,721

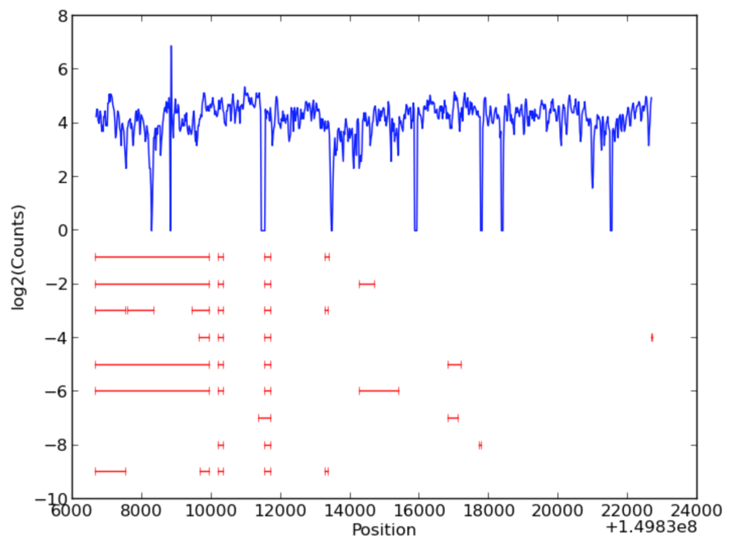


Figure 7.7: Reference based correction. The top figure shows the plot generated by the reference based web tool for gene *Igf2* when run on a mouse DNA-seq dataset. In general, we expect the counts from a DNA-seq dataset to be uniform. Within this dataset, there are many drops in the counts that may be caused by variants in the dataset from the reference genome. When adjacent, overlapping 40-mers both have low counts, the server attempts to change the shared 20-mer sequence to a sequence supported by the data using small *de novo* assemblies. If a new sequence is discovered, it replaces the old one and the counts for that sequence is saved. The bottom figure shows the same genomic region (*Igf2*) using the correction option of the tool. The number of low frequency k-mers is significantly reduced from the unmodified reference leading to a smoother overall plot.

dataset [Keane et al., 2011] near the boundary of a known duplication event on chromosome 2 at 77.7MB [Didion et al., 2015]. In the plot, k-mer frequencies near 2^5 represent the normal expected k-mer frequency in the dataset. However, there is an obvious plateau of high k-mer counts starting near 27.5kb in the plot that represents a duplicate sequence within the organism. This duplicated sequence, known as *R2d*, occurs in two forms: *R2d1* and *R2d2*. The *R2d1* sequence in this organism occurs once, and the start of this sequence has been previously mapped to the position shown in Figure 7.8 [Didion et al., 2015]. In contrast, the *R2d2* version occurs approximately 30 times within this dataset and has been previously mapped elsewhere on chromosome 2 [Didion et al., 2015]. In the plot, k-mers that occur with a normal frequency ($\approx 2^5$) are unique to the *R2d1* sequence. In contrast, k-mers that occur with the much higher frequency ($\approx 2^{10}$) are shared by both versions of *R2d*.

7.7 Targeted assembly tool

One common use of sequencing data is *de novo* assembly of multiple reads into longer sequences. One way to perform sequence assembly is to use a de Bruijn graph [Bruijn, 1946, Simpson and Durbin, 2010, 2012]. In a de Bruijn graph, every k-mer for a fixed size k is extracted from a dataset, and each one is stored as a node in the graph. Edges in a de Bruijn graph indicate that the $(k-1)$ suffix of the source node is the same as the $(k-1)$ prefix of the destination node. In other words, they overlap by $(k-1)$ symbols. Typically, de Bruijn graphs are "pruned" by removing k-mers from the graph that have a low frequency in the dataset because these nodes are more likely sequencing errors than true sequence. The BWT of a sequencing dataset can be used to represent a pruned de Bruijn graph for that same sequencing dataset [Simpson and Durbin, 2010, 2012]. In this representation, k-mer frequencies from a BWT query are used to determine if a node is "present" in the de Bruijn graph.

The targeted assembly tool allows users to access this pruned de Bruijn graph by exploring a dynamically graph from a user-selected "seed" k-mer. The tool displays the graph as a series of nodes connected by directed edges. In order to reduce clutter, adjacent nodes from the de Bruijn graph with one incoming edge and one outbound edge are collapsed into a single node for visualization. In this way, a visual node actually represents an unambiguous path through multiple connected nodes in the pruned de Bruijn graph. Branches (nodes with multiple outbound edges) are preserved in the visualization as these are likely interesting characteristics of the dataset. As

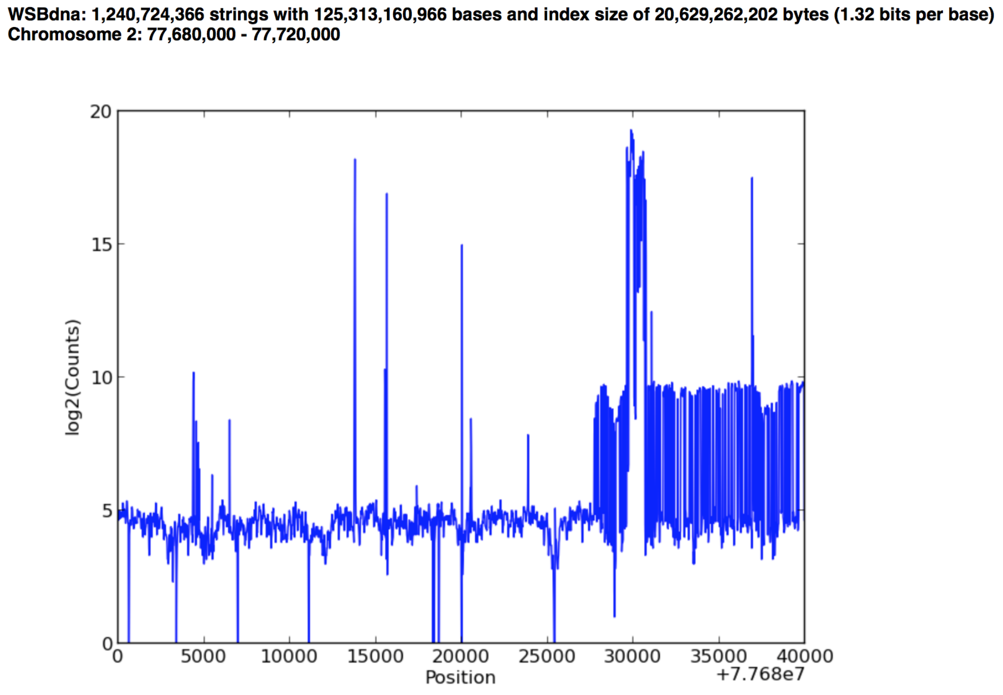


Figure 7.8: WSB/EiJ duplication example. This image shows the reference k-mer counts for a DNA-seq WSB/EiJ mouse dataset [Keane et al., 2011] near the boundary of a known duplication event on chromosome 2 at 77.7MB [Didion et al., 2015]. The left side of the plot represents sequence with normal coverage where most k-mers have a frequency of $\approx 2^5$. Starting at approximately 27.5kb on the x-axis, the frequencies change to have two modes: one centered around 2^5 and one centered around 2^{10}. This change represents the boundary of the duplicated sequence, labeled *R2d*, that is present in two forms in this datasets: *R2d1* and *R2d2* [Didion et al., 2015]. The *R2d1* sequence occurs one time in this dataset and has been previously mapped to this position in the reference genome [Didion et al., 2015]. Since it is a single copy, the k-mers with counts at $\approx 2^5$ correspond to k-mers that are specific to $R2d1$. In contrast, the *R2d2* sequence occurs approximately 30 times within this dataset and has been previously mapped elsewhere on chromosome 2. For k-mers shared between the two versions of *R2d*, the frequency of the k-mers jumps to approximately 2^{10} because it picks up additional k-mers from the 30 copies of *R2d2*.

the tool dynamically assembles nodes in the graph, auxiliary information is stored in a table below the graph including the sequence corresponding to the node, the length of that sequence, and the forward and reverse-complement k-mer counts for each k-mer in the sequence.

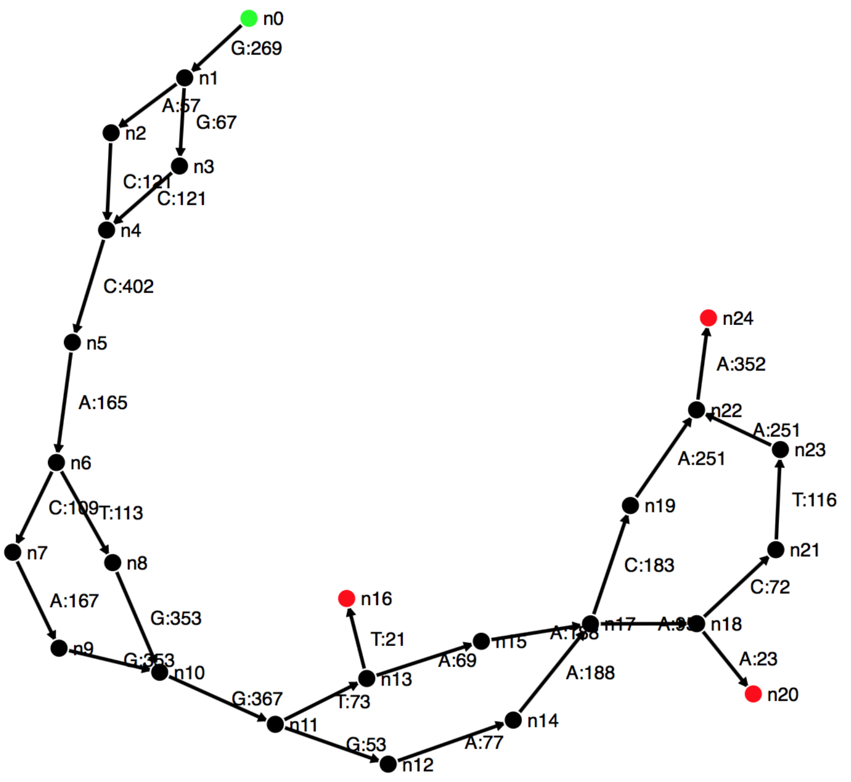
This visualization can be useful for highlighting variation within a sequencing dataset. For example, Figure 7.9 shows the pruned, collapsed de Bruijn graph for gene *Egr3* generated from the BWT of an RNA-seq dataset for an F1 hybrid mouse derived from crosses between two inbred mouse strains CAST/EiJ and PWK/PhJ. Since this is an F1 organism, branch points can be caused by either genomic variation or alternate splicing. For the branch point at node “n1", the de Bruijn graph splits into two nodes: “n2" and “n3". The sequences corresponding to nodes “n2" and “n3" are shown at the bottom of Figure 7.9. These nodes have identical sequence except for a single nucleotide at position 50, “n2" has an ‘A’ whereas “n3" has a ‘G’. Additionally, once the different nucleotide is no longer included in the k-mers in the separate paths, the two paths merge together at node “n4".

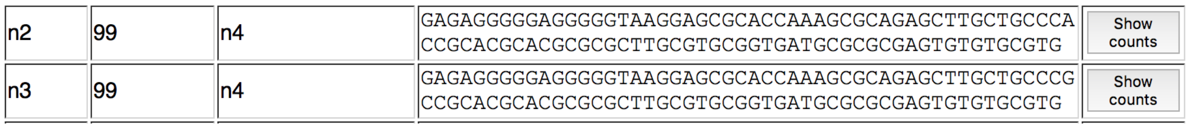
In addition to highlighting variants, the graph can also provide insights into the structure of a sequence. Figure 7.10 shows the graph for mitochondrial sequence in a CAST/EiJ DNA-seq datasets from Keane et al. [2011]. One interesting characteristic of mammalian mitochondria is that they are circular pieces of DNA. In this example, the nodes in the graph also form a cycle that reflects the structure of the mitochondria. A second interesting characteristic is derived from the branch point at node “n3" that indicates possible heteroplasmy, the presence of more than one type of mitochondria. In general, this is not expected because mitochondria are inherited strictly from the mother. Inspection of the sequences in the bottom half of Figure 7.10 reveals the difference to be a single C/T polymorphism that causes the branch.

Finally, the targeted assembly tool allows users to piece together a path through the visual graph to form a long sequence. The user selects a node to start from and then repeatedly selects successor nodes in the graph. The sequences from each node are joined together using the overlapping sequences to form a single long string representing a path through the de Bruijn graph. This string can then be downloaded and saved in FASTA format for future use.

7.8 Final notes

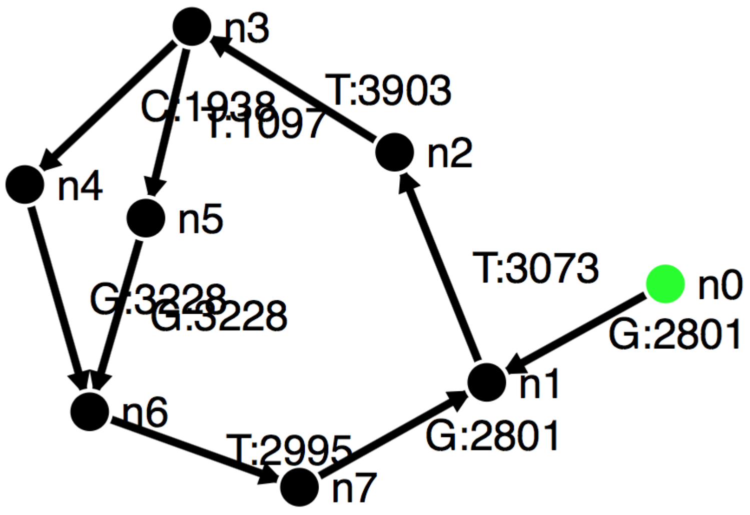
The BWT and FM-index are powerful data structures for storing and quickly accessing raw sequence data. The BWT is a lossless transform, so all sequencing reads are contained within the





n2	99	n4	GAGAGGGGGAGGGGGTAAGGAGCGCACCAAAGCGCAGAGCTTGCTGCCCA CCGCACGCACGCGCGCTTGCGTGCGGTGATGCGCGCGAGTGTGTGCGTG	Show counts
n3	99	n4	GAGAGGGGGAGGGGGTAAGGAGCGCACCAAAGCGCAGAGCTTGCTGCCCG CCGCACGCACGCGCGCTTGCGTGCGGTGATGCGCGCGAGTGTGTGCGTG	Show counts

Figure 7.9: *Egr3* de Bruijn graph. The top image shows the pruned, collapsed de Bruijn graph for gene *Egr3* generated from the BWT of an RNA-seq dataset for an F1 mouse organism. For this graph, we used the FG0125_F dataset from Crowley et al. [2015], an initial 50-mer seed of "CGGGCCACAAGCCCTTCCAGTGTCGGATCTGCATGCGCAGCTTCAGCCGC", and a pruning threshold of 20. The start point is the green node "n0" which corresponds to the initial 50-mer seed. End points are represented as red nodes in the graph. Each edge in the graph has a symbol and numerical value corresponding to the last symbol of the first 50-mer in the destination node and the frequency of that 50-mer in the dataset. Since this is an F1 organism, the graph has multiple branch points generated by genomic variants within the sample. For example, the first branch is located at node "n1" leading to two new nodes "n2" and "n3". The bottom image shows the sequences corresponding to nodes "n2" and "n3". In this particular example, the 50-th symbol (the last symbol on the first row of each sequence) is different between the two sequences: "n2" has an 'A' and "n3" has a 'G'. Once the different symbol is cycled out of the 50-mer, the two paths merge together into node "n4".



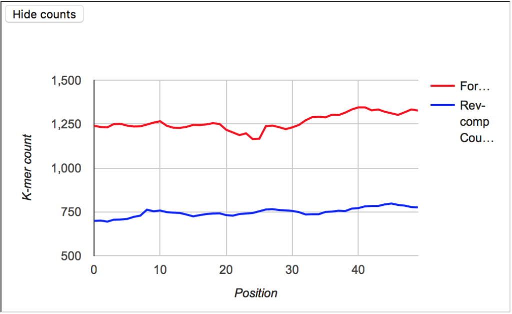
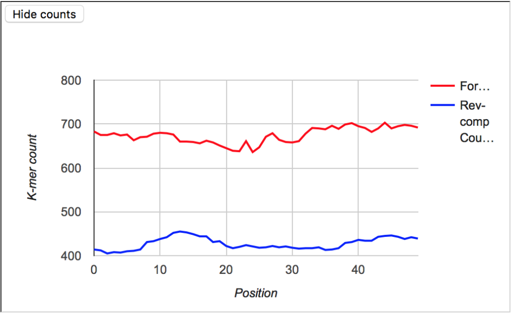
n4	99	n6	CCTCATCTTAATTCAAAAACATGTAGGAACCCTAAACCTCATAATTTTAC CATTCACAACACACACCTTAGACACTTCATGATCTAACAACTTACTATG	
n5	99	n6	CCTCATCTTAATTCAAAAACATGTAGGAACCCTAAACCTCATAATTTTAT CATTCACAACACACACCTTAGACACTTCATGATCTAACAACTTACTATG	

Figure 7.10: CAST/EiJ mitochondria graph. The top image shows the pruned, collapsed de Bruijn graph for the mitochondria generated from the BWT of a DNA-seq dataset for a CAST/EiJ mouse Keane et al. [2011]. The graph was generated using an initial 50-mer seed of "GTTAATGTAGCT-TAATAACAAAGCAAAGCACTGAAAATGCTTAGATGGAT" and a pruning threshold of 400. The mitochondria is interesting because it is an abundant piece of circular DNA that is inherited only from the mother. The circular structure of the mitochondria is reflected by the graph. Additionally, there appears to be a single branch point in the graph between nodes "n4" and "n5". Closer inspection of the sequences and k-mer counts (shown the bottom image) reveal that this is caused by a single C/T polymorphism. This difference indicates possible heteroplasmy (the presence of more than one type of mitochondria) in the dataset.

data structure. Additionally, the FM-index enables both arbitrary k-mer queries and extraction of individual reads from the dataset without needing to reconstruct the entire string collection. However, there is often a learning curve to using APIs associated with BWTs and FM-indices. The web tools discussed in this chapter allow users to access the raw sequence data through sequence-centered queries while masking the complexities of the BWT and FM-index.

In this chapter, we presented several web tools designed to address different types of sequence-centered queries. The first set of tools provides a lot of information about a user-selected k-mer including forward counts, reverse-complement counts, every read with the k-mer, and a consensus sequence of the reads. The main advantage of the tool is that it provides a lot of information about a single k-mer. However, it does not generalize to multiple k-mers because the output becomes excessively large for the servers to compute. In particular, extracting the reads for a given k-mer is computationally expensive, so performing this operation for multiple k-mers can lead to server timeouts.

The second set of tools attempts to address this issue by reducing the amount of information retrieved for each k-mer. These batch query tools allows the user to simply retrieve forward and reverse-complement counts for one or more k-mers from one or more datasets. While this is a reduced set of information, the counts alone are informative for detecting the presence of specific k-mers. Typically, observing a particular k-mer reflects the occurrence of some biologically event (i.e. a probe) such as a SNP, indel, or splice junction. The raw k-mer counts can then be downloaded and incorporated into downstream analyses.

The reference based tool allows users to retrieve k-mer queries for a large region of a reference genome. One downside of this tool is the bias introduced by using a reference genome. In particular, if the sequenced organism has an off-target variant (a variant not included in the reference genome), k-mers from the reference genome that do not have that variant will likely have no counts. While there is an option to correct minor variants, the online correction method is relatively simple. As a result, multiple variants or large structural variations are usually not corrected with the reference based tool.

The targeted assembly tool works well for unbiased assembly of short sequences. It provides users accessed to a compressed, pruned de Bruijn graph of their data starting from a particular k-mer. However, selection of the k-mer and pruning threshold can drastically impact the output graph in ways users may not expect. Additionally, it can be tedious to perform assemblies through areas with a large number of branches. To address these issues, future work may include assisting users with the choice of threshold or using a dynamic threshold that is based on the counts thus far. Furthermore, an option to automatically build the graph in a depth-first or breadth-first manner could help alleviate some of the tedious aspects of the graph construction.

One consistent problem with these tools is figuring out how to select k-mer queries and interpret the results. By default the FM-index counts only exact matching k-mers, so any off-target sequence can cause misleading k-mer counts. This can be caused by true off-target variants (SNPs, indels), RNA editing, sequencing error, or even technical sequencing bias. As a result, users have to be careful when interpreting the k-mer counts to make sure that the results are not convoluted by off-target sequences. One way to try to detect off-target sequences is by using the tools that allow for an edit distance in the search. This enables users to capture exact matching k-mer counts and the counts for k-mers that are close to the original query.

All of the tools discussed in this chapter are publicly accessible through the MSBWT Tools splash page for a selection of mouse datasets[1]. Additionally, biologists have used many of these tools for preliminary analysis of their datasets [Didion et al., 2015]. A subset of the tools have been ported into a publicly available Python package that runs on a Flask server[2]. This package allows anyone to set up a local BWT server for their own datasets.

[1]http://www.csbio.unc.edu/CEGSseq/index.py?run=MsbwtTools
[2]https://github.com/holtjma/msbwtWebTools

CHAPTER 8

CONCLUSION

8.1 Summary

In this dissertation, three algorithms that operate directly on the BWT were introduced. The first algorithm merges two or more BWTs into a single BWT containing all the strings from all inputs. For two BWTs with N total symbols and a longest common substring of length LCS, the algorithm requires a total of $O(N * LCS)$ steps to compute the merge BWT. The algorithm enables BWTs to be combined without needing to reconstruct the original string collections, allowing downstream analyses to perform a single FM-index query across all merged datasets simultaneously.

The second algorithm applies the merge algorithm to create a merge-based BWT construction algorithm. The construction algorithm has two key improvements over the original merge algorithm: divide-and-conquer merge and bucket tracking. For a string collection with m strings composed of N symbols and an average longest common prefix of LCP_{avg}, these changes led to a construction algorithm requiring $O(N * LCP_{avg} * \log(m))$ steps to compute the BWT. In practice, this algorithm works relatively well for long read sequencing datasets. In particular, it performs well on PacBio CLR dataset that have long reads with low LCP_{avg}. Unfortunately, the algorithm does not scale well to datasets with a large number of reads or datasets where the LCP_{avg} is large.

The third algorithm, MSBWT-IS, is another BWT construction algorithm that uses induced sorting to calculate the BWT. First, the algorithm calculates a new alphabet to represent substrings from the input string collection. The alphabet is used to generate a new string collection that is guaranteed to have strings that are at most half as long as the original strings. This process is repeated recursively until the BWT of the string collection can be trivially calculated. While exiting the recursion, the BWT of the short collection represents a partial order of the BWT of the long collection that allows the total order to be induced. This algorithm is a BWT construction algorithm that constructs the BWT of a string collection with N symbols in $O(N)$ steps. Additionally, while many BWT construction algorithms are generalizable to different alphabets, this implementation is trivially adapted because it must already handle large alphabets during the recursion. For long-read sequencing datasets, MSBWT-IS was shown to require less CPU time than other BWT construction algorithms. However, it is not obvious how to parallelize the algorithm. Additionally, while this algorithm computes the BWT for short-read datasets relatively quickly, it does not outperform other implementations that are specifically designed for short-read datasets.

This dissertation also introduced a long-read correction method that uses the BWT of a short-read dataset to perform the correction. The method calculates the BWT of the short reads once, and then uses it as both a short k-mer de Bruijn graph and a long K-mer de Bruijn graph. It then uses these graphs to identify k-mers from the long reads that are weakly support by the short reads. Finally, these weak regions are replaced with strong paths that are assembled form the de Bruijn graph of the short reads. This method is unique in that it uses a long K-mer to perform a second pass of correction, allowing for low-complexity sequences or repeated sequences smaller than K to be corrected in the output. In the results, we showed that this method reduced the number of errors in the alignment of the reads. Additionally, the long K-mer correction method can be used as a post-process to other short k-mer correction algorithms to improve their results.

Finally, this dissertation described several web tools that use the BWT as an underlying storage scheme for raw sequence datasets and the FM-index for performing arbitrary k-mer queries. One set of tools retrieves all reads containing a particular k-mer, generates a surrounding consensus sequence, and displays the results highlighting variations in the sequence. The second set of tools can perform a batch of k-mer queries and report the results as summary counts that can be downloaded for analysis by the user. The third tool uses the reference genome as a source of k-mer queries and can visualize the output in a manner similar to a pileup count from an alignment. The fourth tool allows users access to the de Bruijn graph of a dataset and enables targeted assemblies of the reads into much longer sequences. The tools help mask the complexities of using a BWT and FM-index while simultaneously providing lossless interfaces to the raw sequence data that other data formats cannot provide.

8.2 Future work

Improving the performance of BWT construction algorithms for string collection is one area of ongoing research. MSBWT-IS requires less CPU time for long reads than the main competitor, ropebwt2, but tends to be slower by wall clock time because it is not parallelized. While some components of the MSBWT-IS algorithm can be trivially parallelized (S* substring discovery and replacement), it is not immediately obvious how to parallelize other parts of the algorithm that currently require a sequential execution. Future work will involve exploring both algorithmic and engineering solutions to this problem in order to make MSBWT-IS more competitive in terms of wall clock time. Additionally, MSBWT-IS performs well for long reads and ropebwt2 performs well for short reads, but there is still not a single algorithm that performs well for any type of string collection. Therefore, it is possible that an undiscovered BWT construction algorithm exists that works well on all types of sequencing datasets.

In addition to constructing BWTs quickly and more efficiently, there is actually a large amount of metadata associated with each read that is usually ignored during the construction such as paired end information or the quality strings. The main issues with quality strings is that they do not compress as well as the BWT because there is a larger alphabet and it is not obvious how to do random lookups of a particular quality score. Paired end information can actually be stored in an auxiliary lookup table that pairs reads based on their unique '\$' symbol in the BWT. However, this requires additional overhead in both the storage and computation of the pairs because the information is only associated with the '\$'. There is the possibility of creating paired-end strings that are a concatenation of the two paired end reads with a delimiter character in between. This will have an impact on both the compression of the BWT data structure and possibly the size of the FM-index table. While not discussed in this dissertation, the FM-index queries can also be sped up using an auxiliary data structure called the longest common prefix array (LCP array). Unfortunately, the LCP array is a large data structure that does not compress very well. Future work may involve researching alternate methods for storing the LCP array such that the memory usage of the array is small and random accesses are quick.

The implementation of the BWT-based correction method, FMLRC, also has room for improvement. In particular, we showed that the performance of FMLRC drastically changes depending on the implementation of the indexing structure, suggesting that there may be modification or alternate implementations that are faster and/or more memory efficient than the current implementation. Secondly, there is some evidence that the actual bridging algorithm is not as thorough as the one in LoRDEC. Additionally, many of the k-mer sizes were chosen based on the results of limited experimentation. Further experimentation may reveal that dynamically choosing k and K based on the k-mer frequencies may yield better results than a static, user-defined value.

The web tools presented in the dissertation represent only a small set of possible tools that can be generated using the BWT and FM-index. These web tools are relatively generic and mostly rely on information that is contained solely within the raw sequencing data. The one exception is the reference-based tool that incorporates other expectations about the organism into the queries for the user. There are many other possible tools that can incorporate other outside information in a similar manner. For example, a tool that automatically performs a set of pre-defined probe queries on a BWT or a tool that performs sequence correction on a user defined input sequence. Many of the tools created thus far were made at the request of collaborators in order to suit a specific need. Thus, future research on BWT-based web tools will likely be driven by the needs of researchers to ask specific questions of the sequencing datasets.

In summary, the BWT and FM-index are efficient data structures for compressing and accessing raw sequencing data. The algorithm implementations that are currently available allow for the efficient construction of BWTs for most types of sequencing datasets. This representation is a lossless, unbiased representation of the sequencing dataset that enables access to the raw reads through k-mer queries. Since the BWT and FM-index can now be constructed efficiently, tools and interfaces have been created to assist in solving biologically relevant problems such as *de novo* assembly [Simpson and Durbin, 2010], splice junction detection [Cox et al., 2012b], short-read correction [Greenstein et al., 2015], long-read correction, probe searches, and targeted genomic assembly.

APPENDIX A

MULTI-STRING BWT UTILITY SUITE

The Multi-String BWT Utility Suite is a publicly available Python/Cython package for interfacing with BWTs of genomic sequencing data. The package includes a command line interface for constructing BWTs and a Python/Cython API for developing custom scripts to query the resulting BWTs. Both the command line interface and API are available through PyPI[1] and GitHub[2]. Additionally, wiki pages[3] are available describing various use cases of the package along with more detail on the CLI and API. In the following sections, we briefly discuss some of these use cases and functionality included in the package.

A.1 Command line interface

The Command Line Interface (CLI) is primarily for the creation of BWTs from raw sequencing files (i.e. FASTQ). This interface provides access to two implementations of BWT creation algorithm: a BCR style algorithm and a merge-based construction algorithm. The BCR-style algorithm is basically a Cython re-implementation of the methods first described by Bauer et al. [2011]. In short, the algorithm uses a “column-wise" approach to construction that inserts all bases at a particular index in the strings simultaneously. The method works best for short, uniform length reads primarily due to increased run-times caused by longer strings (there are more “columns" to insert). The second algorithm is the merge-based construction algorithm described in Chapter 4. This algorithm creates many smaller BWTs and repeatedly merges them until only one BWT remains. This method works best on relatively small datasets, but allows for variable length, long reads to be included in the dataset. Both construction implementations are parallelized versions of their respective algorithms.

In addition to construction, the CLI allows for merging two or more BWTs into one larger BWT. This is an implementation of the algorithm described in Chapter 3. The output is a BWT containing the combined string collections from both inputs.

The CLI also has a function to enable converting a text string to one of the BWT formats supported by the API. This allows one to use an alternate BWT construction implementation, such

[1]`https://pypi.python.org/pypi/msbwt`
[2]`https://github.com/holtjma/msbwt`
[3]`https://github.com/holtjma/msbwt/wiki`

as ropebwt2 [Li, 2014], provided it constructs the BWT using the definitions from Chapter 2. The conversion command will then take a plain text input and convert it to the Run-Length Encoded format described in Section A.3.

Finally, the CLI enables some rudimentary queries of a constructed BWT. When queried, the tool automatically generates and saves an FM-index to disk for future use. This FM-index is then used as described in Section 2.3 to enable k-mer searches in the BWT. When a k-mer is found, the CLI will output the reads containing the k-mer. Additionally, a mass query option allows the user to retrieve k-mer counts for many different k-mer simultaneously using the CLI. For more complex queries, the Python/Cython API is recommended.

A.2 Python/Cython API

While the CLI makes BWT construction relatively easy, the types of queries that can be performed through it are somewhat limited by design. In most applications, the k-mers are usually not static search values where only one query is necessary. For example, methods like *de novo* assembly or read correction typically require many queries selected based on the results of previous queries, so the k-mer queries that will be needed are not necessarily known prior to the program execution. The Python/Cython API helps enable these dynamic query cases.

The API is basically a collection of query functions for accessing the BWT and FM-index of a genomic dataset. The API is written in Cython so that it can be easily imported into a Python script while benefiting from the speed of compiled C code. At a high level, the main function of the API is to search for k-mers in a BWT. Usually, a k-mer is returned as a range in the BWT corresponding to the indices of suffixes in the implicit suffix array that start with the queried k-mer (see Section 2.4 for more on this relationship). In the API, this range can be used to retrieve all reads with the k-mer, count the k-mer frequency, or extend the k-mer to a larger $(k+1)$-mer search.

The API also enables access to in-exact matching k-mer searches. The exact match algorithm is an $O(k)$ search for an arbitrary k-mer. With in-exact matching, the search is basically a branch-and-bound k-mer search. Given an edit distance, e, and that we allow for all five non-'\$' characters ('A', 'C', 'G', 'N', and 'T'), this algorithm would search for $O(k * 5^e)$ k-mers if left unbounded. The bounding portion of the algorithm requires that a minimum frequency threshold, t, is required to continue exploring the branch. This helps reduce the run-time of an inexact k-mer search in practice, but it does not reduce the asymptotic performance of the method in the worst case. The specific

Python API for the BWT class can be found on the BasicBWT API wiki page[4].

A.3 BWT formats

Included in the package is code support two different BWT formats. In both formats, the valid characters ([\$, A, C, G, N, T]) are re-mapped to integer values ([0, 1, 2, 3, 4, 5]). In the first format, the BWT is stored in an uncompressed manner such that there is one byte per symbol. In other words, it is a plain text format where each symbol has been remapped to its corresponding integer value.

The second format is a Run-Length Encoded (RLE) format. In this format, the BWT is broken into runs of adjacent symbols. The encoding separates each byte into the low 3 bits for the symbol and high 5 bits for the run length. For example, a run of 'A's of length 10 would be encoded as (10, 1) = 01010 001. If a run is longer than 31 symbols, the next byte is used to extend the run into a 10 bit value. For example, encoding a run of 'T's of length 47 requires more than 5 bits, so it is extended to two 5-bit values 0b00001 and 0b01111 that are encoded as the five most significant bits in two adjacent bytes. The three least significant bits in each byte are then set to 0b110 (corresponding to 'T'), so the resulting bytes are 0b01111110 and 0b00001110. The approach can be extended to encode a run of length, R, in ceil($\log_{32}(R)$) bytes.

In general, the RLE BWT is recommended for two reasons: compression and speed. In most short-read sequencing datasets, an RLE BWT requires 1.0-1.25 bits per base as opposed to 8 bits per base in the uncompressed version. Additionally, since the BWT uses a sampled FM-index, each query requires a linear traversal of a section of the BWT. In a RLE BWT, each page on disk contains more information due to compression. As a result, fewer pages need to be read from disk because the structure is smaller and contains more symbols per page than the uncompressed version.

Regardless of the format chosen, the BWT classes all inherit from a parent class with the core query functions. This means that implementation of alternate storage schemes need to only develop a few subroutines to work with the rest of the API. Once these subroutines are correctly implemented, k-mer queries and string recovery functions will work with the new format. This allows future testing of alternate compression methods without need to re-engineer all of the BWT query functions.

[4]https://github.com/holtjma/msbwt/wiki/BasicBWT-API

REFERENCES

Kin Fai Au, Jason G Underwood, Lawrence Lee, and Wing Hung Wong. Improving pacbio long read accuracy by short read alignment. *PLoS One*, 7(10):e46679, 2012.

Markus J Bauer, Anthony J Cox, and Giovanna Rosone. Lightweight bwt construction for very large string collections. In *Combinatorial Pattern Matching*, pages 219–231. Springer, 2011.

Markus J Bauer, Anthony J Cox, and Giovanna Rosone. Lightweight algorithms for constructing and inverting the bwt of string collections. *Theoretical Computer Science*, 483:134–148, 2013.

Dennis A Benson, Ilene Karsch-Mizrachi, David J Lipman, James Ostell, and David L Wheeler. Genbank. *Nucleic acids research*, 36(suppl 1):D25–D30, 2008.

de NG Bruijn. A combinatorial problem. *Proceedings of the Koninklijke Nederlandse Akademie van Wetenschappen. Series A*, 49(7):758, 1946.

Michael Burrows and David J Wheeler. A block-sorting lossless data compression algorithm. 1994.

Jonathan Butler, Iain MacCallum, Michael Kleber, Ilya A Shlyakhter, Matthew K Belmonte, Eric S Lander, Chad Nusbaum, and David B Jaffe. Allpaths: de novo assembly of whole-genome shotgun microreads. *Genome research*, 18(5):810–820, 2008.

Mark J Chaisson and Glenn Tesler. Mapping single molecule sequencing reads using basic local alignment with successive refinement (blasr): application and theory. *BMC bioinformatics*, 13(1): 238, 2012.

Rayan Chikhi and Guillaume Rizk. Space-efficient and exact de bruijn graph representation based on a bloom filter. *Algorithms for Molecular Biology*, 8(1):1, 2013.

Chen-Shan Chin, David H Alexander, Patrick Marks, Aaron A Klammer, James Drake, Cheryl Heiner, Alicia Clum, Alex Copeland, John Huddleston, Evan E Eichler, et al. Nonhybrid, finished microbial genome assemblies from long-read smrt sequencing data. *Nature methods*, 10(6):563–569, 2013.

1000 Genomes Project Consortium et al. A map of human genome variation from population-scale sequencing. *Nature*, 467(7319):1061–1073, 2010.

Anthony J Cox, Markus J Bauer, Tobias Jakobi, and Giovanna Rosone. Large-scale compression of genomic sequence databases with the burrows–wheeler transform. *Bioinformatics*, 28(11): 1415–1419, 2012a.

Anthony J Cox, Tobias Jakobi, Giovanna Rosone, and Ole B Schulz-Trieglaff. Comparing dna sequence collections by direct comparison of compressed text indexes. In *Algorithms in Bioinformatics*, pages 214–224. Springer, 2012b.

James J Crowley, Vasyl Zhabotynsky, Wei Sun, Shunping Huang, Isa Kemal Pakatci, Yunjung Kim, Jeremy R Wang, Andrew P Morgan, John D Calaway, David L Aylor, et al. Analyses of allele-specific gene expression in highly divergent mouse crosses identifies pervasive allelic imbalance. *Nature genetics*, 47(4):353–360, 2015.

John P Didion, Andrew P Morgan, Amelia M-F Clayshulte, Rachel C Mcmullan, Liran Yadgary, Petko M Petkov, Timothy A Bell, Daniel M Gatti, James J Crowley, Kunjie Hua, et al. A multi-megabase copy number gain causes maternal transmission ratio distortion on mouse chromosome 2. *PLoS Genet*, 11(2):e1004850, 2015.

John Eid, Adrian Fehr, Jeremy Gray, Khai Luong, John Lyle, Geoff Otto, Paul Peluso, David Rank, Primo Baybayan, Brad Bettman, et al. Real-time dna sequencing from single polymerase molecules. *Science*, 323(5910):133–138, 2009.

Paolo Ferragina and Giovanni Manzini. An experimental study of an opportunistic index. In *Proceedings of the twelfth annual ACM-SIAM symposium on Discrete algorithms*, pages 269–278. Society for Industrial and Applied Mathematics, 2001.

Paolo Ferragina, Travis Gagie, and Giovanni Manzini. Lightweight data indexing and compression in external memory. *Algorithmica*, 63(3):707–730, 2012.

Marc Fiume, Vanessa Williams, Andrew Brook, and Michael Brudno. Savant: genome browser for high-throughput sequencing data. *Bioinformatics*, 26(16):1938–1944, 2010.

Seth Greenstein, James Holt, and Leonard McMillan. Short read error correction using an fm-index. In *Bioinformatics and Biomedicine (BIBM), 2015 IEEE International Conference on*, pages 101–104. IEEE, 2015.

James Holt and Leonard McMillan. Merging of multi-string bwts with applications. *Bioinformatics*, page btu584, 2014a.

James Holt and Leonard McMillan. Constructing burrows-wheeler transforms of large string collections via merging. In *Proceedings of the 5th ACM Conference on Bioinformatics, Computational Biology, and Health Informatics*, pages 464–471. ACM, 2014b.

Scott D Kahn. On the future of genomic data. *Science(Washington)*, 331(6018):728–729, 2011.

Thomas M Keane, Leo Goodstadt, Petr Danecek, Michael A White, Kim Wong, Binnaz Yalcin, Andreas Heger, Avigail Agam, Guy Slater, Martin Goodson, et al. Mouse genomic variation and its effect on phenotypes and gene regulation. *Nature*, 477(7364):289–294, 2011.

W James Kent. BlatÑthe blast-like alignment tool. *Genome research*, 12(4):656–664, 2002.

Donald Ervin Knuth. *The art of computer programming: sorting and searching*, volume 3. Pearson Education, 1998.

Sergey Koren, Michael C Schatz, Brian P Walenz, Jeffrey Martin, Jason T Howard, Ganeshkumar Ganapathy, Zhong Wang, David A Rasko, W Richard McCombie, Erich D Jarvis, et al. Hybrid error correction and de novo assembly of single-molecule sequencing reads. *Nature biotechnology*, 30(7):693–700, 2012.

Ben Langmead, Cole Trapnell, Mihai Pop, Steven L Salzberg, et al. Ultrafast and memory-efficient alignment of short dna sequences to the human genome. *Genome Biol*, 10(3):R25, 2009.

Heng Li. Fast construction of fm-index for long sequence reads. *Bioinformatics*, page btu541, 2014.

Heng Li and Richard Durbin. Fast and accurate short read alignment with burrows–wheeler transform. *Bioinformatics*, 25(14):1754–1760, 2009.

Po-Ru Loh, Michael Baym, and Bonnie Berger. Compressive genomics. *Nature biotechnology*, 30(7): 627–630, 2012.

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

Sabrina Mantaci, Antonio Restivo, Giovanna Rosone, and Marinella Sciortino. An extension of the burrows wheeler transform and applications to sequence comparison and data compression. In *Combinatorial Pattern Matching*, pages 178–189. Springer, 2005.

Iain Milne, Micha Bayer, Linda Cardle, Paul Shaw, Gordon Stephen, Frank Wright, and David Marshall. TabletÑnext generation sequence assembly visualization. *Bioinformatics*, 26(3):401–402, 2010.

Ge Nong, Sen Zhang, and Wai Hong Chan. Linear suffix array construction by almost pure induced-sorting. In *Data Compression Conference, 2009. DCC'09.*, pages 193–202. IEEE, 2009.

Enno Ohlebusch, Simon Gog, and Adrian Kügel. Computing matching statistics and maximal exact matches on compressed full-text indexes. In *String processing and information retrieval*, pages 347–358. Springer, 2010.

Daisuke Okanohara and Kunihiko Sadakane. A linear-time burrows-wheeler transform using induced sorting. In *String Processing and Information Retrieval*, pages 90–101. Springer, 2009.

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.

David A Rasko, MJ Rosovitz, Garry SA Myers, Emmanuel F Mongodin, W Florian Fricke, Pawel Gajer, Jonathan Crabtree, Mohammed Sebaihia, Nicholas R Thomson, Roy Chaudhuri, et al. The pangenome structure of escherichia coli: comparative genomic analysis of e. coli commensal and pathogenic isolates. *Journal of bacteriology*, 190(20):6881–6893, 2008.

Kimberly Robasky, Nathan E Lewis, and George M Church. The role of replicates for error mitigation in next-generation sequencing. *Nature Reviews Genetics*, 15(1):56–62, 2014.

Kamil Salikhov, Gustavo Sacomoto, and Gregory Kucherov. Using cascading bloom filters to improve the memory usage for de brujin graphs. *Algorithms for Molecular Biology*, 9(1):1, 2014.

Leena Salmela and Eric Rivals. Lordec: accurate and efficient long read error correction. *Bioinformatics*, page btu538, 2014.

Jared T Simpson and Richard Durbin. Efficient construction of an assembly string graph using the fm-index. *Bioinformatics*, 26(12):i367–i373, 2010.

Jared T Simpson and Richard Durbin. Efficient de novo assembly of large genomes using compressed data structures. *Genome research*, 22(3):549–556, 2012.

Jared T Simpson, Kim Wong, Shaun D Jackman, Jacqueline E Schein, Steven JM Jones, and Inanç Birol. Abyss: a parallel assembler for short read sequence data. *Genome research*, 19(6):1117–1123, 2009.

Jouni Sirén. Compressed suffix arrays for massive data. In *String Processing and Information Retrieval*, pages 63–74. Springer, 2009.

Damian Smedley, Syed Haider, Steffen Durinck, Luca Pandini, Paolo Provero, James Allen, Olivier Arnaiz, Mohammad Hamza Awedh, Richard Baldock, Giulia Barbiera, et al. The biomart community portal: an innovative alternative to large, centralized data repositories. *Nucleic acids research*, page gkv350, 2015.

Helga Thorvaldsdóttir, James T Robinson, and Jill P Mesirov. Integrative genomics viewer (igv): high-performance genomics data visualization and exploration. *Briefings in bioinformatics*, 14(2): 178–192, 2013.

Cole Trapnell, Lior Pachter, and Steven L Salzberg. Tophat: discovering splice junctions with rna-seq. *Bioinformatics*, 25(9):1105–1111, 2009.

Daniel R Zerbino and Ewan Birney. Velvet: algorithms for de novo short read assembly using de bruijn graphs. *Genome research*, 18(5):821–829, 2008.

Jacob Ziv and Abraham Lempel. Compression of individual sequences via variable-rate coding. *IEEE transactions on Information Theory*, 24(5):530–536, 1978.