A CACHE-FRIENDLY BWT LAYOUT

by

Yansong Li

Submitted in partial fulfillment of the requirements
for the degree of Master of Computer Science

at

Dalhousie University
Halifax, Nova Scotia
July 2023

To my family, mentors, friends and everyone else who helped me through all this.

Contents

List of Tables

List of Figures

Abstract

The Burrows-Wheeler Transform (BWT) is a widely used succinct data structure in bioinformatics. However, one of the main concerns that researchers still face when using BWT is its processing time due to the lack of access locality. The (Last to First) LF mapping is derived from the BWT to facilitate backward searches, each step of the LF mapping tends to jump to a completely different spot of the BWT, resulting in at least one cache miss per step.

Our project endeavours to minimize the processing time of BWT through the optimization of BWT layout. This objective is achieved by block partitioning and rearranging the layout of the BWT. Two proxy measures were introduced to represent the cache misses, but the reduced proxies did not correspond to an improved running time. However, the blocked partitioned BWT representation still achieved a 50% speedup with limited memory and a hard disk drive (HDD).

Acknowledgements

First and foremost, I would like to express my sincerest appreciation to my supervisors Dr. Travis Gagie and Dr. Norbert Zeh, for their invaluable and continuous guidance, support, and inspiration during my master's study. Thanks for allowing me to get involved in the research. I would also like to extend my deepest gratitude to my colleagues Nathaniel Brown, Sana Kashgouli, and Nicola Cotumaccio for their help during the data generation of the experiments and companionship during research and study.

Sincerely thanks to Dr. Benjamin Langmead, Dr. Christina Boucher and other researchers in their groups for providing me with very valuable experiences and knowledge from attending their presentations and talks. Special thanks to Dr. Dominik Kempa and Dr. Jouni Sirén for pointing out the research directions.

Last but not least, the completion of my master's degree would not have been possible without the unconditional support and unparalleled love of my parents and grandparents.

Chapter 1

Introduction

Compressing large strings with small alphabets such as the human genome is an essential aspect of research in the field of bioinformatics, where a string is a sequence of characters, an alphabet is a set of distinct symbols or characters from which strings are formed, and the human genome refers to the complete set of DNA (deoxyribonucleic acid) in a human organism consists of A, T, C, G. According to the National Centre of Biotechnology Information (NCBI), human chromosome 19 alone has almost 59 million bases [27]. Succinct data structures are one of the main approaches for compressing large datasets while continuing to support access and other queries, and the Burrows-Wheeler Transform (BWT) is one of the most popular techniques because of its ability to exploit local redundancy, offer no information loss, handle repetitive data well, allow for parallelism and random access, and remain simple and efficient in implementation [24]. The BWT prepares an input string for lossless data compression by restructuring it [3]. A string of characters is transformed to its BWT by listing all the rotations in lexicographical order. This is called the Burrows-Wheeler Matrix (BWM). The last column of the matrix is the BWT. The run length compressed BWT (RLBWT) is a BWT that is compressed using run-length encoding, where a "run" means a sequence of consecutive identical character [29].

Due to its satisfactory performance, different sequence aligners and data structures were developed from BWT such as Burrows-Wheeler Aligner (BWA) [21], Bowtie [20] and r-index [9], which index genome by employing read alignments approach.

The key primitive used in many queries on the BWT and in reconstructing a string from its BWT is called Last-to-First (LF) Mapping. The i^{th} occurrence of a character c in the last column of the BWM and the i^{th} occurrence of the same character c in the first column correspond to the same occurrence in the input string [32]. The last character of each BWM row is the character before the first character in the same row. Thus, the text or any substring of it can be reconstructed by starting at the

position of its last characters in the BWT, and repeatedly jumping to the character in the BWT that corresponds to the occurrence of the current characters in the first column, which is called an LF step. Unfortunately, consecutive characters in the original text tend to be stored far apart in the BWT. This causes at least one cache miss per LF step for most LF steps [32] As a result, the inversion of the BWT and substring extraction as part of other queries tend to be slow [7].

Our hypothesis in this thesis is to improve the access locality of LF steps through the optimization of BWT layout using block partitioning and rearranging the blocks, so the processing time of BWT would be minimized. This requires a data structure to store the BWT that supports the LF steps cache efficiently.

Sirén et al. proposed a block partitioning approach to achieve better access locality of graph operations on BWT-compressed graphs [36]. In this thesis, we investigate whether this block partitioning approach combined with other techniques can also speed up LF steps and, thus, pattern matching and other text-oriented operations on the BWT. We will focus on BWT inversion as the operation whose performance to improve. Since the bottleneck of BWT inversion is the (lack of) cache locality of LF steps, we expect that other LF-based operations experience the same speed-up as we observe for BWT inversion in our experiments.

During the development of our cache-efficient BWT data structure, we used two measures of (cache) locality, the total distance of all LF steps between different clusters and the total number of LF steps that cross different clusters, which are both novel proxy measures of the number of cache misses. Our final experiments evaluate the speed of BWT inversion using a full implementation of our data structure based on the Succinct Data Structure Library (SDSL) [10].

A small string *bananaband*\$ was used to demonstrate the feasibility of improving the access locality through block partitioning and clustering. As shown in Table 1.1, the BWT array BWT and LF steps array LF of *bananaband*\$ were built. Then, LF was block partitioned with a preset block size and the corresponding block numbers (i.e., length of the LF steps array divided by the preset block size) of LF were recorded in the permuted LF steps array $P[LF]$. Since the length of the string is 11, a block size of 3 was chosen as a guess of the optimal block size. Suppose we draw a graph whose vertices are the blocks 0 to 3 for our running example "bananaband\$", in which

blocks	i	BWT[i]	LF[i]	P[LF[i]]
0	0	d	7	2
0	1	n	8	2
0	2	n	9	3
1	3	b	5	1
1	4	b	6	2
1	5	$	0	0
2	6	a	1	0
2	7	n	10	3
2	8	a	2	0
3	9	a	3	1
3	10	a	4	1

Table 1.1: The blocks, BWT, LF and permuted LF of *bananaband*$

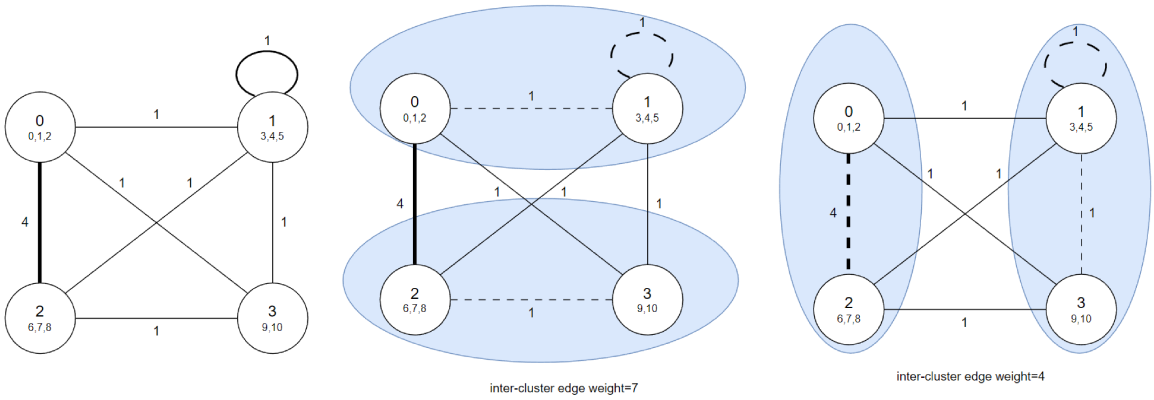

Figure 1.1: The block partitioned graph, normal layout, and optimized layout of the BWT of *bananaband*$

the edge (u, v) has weight equal to the total number of times $\mathrm{LF}(i) = j$ for i in block u and j in block v or vice versa. This graph is shown on the left of Figure 1.1 but we omit 0-weight edges for clarity. For example, edge $(0, 1)$ has weight 4 because $\mathrm{LF}[0] = 7$, $\mathrm{LF}[1] = 8$, $\mathrm{LF}[6] = 1$ and $\mathrm{LF}[8] = 2$ (see column **LF[i]** in Table 1.1). If we lay the BWT out normally and consider clusters $\{0, 1\}$ and $\{2, 3\}$ of blocks, as shown in the center of Figure 1.1, then the total weight of inter-cluster edges is 7, only two LF steps are within the same cluster. On the other hand, if we permute the blocks and consider clusters $\{0, 2\}$ and $\{1, 3\}$ of blocks, as shown on the right of Figure 1.1, then the total weight of inter-cluster edges is 4, then 5 LF steps are within the same clusters, which presumably causes fewer cache misses. This suggests that permuting BWT blocks may improve access locality.

Since the RLBWT provides the data compression necessary to store pangenomic datasets [29], where a pangenomic dataset consists of a set of genome sequences that provides a comprehensive view of the genetic diversity within a species, our primary goal is to improve the cache efficiency of querying the RLBWT. Our experiments revealed that, somewhat counterintuitively, the compression of the RLBWT hurts the cache locality of LF steps using our data structure. This makes our data structure unsuitable for storing pangenomic datasets because pangenomic datasets take huge amounts of space without run length compression. Therefore, the optimization target shifted to uncompressed BWT.

We chose the 14.62 GB Douglas Fir genome as an experimental dataset for optimizing uncompressed BWT, computed the improvement in the total weight of the inter-cluster edges for block sizes between 1.5 and 15 million bases and cluster sizes between 15 million and 1.5 billion, and tested the time to extract 5000-base substrings on a virtual machine with 1 GB of memory on a Dell Inspiron 14-7472 with an Intel i7-8550U processor, consisting of 8 cores running at 1.80 GHz and 16 MiB L3 cache, 1 TB ST1000LM035-1RK172 disk (hard disk drive), 8 GiB of DDR3 main memory and a swap file of 8 GB. Comparisons of the extraction times between a single wavelet tree of the entire Douglas Fir that represents the traditional layout of BWT and a set of permuted block wavelet trees that represent the optimized layout were conducted. We observed up to a 50% speedup with a block size of 9 million bases, but we also observed up to a 30% speedup even when the block size and the cluster size were both 15 million bases, when there should be no clustering and thus no speedup. (In fact, the software we used still clustered even when it should not, so we re-ran the experiment without clustering and observed the same speedup). Further experimentation showed that blocking alone accounted for all or nearly all the speedups, with clustering working in theory but not in practice.

Due to the inefficiency of clustering and the optimal block size being much larger than we expected, another block partitioning representation of wavelet trees was applied to substitute the block wavelet trees built by us, which was the Fixed block compression boosting (FBB) wavelet tree by Gog et al. FBB wavelet trees employed a two-fold block partitioning using block and superblock. Different size combinations of block and superblock were tested and the best extraction speed was obtained with

blocks of 2^{16} bases and a superblock 2^{24} with a disparity of less than 4% compared to the block wavelet trees. Furthermore, 2^{16} and 2^{24} are the maximum limits of the block and superblock sizes, the FBB wavelet trees can not outperform the block wavelet trees with such size limits eventually. Hence, the optimal block partitioned representation of BWT was still observed with the block wavelet trees using a block size of 9 million bases.

The speedup that we gained from block wavelet trees is significant but under very limited conditions. However, we still think the block partitioning approach and the data structure that we developed could be useful when the available memory is significantly smaller than the dataset that researchers are investigating. Furthermore, it was unexpected that the rearrangement of BWT blocks was inefficient in practice even though it decreased the number of inter-cluster LF steps and their distances, but we still think that there exists a better BWT layout that offers better access locality. Different approaches of layout optimization or different proxy measure of cache misses could be the directions of further research of BWT layout optimization.

The structure of the rest of the thesis is as follows: Chapter 2 will provide a review of relevant literature. In Chapter 3, we discuss our approach to (attempt to) improve the cache efficiency of LF steps on the RLBWT including experimental results. Chapter 4 focuses on improving the cache efficiency of LF steps on the BWT. Finally, the thesis will conclude with a summary of the results from both experiments, along with directions for future research in Chapter 5.

Chapter 2

Literature Review

2.1 Succinct Data Structure and SDSL

Succinct data structures are data structures that represent datasets efficiently while minimizing storage space requirements, different types of succinct data structures were used in the thesis. Jacobson et al. introduced these data structures in 1989. They are especially beneficial for applications that require data to be processed in real-time and stored in memory for faster access [15].

Various instances of succinct data structures have been developed. One of the most widely used examples is the succinct trie. A trie is a tree-like data structure used to hold a collection of strings, where each node in the tree represents the prefix of a string within the collection, and each leaf represents the entire string [39]. The succinct trie enables effective string searches, string insertion and deletion within the collection. Moreover, succinct arrays and succinct bitvectors are also considered popular succinct data structures. The succinct array is a compact variant of an array that permits rapid access to individual array elements [10]. The succinct bitvector permits rapid access to individual bits, as well as operations such as rank, which are used to count the number of bits set to 1 within a given range of the vector [10]. Both are commonly exploited in applications for data compression and retrieval.

The SDSL by Simon Gog et al. provides an assortment of succinct data structures, including succinct tries, succinct arrays, succinct bit vectors, and wavelet trees among others [10]. The SDSL is designed for handling data structures that could be represented succinctly or compactly, allowing for more effective memory utilization and faster processing times. The SDSL applies to a variety of applications, including text processing and data compression. For the majority of this project, the SDSL is used to create wavelet trees, the BWT, and suffix arrays, which are discussed in the following sections.

b	a	n	a	n	a	b	a	n	d	$
a	n	a	n	a	b	a	n	d	$	b
n	a	n	a	b	a	n	d	$	b	a
a	n	a	b	a	n	d	$	b	a	n
n	a	b	a	n	d	$	b	a	n	a
a	b	a	n	d	$	b	a	n	a	n
b	a	n	d	$	b	a	n	a	n	a
a	n	d	$	b	a	n	a	n	a	b
n	d	$	b	a	n	a	n	a	b	a
d	$	b	a	n	a	n	a	b	a	n
$	b	a	n	a	n	a	b	a	n	d

Figure 2.1: Rotations of "bananaband$"

2.2 Burrows-Wheeler Transform

The BWT is a well-known text compression and sequence analysis tool that has become commonly recognized since Michael Burrows and David Wheeler initially introduced it in 1994 [32]. The central concept of the BWT is to convert a given input string into a new string that is more easily compressible. Specifically, it rearranges the characters of the input string so that related characters are clustered together in a circular manner [3]. Due to the high degree of character repeats in the rearranged string, it may be efficiently compressed using techniques such as run-length encoding or entropy encoding. These properties make the BWT particularly helpful for manipulating DNA or protein sequencing data in bioinformatics [3]. To compute the BWT of the input string, all the rotations of the string are listed in a matrix form and then sorted lexicographically to construct the Burrows-Wheeler Matrix (BWM). Once the BWM is built, the BWT of the string can be obtained as the last column of the BWM [32]. For instance, the rotations of *"bananaband$"* are listed in Figure 2.1. Next, the rotations are sorted lexicographically as shown in Figure 2.2 and the last column is the BWT of it.

The BWT can be inverted to the original input string by using a technique known as LF mapping [32]. The LF mapping of the previous example *"bananaband$"* is shown in Figure 2.3. However, one of the disadvantages of BWT is the lack of access locality of LF mapping completion, as each backward step tends to jump to a completely new sector of the BWT [7]. Such a characteristic of LF mapping

F										L
\$	b	a	n	a	n	a	b	a	n	d_1
a_1	b	a	n	d	\$	b	a	n	a	n_1
a_2	n	a	b	a	n	d	\$	b	a	n_2
a_3	n	a	n	a	b	a	n	d	\$	b_1
a_4	n	d	\$	b	a	n	a	n	a	b_2
b_1	a	n	a	n	a	b	a	n	d	\$
b_2	a	n	d	\$	b	a	n	a	n	a_1
d_1	\$	b	a	n	a	n	a	b	a	n_3
n_1	a	b	a	n	d	\$	b	a	n	a_2
n_2	a	n	a	b	a	n	d	\$	b	a_3
n_3	d	\$	b	a	n	a	n	a	b	a_4

Figure 2.2: Sorted rotations (i.e., BWM) of "bananaband$"

leads to poor access locality in BWT, which is one of the reasons BWT has a high computational cost; hence, the BWT layout requires further improvement, which is the goal of this thesis.

2.3 Run Length Encoding

Run length encoding (RLE) is an approach for lossless data compression [8]. The input string is split into runs, which is a maximal subsequence of consecutive identical characters. We use c_l to represent a run, where c is a character from the alphabet of the input string and l is the length of the run [8].

For instance, the DNA sequence "$AAGGGGGGGTTTTCCC$" can be encoded as "$A_2G_7T_4C_2$". The BWT of the previous example "$dnnbb\$anaaa$" can be encoded as "$d_1n_2b_2\$_1a_1n_1a_3$". Every run c, is encoded using a fixed number of bits, d, to represent c, and a fixed number of bits, e, to represent l [8]. Such a compression style with fixed bits for the character and its length makes RLE effective for some of the string datasets but not all of them [8]. When the input string has many long and consecutive repetitions of characters such as "$GGGGTTTT$", it will achieve a better compression rate. For instance, if $e = d = 8$, storing "$GGGGTTTT$" would use 8 bytes, the run length compressed string "G_4T_4" would need only 4 bytes. However, if the input string has little or no consecutive repetitions, then the RLE uses more space than the uncompressed string. For example, "$GTGTGTGT$" shares the same alphabet and length with the previous string but it takes 16 bytes to store its run

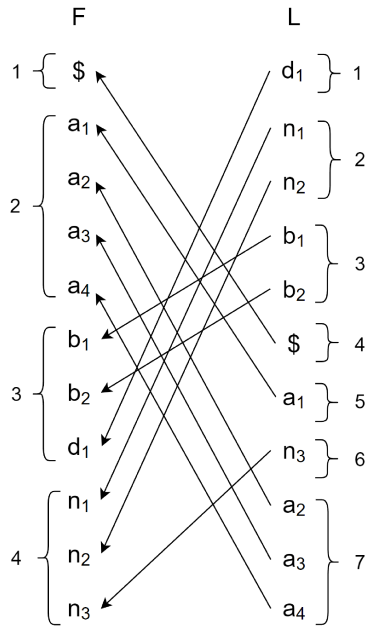

Figure 2.3: LF mapping of "bananaband$"

length compressed form "$G_1T_1G_1T_1G_1T_1G_1T_1$" for $e = d = 8$ [8].

Since BWT tends to reorder the characters in the input string to create runs of consecutive characters, combining RLE with BWT tends to result in a good compression ratio. RLBWT was used as the first target of layout optimization in this thesis.

2.4 Suffix Array

The suffix array is a data structure used to represent a string's suffixes in compressed form [23]. A suffix array of a string S is an array of integers containing the starting positions of the suffixes of S in lexicographical order. The advantage of a suffix array over other data structures supporting string searches, such as a suffix tree, is that it is more space-efficient and easier to construct. Specifically, the suffix array can be constructed in linear time $O(n)$ and linear space $O(n)$ [23].

Suffix arrays have been applied to support many types of string searches; one of the primary applications of a suffix array is pattern matching. The pattern matching problem for a string P is to identify every occurrence of P in a given text T. Using a suffix array over T, this can be achieved in time $O(m + \log n)$, in which m is the length of P and n is the length of T [23]. This is accomplished by conducting a binary

search on the suffix array to find the range of suffixes containing P as a prefix [23].

The suffix array has also become a common bioinformatics tool, especially for DNA or protein sequence alignment and pattern matching across numerous sequences [35]. In this thesis, the suffix arrays were used to build the BWT of a desired string.

2.5 Wavelet Tree

A wavelet tree is a tree-shaped succinct data structure that has been extensively explored and employed for the effective processing and retrieval of vast quantities of data [12]. It was first proposed and implemented by Grossi and Vitter in 2003, and it has been intensively explored and modified since then [12]. Numerous efficient techniques and data structures have been created for the construction and application of the wavelet tree. The format of a wavelet tree is a binary tree that encodes a sequence of values. Each node of the tree divides the values into two groups and maintains information about the size of each group. The wavelet tree provides range searching, rank and select queries, and other data operations [12].

Wavelet tree leverages a binary tree structure to represent data, making it possible to produce condensed representations of vast datasets that can be processed efficiently. Consequently, it facilitates the preservation of compact representations while permitting a wide variety of operations [12]. The wavelet tree has also been utilized in a variety of applications, including text processing, image compression, and range searching. For this thesis, wavelet trees were used as the data structure to store the BWT in the experiments because it offers fast access and rank queries for the backward searches and a great compression ratio of the dataset; for example, the Douglas Fir genome with 14.62 GB was compressed to 2.67 GB with a Huffman-shaped wavelet tree using SDSL.

Despite its benefits, the wavelet tree has disadvantages. It is not always worthwhile to build a wavelet tree of the dataset, especially for smaller ones, because it might be complicated to construct [12]. In addition, a wavelet tree can demand a substantial amount of memory because it saves data at each node. Hence, it could demand a large number of disk accesses for devices when the size of the wavelet tree exceeds the memory size. As a result, the wavelet tree is a robust and adaptable data structure; yet, its compatibility with a particular situation is determined by its requirements.

2.6 Block Partitioning of pBWT

Sirén et al. improve the cache and access locality of the graph extension of the positional BWT (pBWT) using a block partitioning approach, which they used to efficiently index and query haplotype information [36]. According to Sirén et al., haplotype information is first transformed into a genome graph by dividing the haplotype sequences into overlapping k-mers of length k, where k is usually within the range of 31–63 [36]. Next, a de Bruijn graph was constructed from these k-mers. Each node in the de Bruijn graph represented a $(k-1)$-mer and edges connected nodes that overlapped by $k-2$ nucleotides. After that, the de Bruijn graph was simplified by removing spurious nodes and edges, and by collapsing nodes with identical sequences [36]. Such a process resulted in a genome graph, which represented the set of possible paths through the haplotype information.

The construction of the pBWT of the genome graph is similar to the construction of the BWT of a string; However, instead of a single linear string, the pBWT represents the paths through the genome graph [36]. To build the pBWT, starting from the root of the genome graph and recursively traversing the graph in a depth-first order. The pBWT is constructed from the beginning of the genome graph's root and recursively moves in a depth-first order. As each path is visited, the sequence of symbols encountered at each position is recorded [36]. Next, the sub-pBWT of each path is then computed, and the resulting sub-pBWTs are concatenated in the order of the depth-first traversal to form the pBWT [36]. However, the pBWT of the genome graph is computationally intensive, especially for large genomes with many complex regions. Sirén et al. addressed this issue with the block partitioning technique.

The block partitioning approach divides the pBWT into fixed-size blocks, where the block size is a parameter that can be altered based on the available memory and the desired access locality [36]. Each block contains a contiguous range of positions in the pBWT with no gaps between blocks. After the pBWT is divided into blocks, the blocks are reordered so that consecutive blocks are adjacent in memory [36]. This is done using a permutation of the block indices that can be efficiently computed [36].

The result is a new ordering of the pBWT blocks that maximizes cache and access locality and minimizes the number of cache misses and disk accesses during query processing in theoretically optimal. More specifically, each block is stored in the

cache as a single contiguous region of memory; therefore, when querying the pBWT for a specific region of the genome, only the block containing that region needs to be loaded into the cache. The other blocks can be left in the main memory, reducing the number of cache misses. In addition to reducing the number of cache misses, the block partitioning method also reduces the memory usage of the pBWT [36]. This is because only the pBWT for a single block needs to be loaded into memory at any given time, rather than the entire pBWT for the genome graph. This can be especially important for large genomes, where the memory requirements for storing the pBWT of the entire genome graph can be prohibitively high [36]. The success of decreasing the number of cache misses of the pBWT inspired us to try the same idea on the RLBWT and the BWT.

2.7 A Lookup Table of RLBWT

In the traditional approach of conducting backward searches a RLBWT uses rank queries and a sparse bit vector. This data structure can be constructed in $O(\log \log n)$ time and $O(r)$ space, where r is the number of runs in the BWT and n is the number of character in the RLBWT [3]. Nishimoto's lookup table improves the running time of computing LF mapping from $O(\log \log n)$ to $O(1)$ and keeps the same space cost of $O(r)$ [29].

The lookup table is derived from the RLBWT and its LF index, which is c_l and $LF(b)$, where c is the character of the run, l is the length of the run, and b is the index of LF. Furthermore, $LF(b)$ can be represented as a sequence of pairs $\langle k|d\rangle$, the pair $\langle k|d\rangle$ indicates that $LF(b)$ is the d^{th} character in the k^{th} run of the RLBWT, where runs are counted starting from 1 and the characters in each run are numbered starting from 0 [1]. For example, $\langle 2|1\rangle$ in Figure 2.3 means the $LF(9)$, which is the second occurrence of a run of character 'a' at offset 1 in the L column. The lookup table representation of the RLBWT is a table of tuples (c, l, k, d), one tuple per run c_l. The pair $\langle k|d\rangle$ represents the LF index of the first character x in c_l. For instance, the lookup table of *bananaband*\$ is shown in Table 2.1.

The inversion of RLBWT starts with the last character of the RLBWT \$, which is the 0^{th} character in the 4^{th} run. According to the lookup table, the character before it is the 0^{th} character in the 1^{st} run, which is a 'd'. Based on the entry in the 1^{st} row

of the table, the character before it is the 0^{th} character in the 6^{th} run, which is an 'n'. Next, as reported in the table, the character before the entry in the 6^{th} row is the 2^{nd} character in the 7^{th} run, which is an 'a'. Now since the 7^{th} table entry stores $LF(x)$ for the 0^{th} character x in the 7^{th} run, but the indicator currently at the 2^{nd} character in the 7^{th} run. We calculate the LF index of the 2^{nd} character by adding 2 to the d component of the 7^{th} table entry. This gets the indicator to the 3^{rd} character in the 2^{nd} run. However, the second run only has a length 2. Thus, we need to cross the boundary into the next run. The 3^{rd} character from the beginning of the 2^{nd} run is the same as the 1^{st} character from the beginning of the 3^{rd} run. In general, we may be still looking at an index beyond the length of the current run. In this case, the indicator continues skipping to the next run and decreasing the offset within the run by the length of the current run until the offset is less than the length of the current run. In this case, $1 < 2$, so the character before 'a' is the first character in the 3^{rd} run, a 'b'. According to the 3^{rd} table row, the character before the 0^{th} character in the 3^{rd} run is the 0^{th} character of the 4^{th} run. The character before the 1^{st} character in the 3^{rd} run is thus the 1^{st} character of the 4^{th} run. Since the 4^{th} run has length 1, this is the same as the 0^{th} character of the 5^{th} run, an 'a'. Its predecessor is the 0^{th} character of the second run, an 'n', and the predecessor that 'n' is the 0^{th} character of the 7^{th} run, an 'a'. Then, what precedes the 'a' is the 1^{st} character of the second run, an 'n'. What precedes that 'n' is the first character of the 7^{th} run, and 'a'. Its predecessor is the 2^{nd} character of the second run. Since the second run has length 2, this is the same as the 0^{th} character of the 3^{rd} run, a 'b'. Finally, the predecessor of 'b' is the 0^{th} character of the 4^{th} run, which is our end of string character \$. We have successfully reconstructed the string "bananaband\$".

Nishimoto's lookup table improves the running time of LF mapping, but considering the common size of the input string for RLBWT, like a chromosome of the human genome, $O(r)$ space is still a huge number, which means it has to be stored in memory or even in the disk. Therefore, using the lookup table to compute the LF mapping of RLBWT will still cause a significant number of cache misses.

Index	Character(c)	Length(l)	Interval(k)	Offset(d)
1	d	1	6	0
2	n	2	7	0
3	b	2	4	0
4	$	1	1	0
5	a	1	2	0
6	n	1	7	2
7	a	3	2	1

Table 2.1: RLBWT Look-up table of "bananaband$"

2.8 Graph Clustering and METIS

Graph clustering is the technique of partitioning a graph into subgraphs or clusters so that most edges connect vertices in the same cluster [18]. The objective of graph clustering algorithms is to divide the graph into a set number of clusters, with a high degree of similarity inside each cluster and a low degree of similarity between clusters, where similarity means the communication cost between vertices. There are a number of techniques for clustering graphs, including hierarchical clustering, k-means clustering, and spectral clustering [16].

An algorithm of dividing irregular graphs into smaller sub-graphs that was proposed by Karypis et al. in 1994 is widely used [18]. The approach is based on a multilevel system that employs coarsening, initial partitioning, and refining to generate partitions of superior quality. The multilevel k-way partitioning has been demonstrated to be effective in terms of both partition quality and execution time. Karypis et al. reduce the size of the graph by compressing nodes and edges progressively. This coarsening phase is performed until the graph is small enough to be partitioned with an initial partitioning technique. The initial partitioning divides the coarsened graph into a specified number of clusters. Then, Karypis et al. employ a refinement approach to enhance the quality of the partition by locally altering the cluster borders. Moreover, it is shown experimentally that the technique outperforms various well-known graph partitioning algorithms in terms of partition quality and execution time [18].

METIS is a graph partitioning software package that employs Karypis et al.'s algorithm [18]. The software has been continuously updated and enhanced over the

years, and it continues to be one of the most popular and commonly employed graph partitioning tools available. This research uses METIS as its primary graph clustering tool to enhance the BWT layout.

2.9 Minimum Linear Arrangement

The minimum linear arrangement (MLA) problem is an NP-hard problem in graph theory. It is defined as finding a permutation of the vertices of a complete graph that ensures that the total weight of the edges is minimized, where the weight of an edge is defined as the distance between its end-points in the ordering [17].

Many techniques have been developed to address the problem including heuristic algorithms, approximation algorithms, and exact algorithms [37]. With heuristic methods such as the nearest neighbour algorithm and the farthest neighbour algorithm, a solution is constructed by iteratively adding vertices to the ordering based on certain criteria [4]. For instance, the nearest neighbour algorithm selects the closest unordered vertex to the last ordered vertex as the next vertex in the ordering, which is the one that is adjacent to the last ordered vertex and has the smallest weight of the connecting edge [17]. With approximation methods such as Christofides's algorithm, they attempt to provide a solution that is within a specified ratio of the optimal answer. Christofides's algorithm provides a 3/2 approximation for the MLA problem [4]. Exact algorithms guarantee finding the optimal solution and include branch and bound algorithms and branch and cut algorithms, [14]. These algorithms function by systematically examining the solution space and eliminating any branches that cannot lead to a better solution [14].

Some of the approaches that have been mentioned were proposed a while ago, such as the earliest exact algorithms [14], and subsequent studies have been based on them. In recent years, several researchers have proposed novel solutions to the MLA problem, such as the multilevel weighted edge contractions proposed by Safro et al. [31]. The multilevel weighted edge contractions approach groups vertices with high levels of connection into a single super vertex, which is then interpreted as a single vertex at the following graph level. This procedure is repeated until the graph is reduced to a single vertex [31]. The findings of this paper demonstrate that their method for determining the ideal linear arrangement of vertices is highly effective.

As a result, the multilevel weighted edge contractions approach gives a more effective and novel solution to the graph MLA problem and can be used to solve a variety of graph theory and computer science problems.

2.10 Linear Programming and Integer Linear Programming

Linear programming (LP) and Integer Linear Programming (ILP) are mathematical optimization approaches used to optimize a linear objective function subject to linear constraints [5]. In several different areas, including manufacturing, banking, and transportation, LP is applied to optimize the decision-making process. ILP, on the other hand, is a particular example of LP in which all decision variables are limited to integer values [38]. In contrast to LP, ILP is NP-complete [38].

The general form of an LP problem can be shown as the following:

$$\begin{aligned} \text{Maximize/Minimize: } & Z = c_1x_1 + c_2x_2 + ... + c_nx_n \\ \text{Subject to: } & a_{11}x_1 + a_{12}x_2 + ... + a_{1n}x_n \leq b_1 \\ & a_{21}x_1 + a_{22}x_2 + ... + a_{2n}x_n \leq b_2 \\ & ... \\ & a_{m1}x_1 + a_{m2}x_2 + ... + a_{mn}x_n \leq b_m \\ & x_1 \geq 0, x_2 \geq 0, ..., x_n \geq 0 \end{aligned} \tag{2.1}$$

where Z is the objective function, $x_1, x_2, ..., x_n$ are the decision variables, $c_1, c_2, ..., c_n$ are the coefficients of the objective function, $a_{ij}(i = 1, 2, ..., m; j = 1, 2, ..., n)$ are the coefficients of the constraints and $b_1, b_2, ..., b_m$ are the right-hand side values of the constraints [5]. In addition, if it is an ILP problem, then some or all decision variables would be restricted to integer values. Both LP and ILP have been applied across diverse fields. The applications of LP include resource allocation, portfolio optimization, and transportation planning.

2.11 Simulated Annealing Algorithm

Simulated Annealing (SA) has been successfully applied to solve challenging combinatorial optimization problems [19]. It is a meta-heuristic approach that imitates the cooling process of metals. Initially, SA starts with a predefined solution. By

randomly making alterations to the current solution, SA repeatedly generates new solutions. If the new solution decreases the value of the objective function, it replaces the current solution. However, there is still a possibility of acceptance, even if the new solution increases the objective function's value. As the algorithm progresses, the likelihood of accepting such a solution decreases. Accepting a higher objective function enables the algorithm to explore additional areas of the solution space and escape local minima, possibly resulting in a global optimum [19]. SA has been applied to various combinatorial optimization problems.

2.12 FM-Index and Fixed Block Compression Boosting

FM-Index is a popular data structure in bioinformatics and computational biology for efficient encoding and querying of massive text datasets, such as genomic sequences [6]. The FM-Index leverages the features of the BWT and the suffix array to facilitate fast pattern matching, substring search, and other text processing operations [6]. In particular, the FM-Index maintains the BWT, the suffix array, an array C that records the cumulative count of each character in a text string T, and a data structure for quickly responding to range queries on C [6]. Given a pattern P, the FM-Index can be used to find all instances of P in T by conducting a backward search on the BWT. This is accomplished by employing the cumulative count array C to find the interval in the BWT that represents the set of suffixes beginning with P, and then iteratively expanding the interval by following the corresponding characters in the BWT. The reason it needs to expand the interval is that the backward search in the BWT based on C does not necessarily give all the suffixes that start with P. The backward search only gives the range of positions in the BWT that correspond to the characters in P when they are read in reverse order (i.e., from right to left). This operation has a time complexity of $O(m \log n)$, where m is the length of the pattern and n represents the length of T [6]. In bioinformatics and computational biology, the FM-Index has been extensively employed for tasks such as genome assembly. It has been demonstrated that is highly efficient because of its capacity to manage enormous volumes of data and perform rapid pattern matching and substring search [26].

FBB by Gog et al. is one of the most cutting-edge techniques for optimizing the time and space efficiency of FM-Indexes further [11]. FBB is based on the principle

of dividing the BWT of a text string into blocks and applying lossless compression to each block. This reduces the size of the BWT, and consequently, the size of the FM-Index while maintaining the index's functionality. The most difficult aspect of FBB is selecting a compression technique that strikes a balance between compression ratio and decompression speed. Gog et al. investigated several popular compression approaches, including RLE, Huffman coding, and Arithmetic coding, and demonstrated that Arithmetic coding outperforms the others in terms of compression ratio and decompression time [11]. Gog et al. showed that FBB can drastically reduce the size of the FM-Index, often by more than 50%, while preserving the index's rapid search performance [11].

2.13 A low Out-Degree Proof of the Graph Built from RLBWT Look-up Table

The feasibility of decreasing the number of cache misses of backward searches of RLBWT is based on the assumption that the graph built from the RLBWT look-up table by Nishimoto et al. has a low out-degree. The graph built from the RLBWT table is defined as the following: first, the RLBWT table will be block partitioned based on the size of the dataset and the cache size. Next, the blocks are used as vertices $v \in V$, and the jumps across different blocks are used as edges $e \in E$ to build a graph $G = (V, E)$. The weight $w \in W$ of each e in G is determined by the cumulative length of the runs between two vertices $(v, u) \in V$, and the loops from each block to itself are ignored since they do not affect anything. In addition, the following theorem was proven.

Theorem 1. *Let d be the average number of runs in the L column spanned by a run in the F column. The average degree of a vertex v in the graph built from Nishimoto's look-up table is at most $d + \sigma$, where σ is the size of the alphabet.*

Proof. Suppose we gather every b runs into a block. If we pick a character c at random then the expected number of runs of c's in any block is b/σ, where σ is the size of the alphabet. Suppose we also pick a block B at random. If p_c is run to which the start of the first run of c's in B maps (according to the 3^{rd} column of Nishimoto et al.'s table i.e., the Intervals) and q_c is the run to which the start of the last run of c's in B

maps, then $q_c - p_c <= db/\sigma$, where d is the average number of runs in the L column spanned by each run in the F column.

Since the starts of the runs of c's in B map to an interval of runs of expected length db/σ, they map to an interval of blocks of expected length at most $d/\sigma + 1$. Consider the number of blocks to which B points in the graph. The expected number of edges for the character c is $d/\sigma + 1$, so the total expected degree is at most $\sigma(d/\sigma + 1) = d + \sigma$. □

Chapter 3

RLBWT Inversion

The primary goal of the RLBWT inversion experiment was to optimize the processing time of RLBWT by reducing the number of cache misses caused during backward searches. To achieve such an objective, the rows of the RLBWT needed to be permuted so that the total distance of LF steps is minimized. However, it is irrational to rearrange the RLBWT by rows because it requires a table to record the destinations of all the LF steps and the sizes of all the rows. Thus, block partitioning was applied such that consecutive rows of the RLBWT were grouped into blocks with a specific size. Then the RLBWT was rearranged by blocks using clustering. In that case, it only demanded a smaller table that records where every block was stored. First, this experiment converted the RLBWT lookup table that was proposed by Nishimoto et al. [29] to an undirected weighted graph with block partitioning approach by Sirén et al. [36]. Then the graph was input to METIS [18] for clustering. The parameter used to estimate the number of cache misses is the sum of edge weight after clustering. Three kinds of datasets were used for this experiment: the reference genome of Salmonella, copies of human chromosome 19, and a set of noisy strings. The salmonella genome was chosen because it is small and was studied recently in parallel research in my group [2], the human chromosome 19 was chosen because it offers a bigger average run length, and the noisy strings were applied to analyze the relationship between inter-cluster edge weights and the average run length of input genome sequence. The first two datasets' block partitioning and clustering indeed reduced the sum of edge weights between clusters by a factor of 2 to 4 depending on the number of clusters. On the other hand, for noisy strings, the run length compression seemed to hinder decreasing the sum of weight via block partitioning and clustering.

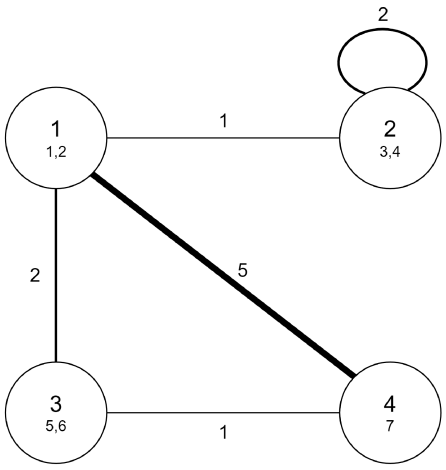

Figure 3.1: Preliminary graph clustering: step 1.

3.1 RLBWT Inversion Methodology

The RLBWT experiment aimed to speed up the processing time of RLBWT by improving its cache efficiency using block partitioning, building the graph of blocks, and clustering the graph based on the RLBWT tables that were proposed by Nishimoto et al. [29].

3.1.1 A Graph Building and Clustering Example

Before conducting experiments on real genome datasets, a small string example was used to demonstrate the feasibility of improving total edge weights by block partitioning and clustering. A lookup table was derived from the string "bananaband$" as shown in Table 2.1.

The block partitioning approach was applied by dividing the table into four blocks, each containing two rows; the fourth block contains one row. Next, the blocks were used as vertices, and the weight of edges is defined as the sum of the lengths of the rows that are the source of the jumps as shown in Figure 3.1. For example, the edge between block 1 and block 4 is 5 because index 2 goes to interval 7 with a row length 2 and index 7 goes to interval 2 with a row length 3 and index 2 belongs to block 1 and index 7 belongs to block 4, the sum of the lengths is 5.

After building the graph, the loops from each block to itself are ignored as they do not affect the number of cache misses during the backward searches. Next, the graph is

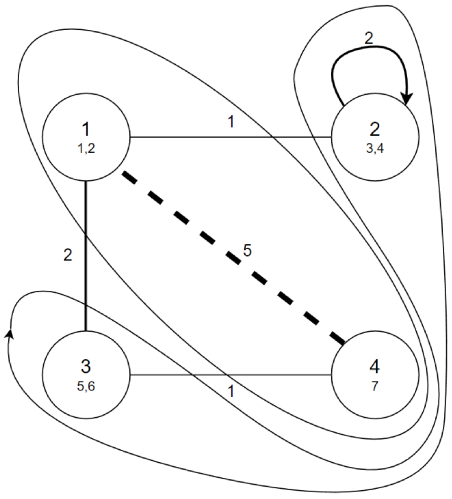

Figure 3.2: Preliminary graph clustering: step 2.

clustered using an algorithm that minimizes the sum of edge weight between clusters. This is because the weight of the edges represents the number of backward steps between two blocks and each step is assumed to cause at least one cache miss. The process of clustering just mentioned is repeated until all the blocks are clustered, and all the clusters are of similar size. The result of clustering the graph of "bananaband$" is shown in the Figure 3.2.

As can be observed in Figure 3.2, the edge with the highest weight, which connects block 1 and block 4 has been included in the same cluster, while the other three edges have been retained. Furthermore, block 1 and block 4 have been grouped, and blocks 2 and 3 have been clustered together in the same group. As a consequence, the total weight of inter-cluster edges in this graph is 4.

As a baseline, we also constructed a clustering that groups consecutive blocks as a simulation of the current LF mapping processing approach, and the total weight of inter-cluster edges are computed, as illustrated in Figure 3.3. The sum of the inter-cluster edges increases to 7. This small dataset suggests that the number of cache misses during backward searches would decrease by rearranging the look-up table because the sum of edge weights decreased. To further investigate the feasibility of our idea, the same approach was applied to real genome datasets by building graphs and clustering them with METIS.

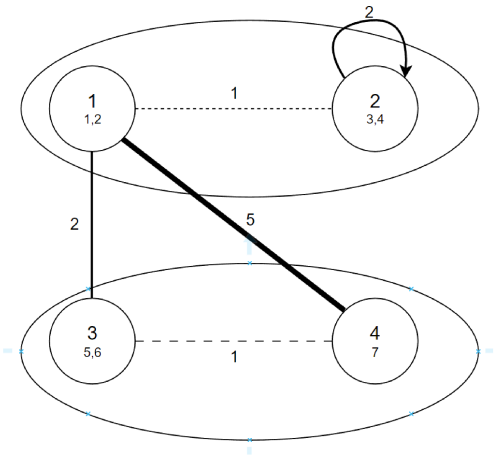

Figure 3.3: Preliminary graph clustering: step 3.

3.1.2 Clustering the RLBWT of Real Genome Data

The previous example suggests that block partitioning and cluster should help to improve the cache efficiency in LF steps; hence, the improvement of cache efficiency using such an approach is evaluated with real genome datasets in this section. Due to the size of the genome datasets, the clustering was conducted using METIS. RLBWT look-up tables were built from real genomes. After reading the look-up table from a binary file, the procedure mentioned in preliminary clustering was followed to cut the look-up table into blocks based on the L1 cache size. For this particular experiment, 1024 rows were used, which is small enough to be stored in a common L1 cache of 64K. Then the graph was built using the procedure mentioned in Section 2.13. The graph construction for genome datasets was implemented using C++.

By Theorem 1, the average degree of V_i is at most $d + \sigma_i$. Both d and σ can be expected to be small. A low average out-degree indicates the graph is cache-friendly and cluster-able. After building the graph using the block-partitioned RLBWT table, it was clustered using METIS with different numbers of clusters to observe the differences in the total weights of the remaining edges. The algorithm that was used for METIS clustering was multilevel k-way partitioning [18].

As a comparison to the METIS clustering approach, the graph G was also clustered sequentially, which means that the blocks are grouped into clusters of equal size such that consecutive blocks in the RLBWT look-up table end up in the same block. The total weight of the remaining edges of sequential clustering is then calculated as W_{s}.

If $W_{\mathrm{r}} < W_{\mathrm{s}}$ significantly, then it indicates that the method proposed by Sirén et al. can decrease the number of cache misses of the RLBWT table in Nishimoto et al.'s construction.

3.2 RLBWT Inversion Results

3.2.1 Salmonella Dataset

The salmonella genome was chosen because it was relatively small compared to other genome sequences such as the human reference genome and it is widely studied by researchers who work on foodborne bacterial disease. Furthermore, there was an existing RLWBT look-up table built for the salmonella genome sequence from Brown [2], which makes the salmonella genome sequence a good start for experiments on real genome datasets. As a result, the RLBWT table was built from the Salmonella genome sequence to verify if the block partitioning and clustering offer a lower sum of edge weight like the short-string example that has been analyzed.

The sequence length is 145,595,456, and the number of runs is in the RLBWT 12,823,516, resulting in an average run length of 11. The graph built from this table has 12,523 nodes and 62,556 edges, with a total edge weight of 145,589,783. The results of METIS clustering and sequential clustering are presented in Table 3.1 The inter-cluster edge weight of METIS is observed to reduce by a factor of 3 when the cluster number is 10 compared to sequential clustering. However, the difference is relatively small when the cluster number is 1000.

#Clusters	10	50	100	500	1000
W_m	41,393,086	56,002,575	62,137,611	103,647,402	128,998,476
W_s	127,980,098	142,549,115	143,815,338	145,258,185	145,482,565
W_s/W_m	3.09	2.55	2.31	1.4	1.13

Table 3.1: Salmonella dataset clustering results

As shown in Figure 3.4, until 100 clusters, there is a clear difference in weights between METIS clustering and sequential clustering. The inter-cluster weights using METIS are still 2 times smaller than when using sequential clustering when the cluster number is 100. Once the cluster number is over 100, the weight difference shrinks significantly.

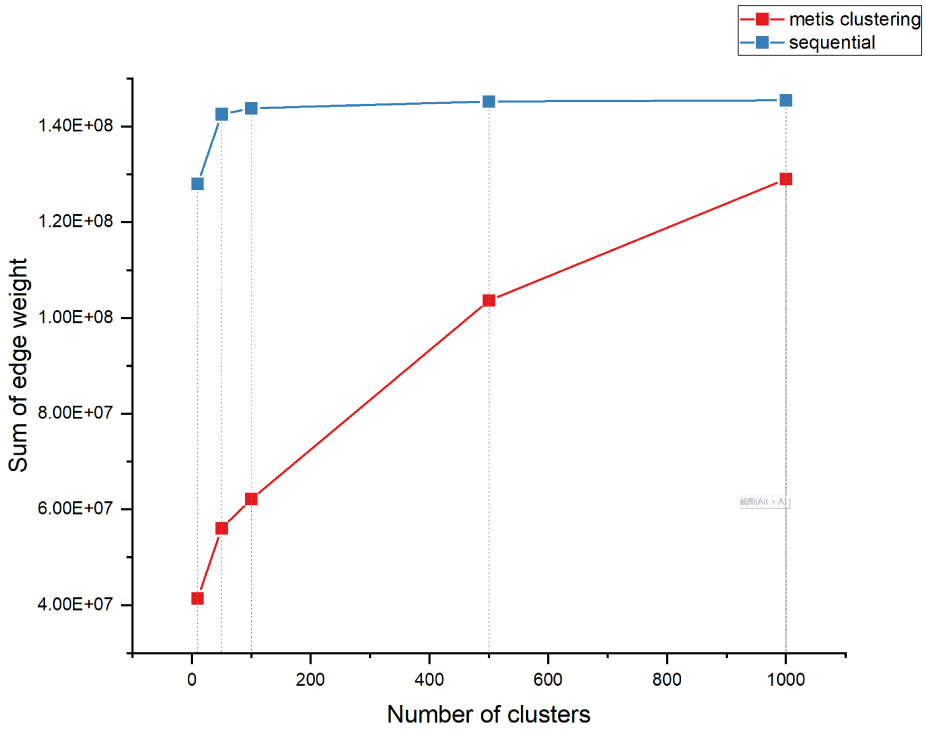

Figure 3.4: Salmonella weight change before and after METIS clustering.

As a result, the salmonella dataset demonstrated a decrease in edge weights by a factor of 3 when the cluster number was 10. However, the average run length was not very large in this dataset. Therefore, we also conducted experiments on human chromosome 19 to determine if it provides better results since the human genome is a primary research target in bioinformatics.

3.2.2 Human Chromosome 19 Dataset

In this section, the clustering results generated from the human chromosome 19 dataset are presented. The reason the dataset was switched to human chromosome 19 is that the decrease in the sum of inter-cluster edge weights was not significant. A hypothesis was made that the limited decrease was due to the average run length of the salmonella genome sequence being rather small, so it was expected the human chromosome 19 sequences would provide a more significant decrease in terms of the sum of edge weights with block partitioning and clustering because of its much bigger average run length. Moreover, the human reference genome is one of the most studied genome sequences in practice. The optimized layout would be more valuable

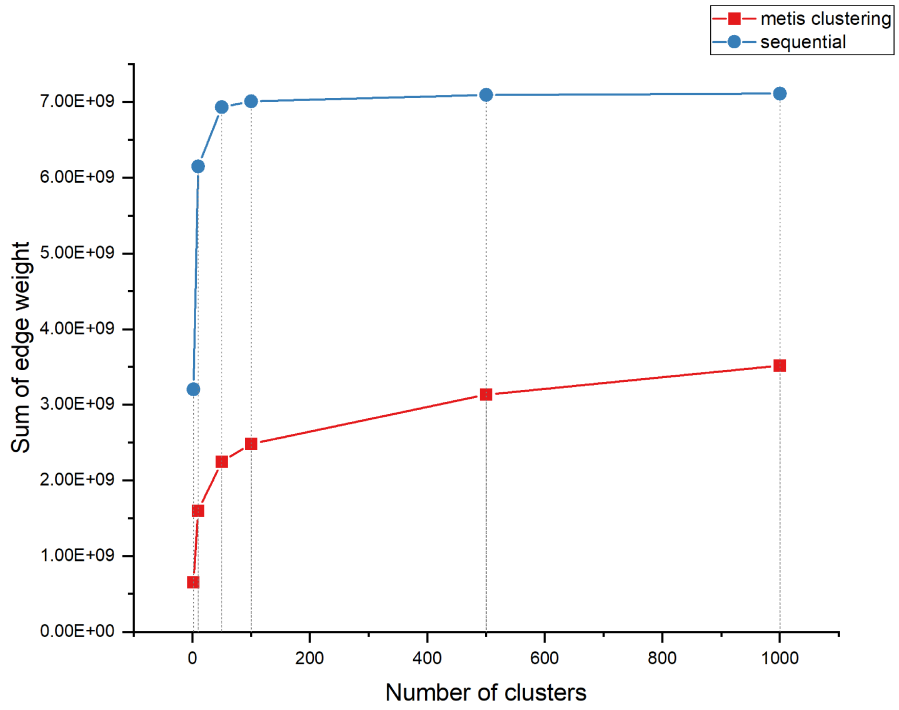

Figure 3.5: 128 copies human chromosome 19 weight change before and after METIS clustering.

to real-world research, if the edge weights decrease is more significant on the human chromosome 19.

First, the experiment was conducted with 128 copies of chromosome 19. The original chromosome 19 sequences include 7,568,015,632 characters, 34,053,959 runs, and a much higher average run length of 222.236. This graph has 33,256 nodes and 166,104 edges. The total weight of the edges is 7,142,005,530. As shown in Table 3.2, comparing the METIS clustering result and sequential clustering result of this dataset, it shows that METIS clustering reduces the weight of inter-cluster edges almost by a factor of 4 when the cluster number is 10, and there still exists a difference of a factor of 2 when the cluster number is 1000.

#Clusters	10	50	100	500	1000
W_m	1,601,510,950	2,249,127,139	2,483,372,420	3,132,482,264	3,518,451,513
W_s	6,149,247,602	6,934,085,819	7,010,231,320	7,093,603,030	7,110,940,769
W_s/W_m	3.94	3.09	2.82	2.26	2.20

Table 3.2: Human chromosome 19 dataset clustering results

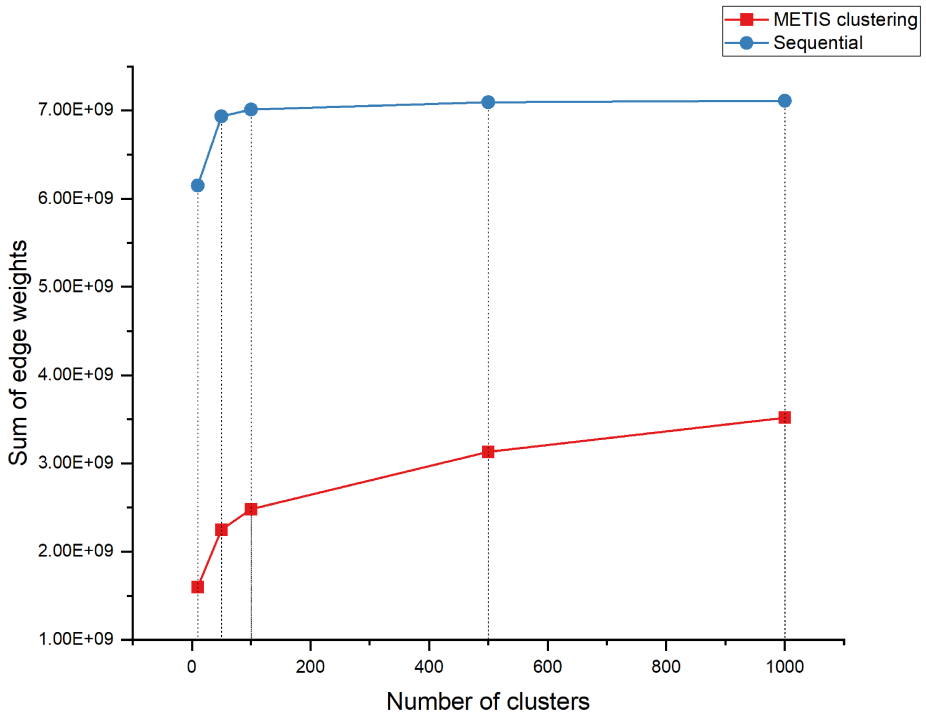

Figure 3.6: 256 copies human chromosome 19 weight change before and after METIS clustering.

From Figure 3.5, it can be observed that there is a notable difference between the results of METIS and sequential clustering for the human chromosome 19 dataset. Additionally, the weight of METIS clustering does not increase as rapidly as it does in the case of the Salmonella dataset when the cluster number is over 100. However, the weight of METIS clustering continues to rise after the cluster number exceeds 100, and it is approximately three times smaller than sequential clustering when the cluster number is exactly 100.

Next, the experiment was conducted using 256 copies of chromosome 19 with the expectation that may lead to a more significant decrease in the sum of edge weights because the average run length is proportional to the number of copies. As shown in Figure 3.6, the decrease of the sum of edge weight keeps the same ratio from 2 to 4 depending on the number of clusters with an increased average run length of 2 times of 128 copies. Therefore, increasing the number of chromosome copies did not improve the results but instead increased the weight of each edge in the graph, the relationship between the decrease of the inter-cluster edge weights and the average

run length remained undefined. The best result obtained from chromosome 19 was a decrease in the sum of edge weights by a factor of 4 when the cluster number is 10; however, a reduction of a factor of 2 to 3 would be much more realistic based on the size of the caches.

3.2.3 Noisy String Dataset

To investigate the relationship between the decrease of the sum of edge weights and the average run length of the input sequence further, an experiment was conducted to cluster artificial sequences with different numbers of flipped characters (i.e., a character can be substituted to any other character in the English alphabet). The sequence alphabet remained $[A, T, C, G]$, and it had a total length of 100,000,000 characters consisting of 100 copies of a subsequence of length 1,000,000 characters that were randomly generated. The sequence $noisy_0$ means that no characters were flipped. As a result, the BWT consists of runs of a length of at least 100. $noisy_1$ means that every character was flipped, $noisy_{10}$ means that every 10 characters were flipped, and so on until $noisy_{10000}$. The RLBWT of each noisy string and the RLBWT lookup table were generated accordingly. Fewer flipped characters mean a higher average run length in the RLBWT as shown in Table 3.3 because the character flipping tends to break runs. Figure 3.7 shows the METIS clustering result of six input sequences. As shown in the figure, bigger average runs indicate less significant edge weight decreases when the dataset is run-length compressed. However, when the average run length exceeds a certain threshold, the edge weights start to plateau.

1/X flipped	#runs	average run length
0	749369	133.446
10000	844614	118.397
1000	1728294	57.8605
100	9793615	10.2107
10	53620161	1.86492
1	75001430	1.33331

Table 3.3: The number of runs and average run lengths of noisy strings

The results of the noisy strings experiment showed the opposite of the expected outcome that higher average run lengths of the RLBWT of the input sequence would demonstrate a lower sum of edge weight. Such a scenario in theory indicates that

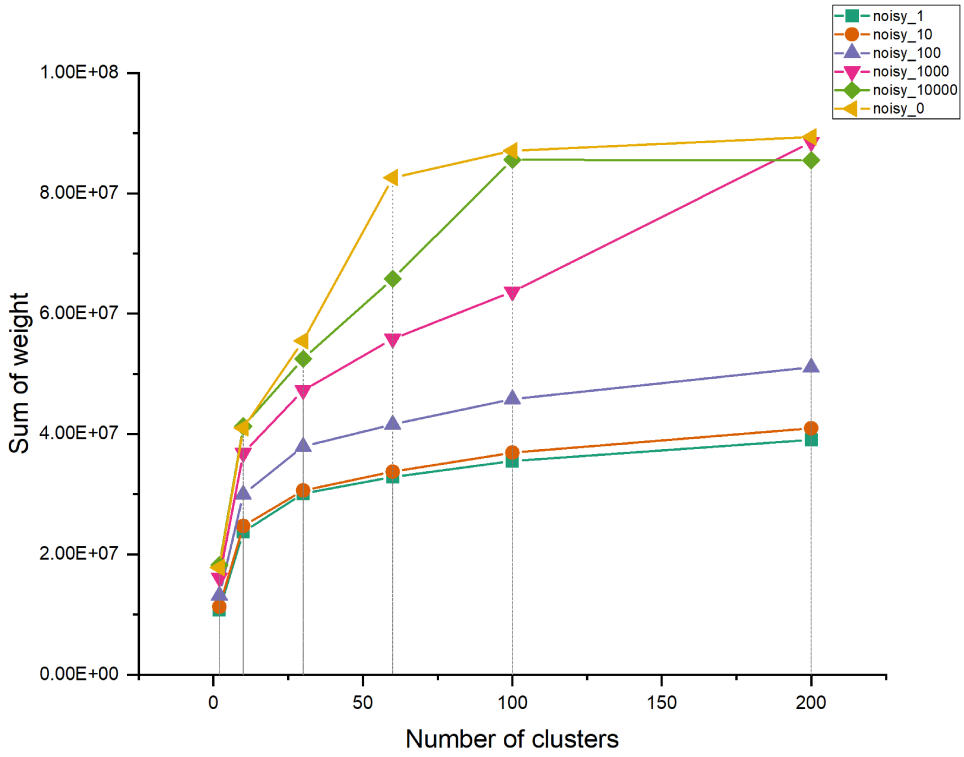

Figure 3.7: Noisy strings weight change with METIS clustering.

longer average run lengths cause more cache misses in practice when the BWT of the input sequence is run-length compressed, which means that run-length compression is redundant uncompressed BWT would demonstrate a better performance compared to RLBWT in terms of fewer cache misses.

3.3 Discussion of RLBWT Experiment

Three types of datasets were tested in this experiment: the Salmonella reference genome, 128 copies and 256 copies of the human chromosome 19, and noisy strings with different numbers of flipped characters. Based on the results obtained from the Salmonella dataset and the human chromosome 19, we can say that METIS clustering provides statistically better results (i.e., a decrease of the sum of edge weight of at least a factor of 2) compared to the sequential orders of the BWT when the number of clusters is less than 100. Moreover, the most significant decreases in both datasets occur with 10 clusters, where the clustering offers a decrease of a factor of 3.84 for the chromosome 19 dataset and a factor of 3.09 for the Salmonella dataset. Additionally,

if the size of the caches was taken into consideration, a decrease of a factor of 2.31 was obtained for the Salmonella dataset and a factor of 2.82 for the 128 copies of human chromosome 19 with 100 clusters because the L1 and L3 cache size of the Waverley server is 64KB and 16MB, respectively.

On the other hand, the noisy string experiment reveals that the run length compression on BWT could cause more cache misses than uncompressed BWT when optimizing the layout of RLBWT. The underlying reason is that each block contains a fixed number of rows of the RLBWT, and the length of each row (i.e., the length of a run) of the RLBWT varies. Hence, there would be blocks that contain huge numbers of rows of BWT, which make them significantly larger than other blocks. In that case, the majority of blocks are not rearranged and most of the edges would still jump to greatly varying locations in the BWT, which make the clustering procedure inefficient. The flipped characters in the sequences help to decrease edge weights because what used to be a single run (i.e., a single row in the look-up table) now becomes multiple shorter runs, which means compressing BWT into runs and then using clustering to improve the access locality is inefficient, and abandon run length compression would be the choice under such a scenario. Therefore, the run-length compression may not be the best compression algorithm to apply on BWT because it was expected that the run-length compression would help decrease inter-cluster edge weight with block partitioning and clustering. Furthermore, the sum of inter-cluster edge weights is just a proxy measure of the number of cache misses that occurred during backward searches of RLBWT in theory. Running time experiments still need to be conducted in the next stage of the research to see Whether the optimized layout would decrease the processing time of BWT in practice.

Chapter 4

BWT Inversion

Based on the experiments conducted on RLBWT, we have learned the optimization of BWT layout using block partitioning and clustering does not work with run-length compression because different lengths among runs of RLBWT and a fixed number of runs of blocking would cause inefficient clustering. Due to such circumstances, the run-length compression was discarded and the focus of the experiment shifts to developing a cache-efficient layout using regular BWT instead of RLBWT. This experiment was divided into two main parts: first, optimizing the layout of uncompressed BWT to minimize the number of LF steps that cause cache misses and their distances with block partitioning the BWT, building the graph of BWT blocks, and clustering it. Also, the block order within each cluster is permuted using ILP or SA because the blocks are still sequentially stored inside each cluster and the cluster size is still relatively too big to fit in the cache that is built from real reference genomes, so some LF steps within the cluster would still cause cache misses. Second, build a data structure with this optimized layout of BWT to verify if it provides any improvement in terms of access locality compared to the normal layout of the BWT. The results of this experiment would be presented and discussed in this chapter too.

4.1 BWT Inversion Methodology

The methodology of this experiment was composed of six steps. First, the BWT was block partitioned with a specified block size that depends on the size of the dataset. Second, a graph of the BWT blocks was built based on the number of LF jumps between blocks, which is the same type of graph that was used in the RLBWT experiment. Third, the graph of BWT blocks was clustered using METIS to obtain a lower sum of inter-cluster edge weights. Fourth, the order of the blocks in each cluster was optimized using an ILP solver to solve the MLA problem; the goal was to obtain a permutation of the blocks that further improves the cache efficiency of the layout by

bringing the sources and targets of LF steps in the same cluster closer to each other. Fifth, a data structure was built for backward searches with the optimized layout of BWT. Finally, the backward search of the BWT was performed on different datasets to compare the performance with the sequential layout of BWT. Figure 4.1 is a flow chart that helps to visualize the entire process of optimizing the layout of BWT and comparing its performance with the sequential layout.

The methodology of the BWT experiments employed the same approach as the RLBWT experiments to reduce the sum of inter-cluster edge weight because we obtained meaningful decreases in the previous experiments using this approach. Moreover, it was expected the uncompressed BWT would provide more significant decreases since the run-length compression trended to make the clustering process inefficient. In addition, since the lookup table of Nishimoto et al. only supports RLBWT, we used the SDSL to represent the BWT and support LF steps [10].

4.1.1 Previous Approach Validation

The "bananaband$" example from the introduction showed a decrease of inter-cluster edge weights using block partitioning and clustering, which indicated the approach would also work on uncompressed BWT. However, the size of the dataset was limited, bigger datasets were needed to verify the efficiency of blocking and clustering. Meanwhile, datasets like the human reference genome were too big to apply in the current stage of the experiment because we were still verifying the correctness of the graph-building code. Therefore, we needed a bigger dataset than "bananaband$" but not too big, and the Hamlet from Shakespeare [33] with 162,850 characters was chosen. The Hamlet dataset was processed as mentioned using block partitioning, building a graph based on BWT blocks and clustering the graph. The clustered graph of the BWT of Hamlet gives a sum of the edge weights of 75,116 with a block size of 100 characters and a cluster size of 100 blocks. The unblocked and unclustered arrangement gives 162,848. Thus, blocking and clustering reduces the total number of inter-cluster edges by a factor of 2.17, similar to the decrease that was obtained in the RLBWT experiment. In particular, using the uncompressed BWT did not increase the effectiveness of blocking and clustering on this dataset, contrary to our

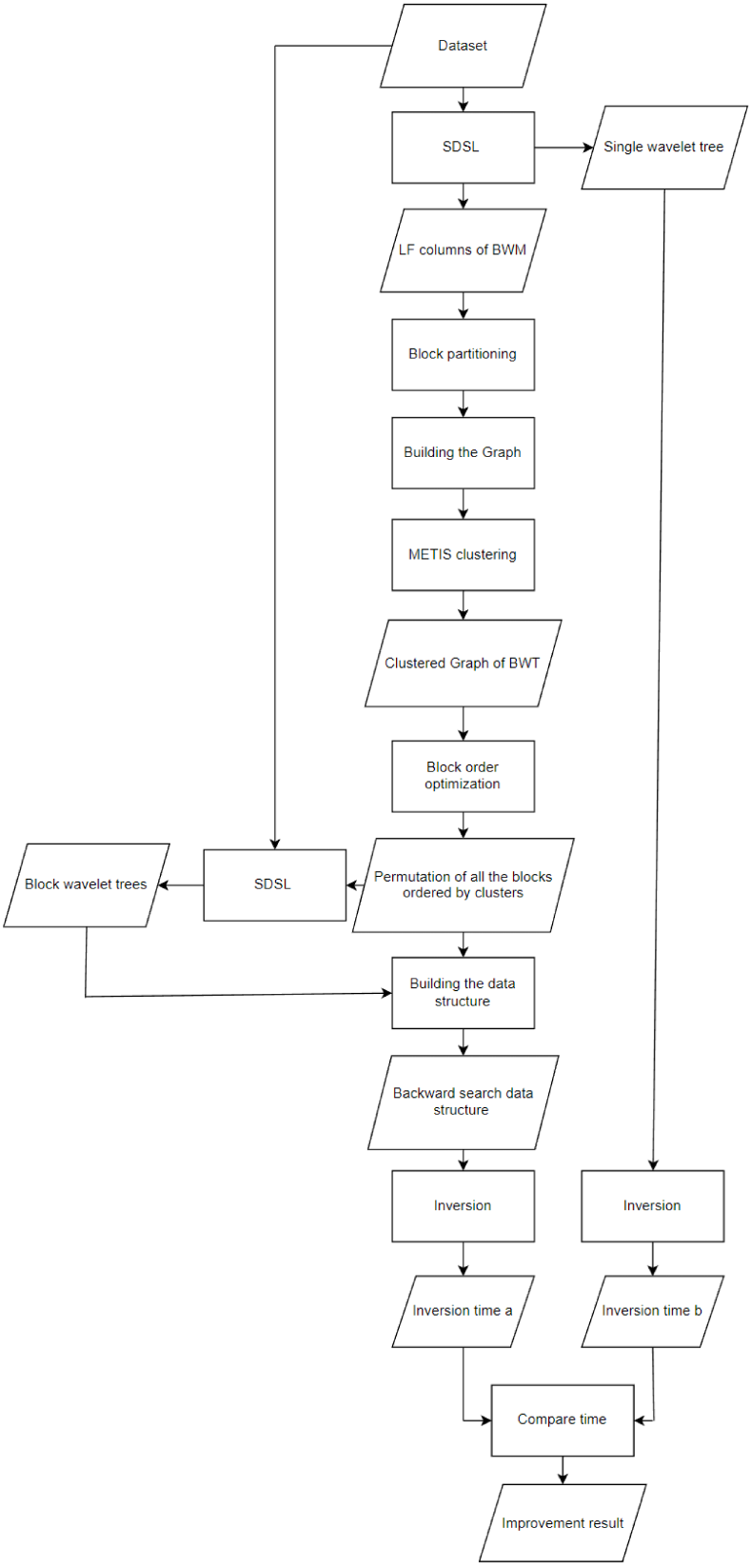

Figure 4.1: A flow chart of BWT inversion experiment.

experiment. This result could be because Hamlet has a shorter length and larger alphabet compared to typical genome datasets, which means it is not repetitive enough and the clustering would not give a decrease of the sum of edge weight as significant as the genome dataset.

Another measure was introduced to represent the cache efficiency of BWT queries, which is the jump distance of the inter-cluster LF-mapping steps (i.e., the difference between the new position and the old position on the BWM of each LF step) to further verify the improvement of cache efficiency. The reason was that the weights of the edges represent how many times the LF steps jump outside of the current cluster and cause cache misses; the jump distances represent how far the steps are going to jump because longer jumps indicate cooperation with strategies such as read-ahead is less effective, which means the cost of cache misses can not be hidden. Therefore, both the number of jumps that would cause cache misses and their distances can be used to model the anticipated improvement of cache efficiency. For the Hamlet dataset, the sum of the jump distances of the LF steps before METIS clustering was 8,494,543,292, and after the METIS clustering, it was 4,271,734,940, which is about 1.99 times better. Therefore, based on the results from the Hamlet dataset, it can be concluded that block partitioning and clustering indeed help to reduce the sum of edge weights and the sum of jumping distances and imply a layout of the BWT that provides more access locality with fewer cache misses.

Since the current clustering strategy directly clustered the blocks into a certain number of groups, which raised a hypothesis that a better clustering strategy may offer more significant decreases in inter-cluster edge weight. Hence, the clustering approach was switched to hierarchical clustering. For Hamlet, it can be hierarchically clustered 7 times from 813 clusters (one cluster per block) to 10 clusters as shown in Figure 4.2. Specifically, the new graph of each round is built from the clusters of the previous round, where the clusters are treated as new vertices and the sum of edges between clusters are treated as new edges. The total weight of inter-cluster edges decreases to 58,866 after hierarchical clustering with a final cluster number of 10. Meanwhile, the total weight of inter-cluster edges decreased to 72,127 using direct clustering with 10 clusters. Hierarchical clustering provides better results in terms of the sum of the weights of inter-cluster edges by 22.5%. In order to double-check the effectiveness of

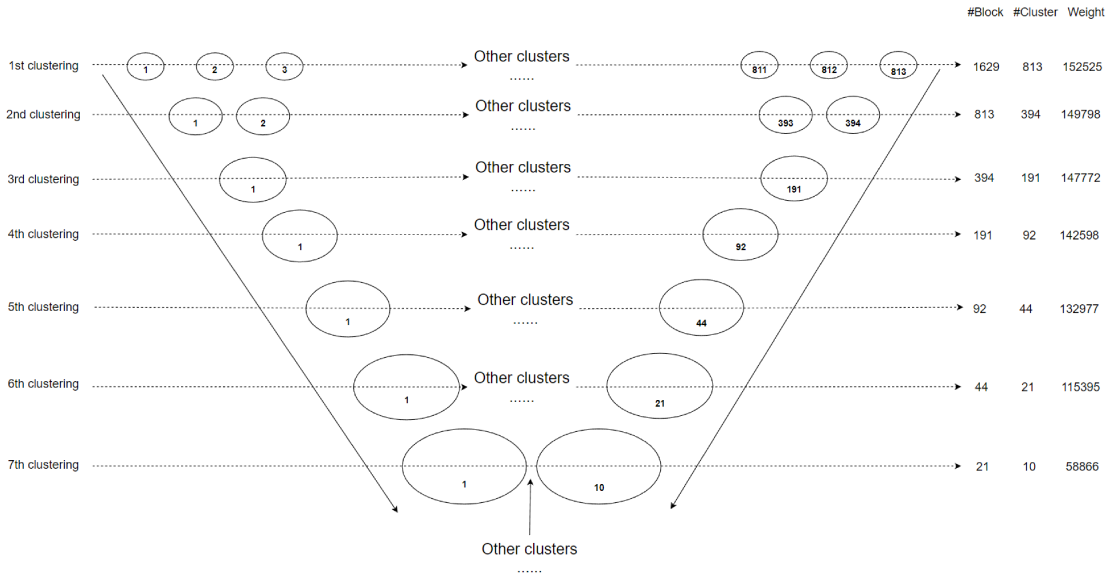

Figure 4.2: The hierarchical clustering of Hamlet

hierarchical clustering, another version of Hamlet (i.e., with a much bigger alphabet) was introduced with 228,320 characters [34]. The long version of Hamlet showed the opposite results, where the hierarchical clustering decreased the sum of edge weight to 94,554 with 6 clusters by clustering it 8 times; whereas the weight dropped to 74,996 using direct clustering with 6 clusters. As a result, hierarchical clustering does not necessarily give a better total edge weight compared to direct clustering. Also, when the dataset gets larger, such as the Human Reference Genome (HRG), then the hierarchical clustering procedure becomes much more costly, as the graph needs to be rebuilt for each round of clustering. For instance, the size of the HRG is 3,339,661,374 (after removing the headers of the reads). If the block size is set to 10,000 bases, we obtain 333,967 blocks and 166,984 clusters in the first round, then it would take 17 or 18 rounds of clustering to find the final result. As a result, the hierarchical clustering was discarded because of its unstable performance of reducing the total edge weight and high cost.

Since the block orders within clusters are still sequential and the current cluster size is relatively big, further permutations of the blocks within clusters could potentially form a layout that offers better access locality.

4.1.2 Further Optimization of the Block Orders Inside Clusters

The overall order of the blocks was permuted by clustering the blocks, but the blocks inside each cluster and the order of clusters are still sequentially stored. The blocks inside each cluster and the clusters themselves can also be further permuted to gain a more cache-efficient BWT layout. However, since the size of each cluster is relatively big compared to the total length of the dataset (e.g., 1/10 of the size of the dataset), the expected improvement of cache efficiency from rearranging clusters is limited due to the distance between the majority of clusters remaining the same. On the other hand, big cluster sizes mean that a cluster is unlikely to fit into the cache; hence, some relatively long jumps within the cluster would still cause cache misses. Therefore, the focus is on optimizing the order of blocks within a cluster to see if they give better cache efficiency.

Random permutation of the blocks within each cluster was the first approach applied because it can generate a new block layout inside each cluster rapidly and easily. Random permutation means generating some random numbers according to the block size and permuting the blocks in each cluster by either the same permutation or different permutations. Moreover, the predictor of cache efficiency was replaced by the total jumping distance. Fifty different permutations were generated using C++ built-in functions, and the performance of different block permutations was measured by their sum of the jump distances of LF mapping. The original sum of jump distances without any permutation of the long version of Hamlet with 228,320 characters was 417,406,828 after filtering out the inter-cluster jumps. When using the same random permutation on all the clusters, the best result out of 50 random permutations was 417,327,446, which is just an improvement of 0.27%. To explore a better performance using random permutation, the best-performing permutation was also found for each individual cluster, and the sum of jump distances was calculated using these best-performing permutations on the clusters. However, the sum of the jump distances still only decreases by a very small amount. Therefore, it was concluded that random permutation cannot significantly decrease the inter-cluster edge weights further compared to directed clustering.

Due to the necessity of block permutation inside clusters and random permutation did not offer satisfactory results, a more effective solution was needed to address

the problem of permuting the blocks within clusters. Since our goal is to minimize the total distance and total edge weight of LF steps, an ILP system with the objective function of minimizing the product of edge weights and edge distances was constructed.

$$\begin{aligned}
&\text{Minimize} \sum_{(u,v)\in E} w_{u,v} d_{u,v} \\
\text{s.t.} \quad & d_{u,v} \geq p_u - p_v \quad \forall (u,v) \in E \\
& d_{u,v} \geq p_v - p_u \quad \forall (u,v) \in E \\
& p_v = \sum_{i=1}^{n} i x_{v,i} \quad \forall v \in V \\
& \sum_{i=1}^{n} x_{v,i} = 1 \quad \forall v \in V \\
& \sum_{v\in V} x_{v,i} = 1 \quad \forall 1 \leq i \leq n \\
& x_{v,i} \in \{0,1\} \quad \forall (v,i) \in V
\end{aligned}$$

For the objective function, w is the weight of an edge, d is the distance of an edge, and u, v is a random edge between two blocks (i.e., vertices). For the constraints, the first two calculate the distances between endpoints. The third one translates the permutation points to vertices in the graph; the fourth and fifth ones limit the vertices so they can only appear once in the permutation. Also, it is necessary to ensure that x can only take integer values of either 0 or 1 since x represents the appearance of a vertex, so the binary variable $x_{v,i}$ is defined as 1 if vertex v appears at position i of the permutation and 0 otherwise.

This ILP was too complex to solve by hand, particularly when dealing with a large genome dataset. Hence, a software called Gurobi was applied to help solve the ILP [13]. A license was obtained for one year and the Gurobi optimizer was installed on the school server, Waverley. After proper installation and testing, the ILP was implemented in Python, we then conducted multiple weight-distance optimization experiments, with different parameters, to determine the best BWT layout. The dataset was changed to the HRG (GRCh38 version) [27] because Hamlet was not repetitive enough, not long enough, and had a rather large alphabet compared to genome datasets, and genomes of different species, especially the human genome,

which are the targeted dataset for this research. However, only snippets with different lengths were used with Gurobi optimization because the number of variables in the optimization matrix of Gurobi is the square of the number of blocks and too many variables in the matrix slow down the optimization process dramatically.

The optimization strategy used by Gruobi explores the upper bounds and lower bounds of the different solutions to narrow down the gap between the lower bound and upper bound; when the lower bound meets the upper bound, the optimal solution of the permutation of blocks is found. The first experiment was conducted with a portion of the HRG with 248,000 bases, a block size of 100 bases and an average cluster size of 99 blocks, resulting in 25 clusters in total and an optimization matrix of 9801 variables for each cluster, which was a challenge even for Gurobi. After running the optimizer for over 48 hours on the block permutation of size 99, the gap still remained over 90%. We decreased the cluster size to only 20 blocks, which generated a matrix with only 400 variables. The optimizer was able to solve the ILP with 20 blocks in 90 minutes, resulting in a decrease of $weight * distance$ from 180288 to 53316 for that particular cluster. This indicated that Gurobi provides a better block permutation of the cluster by decreasing the product of the weights and distances of the edges by a factor of 3.4. If all clusters were optimized in this manner, it should be possible to form a more cache-efficient layout of the BWT that can inverse the BWT in a much shorter time compared to the original layout within clusters. However, to obtain better performance of the inversions and meaningful clustering, block sizes are typically small compared to the entire length of the BWT. Therefore, it would result in a large number of clusters if the cluster size is limited to as small a number of blocks as 20, and optimizing all of the clusters using Gurobi would be costly in terms of time. Thus, reducing the running time of the optimization procedure with a reasonable cluster size is necessary. Manipulating different parameters was attempted, but only two of them actually helped to reduce the running time of the optimization, which were *MIPFocus* and *NoRelHeurTime*.

According to Gurobi's definition, the *MIPFocus* parameter "allows modification of the high-level solution strategy, depending on the user's goals" [13]. By default, the Gurobi optimizer focuses on "finding new feasible solutions" while "proving the current solution is optimal" [13]. However, users can customize the solver's behaviour

by setting the *MIPFocus* parameter [13]. If $MIPFocus = 1$, the optimizer would focus on finding new feasible solutions. Conversely, setting $MIPFocus = 2$ causes the optimizer to prioritize providing the optimal solution to the problem [13]. In the third case, if the best objective bounds of the ILP still have a large gap after running for a long time, setting $MIPFocus = 3$ makes the optimizer prioritize reducing the gap between the best objective bounds [13]. As this third scenario addresses the issues encountered in this study, *MIPFocus* was set to 3, and the optimization of the order of 20 blocks was re-conducted. This time, the optimizer finished in under 50 minutes, yielding the same block permutation result, which suggests that using *MIPFocus* resulted in a meaningful improvement. Hence, $MIPFocus = 3$ was added to the code of the ILP optimizer.

NoRelHeurTime "limits the amount of time (in seconds) spent in the NoRel heuristic", where NoRel stands for No Relaxation, and it "searches for high-quality feasible solutions before solving the root relaxation" [13]. By setting *NoRelHeurTime*, the optimizer prioritizes finding the optimal objective upper bounds first. For a cluster size of 20 blocks, the optimizer can find the optimal objective upper bound in under 5 seconds. It has also come to our attention that the block permutation results were the same as those obtained after running the optimizer for 50 minutes if the optimization is stopped after 5 seconds. This suggested that finding the optimal objective lower bounds can be deprioritized for this particular ILP problem. Therefore, the focus was shifted to optimizing the upper bounds for larger cluster sizes. However, it still took around 7,500 seconds to obtain the optimal objective upper bound for a cluster size of 100 blocks. After examining the changing trends of the objective upper bounds over time, it was observed that the upper bounds decrease dramatically at the beginning of the optimization and tend to drop very slowly after a few minutes. As a result, the decision was made to find the best block permutation with a time constraint, instead of attempting to find the ultimate optimal permutation, which would take too long. By setting the $TimeLimit$ parameter to 200 seconds, the optimization was attempted on longer portions of the HRG with 1,000,000 bases and 10,000,000 bases, with different combinations of block and cluster sizes. With a block size of 500 and a cluster size of 100 blocks, the best result obtained was a decrease of the sum of $W * D$ by a factor of 3 for each cluster for the 1,000,000 base portions.

Since the goal of optimizing the block permutation has shifted from finding the optimal solution to finding an acceptable solution, another optimization technique that finds the global optimum was also applied, which was the SA algorithm. It is a probabilistic method for finding the approximate global optimum of a given function [25]. An SA algorithm was developed for optimizing the block permutation of each cluster using Python. Following the general guideline of applying the SA algorithm, the code consists of three aspects: the initialization function, the objective function, and the optimization function. The initialization function creates a list of integers from 0 to the size of the cluster, which serves as a starting point for the optimization. The objective function calculates the cost by iterating over the list of edges and adding the product of the weight and distance of that edge, returning the cost. The optimization function takes the objective function, the starting points, the number of iterations, the temperature, and two other lists that contain the edges and their indices as inputs. It performs the swap operation on a random pair of edges in the current permutation and calculates the new cost. If the new cost is better than the current cost, it updates the current permutation and cost; otherwise, it does not. Meanwhile, it uses a probability function to decide whether to accept the new permutation or not. Finally, it returns the best permutation and cost found during the optimization process and the list of costs at each iteration. After a few attempts using different temperatures and numbers of iterations, the best result achieved in a reasonable time frame was 20,000,000 iterations. However, the product of weight and distance of the edges in the cluster only dropped from 176108 to 162746 for that particular cluster, which is only 8%. This reduction was insignificant compared to the result obtained using the Gurobi optimization. Therefore, the Gurobi optimizer was chosen as the final block permutation optimizing strategy.

The blocks have been clustered and the order of blocks inside each cluster has been optimized, the next step was to build a data structure to verify whether the optimized layout of BWT actually provides a better cache efficiency compared to the traditional layout.

4.1.3 Building the Data Structure

We have confirmed that the weights of inter-cluster edges and jump distances of the LF steps can be decreased by rearranging the layout of BWT using clustering and Gurobi optimizer, but they are just proxy measures for estimating the final cache efficiency of the permuted BWT layout. In order to observe the real BWT inversion performance after replacing the regular BWT layout with the optimized layout, a data structure was built to test the speed of backward searches. The data structure was built to support *access* and *rank* queries on any random string but mainly focuses on the BWT of genomes. To compare the data structure, a wavelet tree of the BWT of the input dataset was also built using SDSL, which also supports *access* and *rank* queries of the BWT.

The data structure used a lookup table to conduct *access* and *rank* queries of the BWT. The table was created where each entry of it corresponds to one BWT block. For each BWT block, there is a rank header that identifies the rank of the first index of this block concerning each character in the alphabet. All the BWT blocks were permuted according to the optimized BWT layout and stored in the table. In addition, the substring inside each BWT block was represented as a wavelet tree, in order to support *access* and *rank* queries space-efficiently. With the help of the table and the wavelet tree of the BWT blocks, *access* and *rank* queries on the BWT of the dataset can now be supported.

For *access* queries, only one parameter is required, which is the position I. Using I with the block table T, block size B, and the block permutation P, the ith character of the BWT can be obtained using the following formula: $T[P[I/B]].block[I\%B]$, where *block* is the BWT block entry of the table.

For *rank* queries, two parameters are required: the querying character C and the querying position I. First, the BWT block in which C is located can be found by $C_B = I/B$, and the offset of C inside that BWT block is found by $C_O = I\%B$. If C is located in the first block ($C_B = 0$), then the rank of C at I can be found by $T[P[C_B]].block.rank(C_O, C)$, where *rank* is calling the rank query of the wavelet tree. If C is located in the another block ($C_B \neq 0$), then the rank of C at I can be found by $T[P[C_B]].headers[A[C]] + T[P[C_B]].block.rank(C_O, C)$, where A is the alphabet of the BWT and $headers$ is the rank headers of the current block.

After running some test cases, it was discovered that the running time of building the data structure was much longer than expected. We determined the part of the code that has been slowing down the process is the building of the block wavelet trees because it was in integer sequence wavelet trees, which are just numbers separated by white spaces that are intended to serve an educational purpose [10]. To improve the performance, the integer sequence wavelet trees were replaced with byte sequence wavelet trees because they are more space-efficient. Specifically, the byte sequence wavelet trees use a sequence of 1-byte unsigned integers (i.e.,uint8_t), which means they use 1 byte for each character in the string; whereas the integer sequence wavelet trees use decimal numbers, which means they use 4 bytes per character in the string. Moreover, balanced wavelet trees were transformed into Huffman-shaped trees to reduce the size of the bit vectors because the total length of the bit vectors of the balanced wavelet trees is between $|X|\lfloor \log |\Sigma| \rfloor$ and $|X|\lceil \log |\Sigma| \rceil$, but Huffman-shaped wavelet trees use exactly $|X|$, where X is the string and Σ is the alphabet [11]. After these modifications, the building time of the table was reduced. The next step is to use this data structure to perform a BWT reverse search and compare its performance with a single wavelet tree constructed from a traditional BWT layout to determine whether the optimized layout has better cache efficiency compared to the traditional layout.

4.1.4 The Final BWT Inversion

The purpose of the BWT inversions is to verify whether the two steps of block order optimizations were actually decreasing the time of backward searches of the BWT and improving the cache or memory efficiency of the BWT. The inversions were first performed on the Waverley server. In order to verify the effectiveness of the block order optimization within clustering using Gurobi, the inversion tests were first conducted with a portion of 1,000,000 bases of the HRG, comparing the inversion time with the BWT layout without Gurobi optimization. The results revealed that the Gurobi optimizations on the block order within clusters were ineffective because the inversion time remained the same; therefore, the block order optimization within clusters was discarded. Next, the dataset expanded to the entire HRG since it was no longer limited by the Gurobi optimizer and the inversion was conducted using the

layout with block partitioning and clustering optimization and the traditional layout. The inversion results showed that the optimized layout was even slower than the traditional layout. Such an unexpected result expressed that the optimized layout did not offer better access locality compared to the traditional layout. However, it could also be because the Waverley server has more than enough memory to store the entire single wavelet tree of the BWT of HRG, which means the inversion using the traditional layout would not cause more disk access and lead to a longer inversion time. Meanwhile, the data structure that was built to invert the BWT was not as optimized as the BWT inversion algorithm from the SDSL. In order to address the problem of sufficient memory will not cause more disk access, the genome of Douglas Fir of 14.62 GB was applied as the input dataset and virtual machines were built on a laptop with limited memory from 8 GB to 1 GB. The Douglas Fire was chosen because the predicted size of its single wavelet tree is about 4 GB based on the types of bases of 4. Meanwhile, the backward searches were conducted on a virtual machine rather than the university's server because we have trouble limiting the memory on the server, which could be because we did not have permission. The inversions using an optimized layout started to outrun the inversions with the traditional layout when the memory size is less than 3 GB. Hence, the optimized layout offered more cache efficiency but only when the memory size is less than the size of the wavelet tree of the BWT.

Human Reference Genome Inversion

The inversion performance test was first conducted on a portion of the HRG of 1,000,000 bases, due to the limited number of variables in the optimization matrix that Gurobi could handle. The BWT layout of the portion was optimized, which includes block partitioning, clustering using METIS, and optimizing the block orders within clusters using Gurobi.

According to the previous result, the Gurobi optimizer reduced the product of distances and weights of the edges by a factor of 3 for the 1,000,000 bases portion with a block size of 500 and a cluster size of 100 blocks, so the expected inversion time with the Gurobi optimized was supposed to be faster compared to the inversion using the layout after clustering and without block order optimization inside the clusters.

However, the inversion running time remained the same. To ensure the reliability of this observation, different block sizes and cluster sizes of the portion were applied with the Gurobi optimization, but the inversion times did not show any noticeable decrease. Therefore, it was concluded that although the Gurobi optimizer decreases the $weight * distance$ of edges by a factor of 3, it does not give better performance in terms of inversion speed. Hence, Gurobi optimization was not included in future BWT inversion experiments, and only block partitioning and METIS clustering remained as approaches to optimize the order of blocks. However, the size of the input dataset was no longer limited by the number of variables in the optimization matrix of the ILP, and the following inversion tests were conducted on the entire HRG.

The inversion of the BWT of the entire HRG was performed using two different approaches in order to verify if the optimized layout is more cache or memory efficient than the traditional layout: a single Huffman-shaped wavelet tree with Ramen Ramen Rao (RRR) compressing algorithm [30] to represent the traditional layout and the data structure that uses a table of block Huffman-shaped wavelet trees with RRR compressing algorithm with optimized layout. The purpose of compressing both types of wavelet trees with the RRR algorithm was to reduce the space occupied by the wavelet trees. Both approaches were able to correctly invert the BWT to the original HRG sequence, but the inversion time using the data structure was slower than using the single wavelet tree, which indicates that the inversion did not take advantage of the permuted BWT layout, at least not with the current setup of sufficient memory.

We conjectured that the Waverley server, which has a total memory of 94.132 GB, could potentially store the single wavelet tree of the BWT of the HRG entirely in memory because the size of the single wavelet tree was only 731 MB. Therefore, a larger dataset with a limited amount of memory was required in order to guarantee that the whole single wavelet tree must be stored on disk, leading to significantly more disk accesses and slowing down the inversion if cache locality is limited. In contrast, the data structure only has block wavelet trees, which means that only some of the block wavelet trees would be stored in memory at once, and they were permuted based on METIS clustering, potentially resulting in fewer disk accesses and achieving a better inversion speed because the LF steps tend to keep within the cluster as often as possible.

Douglas Fir Genome Inversion

In order to increase the size of the single wavelet tree that was built from the traditional layout of BWT so that the single wavelet tree can exceed the size of the available memory, the Douglas Fir genome with 15,703,424,632 bases is chosen as the new dataset, which requires 14.62 GB of storage [28]. The genome was chosen based on its length and not relevant to its species, a single wavelet tree with a predicted length of 4 GB was expected to build from the genome based on its alphabet size. In practice, the Huffman-shaped RRR-compressed single wavelet tree of the BWT of Douglas Fir is serialized using SDSL, and its size is 2.67 GB, which could be because of the Huffman code compression and RRR compression. However, other system processes also require memory while performing BWT inversion. Thus, it is anticipated that BWT inversion would experience increased disk access when conducted with 3 GB of memory or less since the whole wavelet tree cannot fit into memory. The data structure that employs block wavelet trees would demonstrate a speedup (i.e., improvement of inversion time) in comparison to the single wavelet tree due to less disk access. In terms of limiting the memory that can be allocated by the inversion process, the first approach was using commands to limit the amount of memory on the Waverley server. However, the command did not work well, so the second approach was to build virtual machines with limited memory.

The first attempt to limit the memory usage of the inversion process on the Waverley server was to use the *ulimit* command, which provides resource control for particular processes on systems that allow it [22]; therefore, we were expecting the *ulimit* command could limit the amount of memory that can be accessed by the inversion process. The manual page of *ulimit* indicates that $-Sv$ provides a soft limit of "the maximum amount of virtual memory available to the process," while $-Hv$ provides a hard limit, where "the soft limit is the value that the kernel enforces for the corresponding resource and the hard limit acts as a ceiling for the soft limit" [22]. Initially, the memory limit was set to 10 GB using $-Sv10000000$, which was smaller than the BWT of the reference genome of Douglas Fir but larger than the single wavelet tree that was built from it. However, the inversion time using the single wavelet tree remained the same, indicating that the inversion did not cause more disk accesses than before. Then the memory limit was changed to a hard limit of 10 GB

using $-Hv10000000$, but the inversion time remained the same. Next, the memory usage was reduced to 3 GB, which was supposed to be smaller than the total size of the single wavelet tree combined with other essential processes that support the system. However, the inversion program threw a "bad_alloc" instance and aborted for both a soft limit and a hard limit. Such a scenario that cannot limit the amount of memory used by a process could be because the Waverley server has enough memory to buffer the inversion process by swapping out the current memory block that has been occupied by the single wavelet tree during the inversion, even if the process itself does not have control of that piece of memory. Furthermore, we did not have permission to stop the server from swapping out the memory block. Therefore, the *ulimit* command could not be used to limit the memory usage of the inversion program and virtual machines built on a laptop were applied.

As the inversion experiments cannot be effectively conducted on the Waverley server with limited memory, an alternative solution was proposed to run the experiments on a virtual machine where the memory size can be limited during the installation and configuration of the OS. A virtual machine with Ubuntu 22.04.1 LTS and kernel 5.15.0-58-generic was created on VMware17 on a Lenovo Legion 5 with AMD Ryzen 7 5800H at 3.2 GHz, 32 MiB L3 cache, 1 TB Solid-state drive (SSD) SKHynix-HF001, 32 GiB of DDR4 main memory and an 8 GB swap file. The memory size of the virtual machine was varied from 8 GB to 1 GB with a decrement of 1 GB to confirm the hypothesis that the single wavelet tree of the BWT would cause more disk accesses when the memory size is 3 GB or smaller. The substring extraction method was used to replace the BWT inversion because the accuracy of backward searches was confirmed and inverting entire genomes was time-consuming. The length of the substring was set to 100,000 bases based on the virtual machine's performance (i.e., the substring extraction can be finished within 10 seconds). The substring extractions were conducted on the single wavelet tree that uses the traditional layout of BWT and block wavelet trees that use the optimized layout of the BWT with a block size of 1,500,000 bases, and a cluster number of 10, where the block and cluster size were preliminary guess of optimal sizes based on the size of the dataset.

As shown in Figure 4.3 , the extraction time using the single wavelet tree significantly increases from 4 GB to 3 GB of memory, which is consistent with the earlier

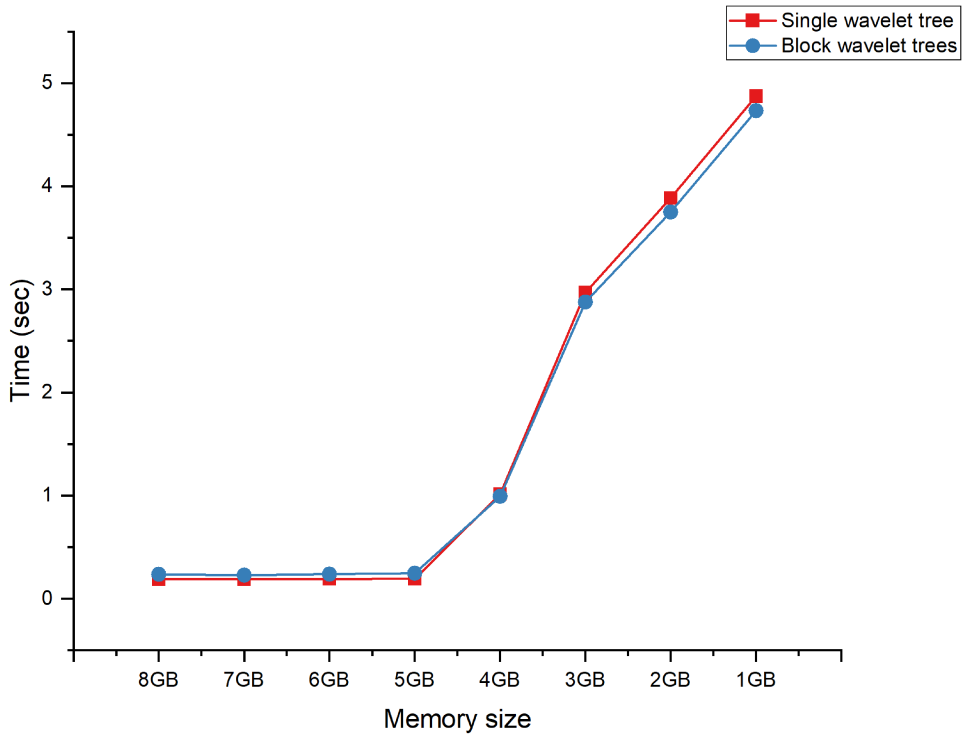

Figure 4.3: Substring extraction using the single wavelet tree and block wavelet trees with block size 1,500,000 bases.

proposed hypothesis that 3 GB of memory would not be sufficient to store the entire single wavelet tree after subtracting the memory used by system processes. Therefore, when the memory size is 3 GB or less, the system needs to access the disk more frequently to complete the extraction. The substring extraction time using the block wavelet trees shows a similar trend to the single wavelet tree and took slightly longer times to finish the substring extraction when the memory is between 5 GB and 8 GB. When the memory is equal to or less than 4 GB, the block wavelet trees exhibit a small advantage over the single wavelet tree, and the gap between these two types of wavelet trees increases as the memory size decreases. However, the speedup is only about 5% when the memory shrinks to 1 GB, which is the minimum memory requirement of the OS. On one hand, the substring extraction experiment showed that the block wavelet trees provided a speedup over the single wavelet tree when the available memory is smaller than the size of the single wavelet tree, which means the optimized layout is more memory-efficient than the traditional layout with certain setups. On the other hand, the speedup is rather limited, a 5% improvement is not

worth the effort to optimize the BWT layout. However, the applied block and cluster sizes were just randomly chosen. More different block and cluster sizes needed to be tested in order to find the optimal combination of block and cluster size that offers the most significant speedup. The results of testing different block and cluster sizes will be presented in the next section.

4.2 BWT Inversion Results

To obtain the most significant improvements in terms of substring extraction speeds, the virtual machine's memory size has been set to 1 GB for all subsequent substring extractions because that is the minimum memory requirement of the OS and the speedup grows as memory size drops. More block and cluster sizes were explored to determine if they provide better performance (i.e., bigger speedup). Additionally, a switch from a solid-state drive (SSD) to a hard disk drive (HDD) was applied for the purpose of expanding the difference between the extraction time using the single wavelet tree and the block wavelet tree. This is because block wavelet trees produce fewer disk accesses, and each disk access using HDD would take a longer time than using SSD. Furthermore, since a disadvantage of generating block wavelet trees using SDSL was noticed, an alternative approach for generating block wavelet trees has been proposed, which is a program implemented by the author of SDSL Gog et al. [11].

4.2.1 BWT Substring Extraction with SSD

In pursuit of optimizing block and cluster sizes for bigger speedup in substring extraction, the range of block sizes was extended from 1500 bases to 150,000,000 bases, while cluster sizes ranged from 14.98 MB to 1.46 GB when allowed by the current block size. The experiment began with smaller block sizes, but it was found that wavelet trees with block sizes of 1500, 15,000, and 150,000 bases were impractical because they demonstrate slower extraction speeds compared to the single wavelet tree. This is likely because the total size of the block wavelet trees exceeds the size of the single wavelet tree when block sizes are that small. Specifically, the total size of the block wavelet tree with 1500 bases is 31 GB, and it decreases to 2.9 GB when the block size is 150,000 bases, while the size of the single wavelet tree is 2.67 GB. The underlying reason is that each wavelet tree included its own universal table, which is

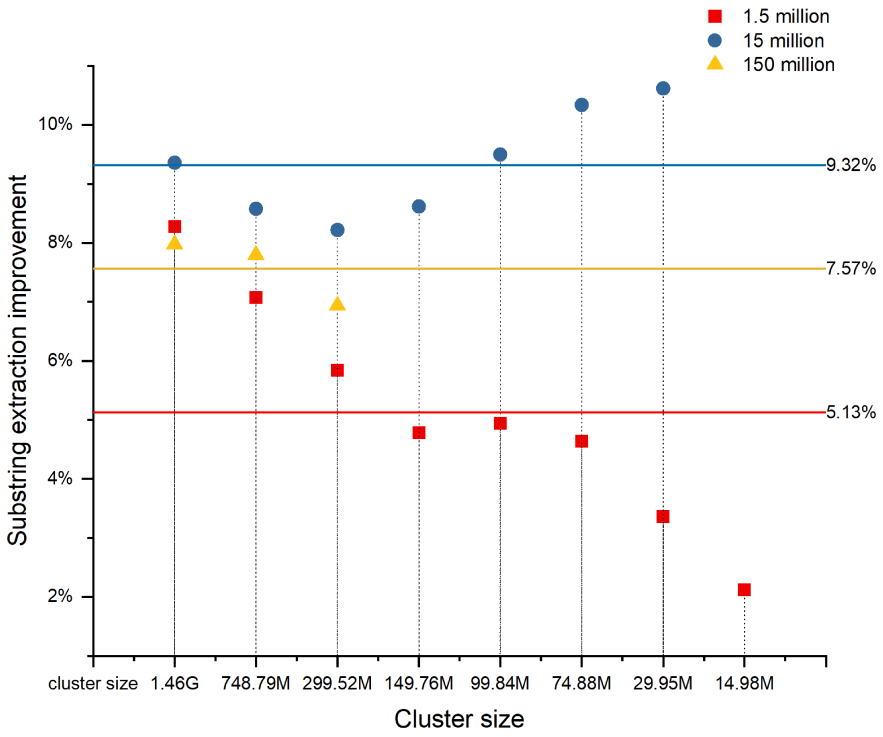

Figure 4.4: Substring extraction using block wavelet trees of size 1.5 million to 150 million with different cluster sizes (the x-axis is reversed because it follows the number of clusters from small to big).

a two-dimensional array that stores information about the frequency of occurrence of each symbol in each prefix of the input [12], so the size of the universal table depends on the size of the alphabet. Therefore the universal table of the block wavelet tree with 1500 bases has the same size as the universal table of the single wavelet tree and more block wavelet trees means more space occupied by the universal tables. For instance, the BWT of Douglas Fir was divided into 10,468,950 blocks when the block size is 15000 bases, and 10,468,950 universal tables would take a lot of space. This clearly indicates that using small block sizes will not offer any improvement but drawbacks compared to the single wavelet tree.

Therefore, the next phase of substring extraction was carried out using block sizes of 1,500,000 bases, 15,000,000 bases, and 150,000,000 bases with the cluster sizes mentioned previously. The results of such substring extraction are shown in Figure 4.4, it can be observed that block wavelet trees with a block size of 1,500,000 bases exhibit a significant decreasing trend as the cluster size decreases, from over 9% to

below 2%. Conversely, wavelet trees with a block size of 15,000,000 bases only exhibit a decline to the cluster size of 299.52 MB, after which it demonstrates a significant increase, contrary to the trend observed with 1,500,000 bases. For block wavelet trees with a block size of 150,000,000 bases, only a few clusters could be formed, and the only three scatter points obtained show a declining trend, consistent with the 1,500,000 bases wavelet tree. Therefore, there was no clear pattern shown regarding the speedup of using the optimized layout and the cluster size.

Nonetheless, the 15 million bases block wavelet tree demonstrates a most significant average speedup of 9.32% among three block sizes, regardless of cluster size. Hence, it can be concluded that progress has been made in finding the optimal block size, which is around 15 million bases. Additionally, the 1.5 million bases block wavelet trees outperform the 150 million bases when the cluster is 1.46 GB. Since larger block sizes like 150 million did not provide many options for cluster sizes, which means the majority of the layout of the BWT was not permuted. There, the subsequent substring extractions were conducted with block sizes ranging from 1.5 million to 15 million bases.

Since the focus was on block sizes between 1.5 million and 15 million, new test points were selected with block sizes of 5 million and 10 million in order to cover the block sizes that have not been tested, while keeping the cluster sizes the same. As seen from the Figure 4.5, both the 5 million and 10 million block sizes offer better speedups compared to the previously investigated block sizes, with an average improvement of 13.55% and 10.39%, respectively. However, the improvements still do not demonstrate a clear pattern of changes in terms of cluster sizes, which leads to some concerns about whether the cluster sizes have a regular impact on the improvements. As both 5 million bases and 10 million bases block size wavelet trees gave better speedup compared to the block sizes that have been tested, further substring extractions were conducted on block sizes between 10 million bases and 5 million bases.

Despite the improvement in substring extractions using block wavelet trees with optimized layout, 13.55% is still not a notably high percentage. However, it was realized that the Lenovo Legion 5 does not include an HDD, only an SSD for disk equipment. It was believed that the block wavelet trees would provide an even better improvement over the single wavelet tree if substring extraction is conducted on an

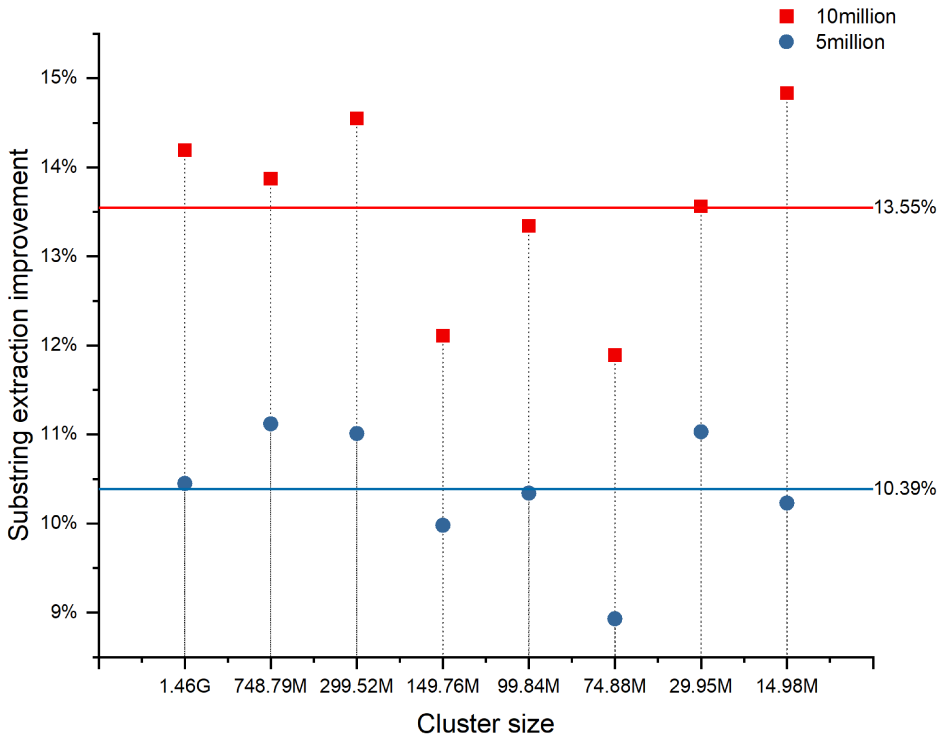

Figure 4.5: Substring extraction using block wavelet trees of size 5 million and 10 million with different cluster sizes.

HDD because each disk access takes a longer time on an HDD than an SSD; hence, the same number of disks accesses on HDD would accumulate a bigger gap between the inversion using block wavelet tree and the single wavelet tree compared to SSD. As a result, a laptop with an HDD was used as the host of the virtual machines and more substring extractions were performed on it.

4.2.2 BWT Substring Extraction with HDD

As the previous host of the virtual machines did not include any HDD and the disk access time of HDD is longer than SSD, the host of the virtual machine was switched to a laptop that has an HDD for the purpose of expanding the gap between the inversion times with the traditional layout and the optimized layout. The new virtual machine host was a Dell Inspiron 14-7472 with an Intel i7-8550U processor, consisting of 8 cores running at 1.80 GHz and 16 MiB L3 cache, 1 TB ST1000LM035-1RK172 disk, 8 GiB of DDR3 main memory and a swap file of 8 GB. The virtual machine still used the same operating system and memory size of 1GB. Additionally, the substring

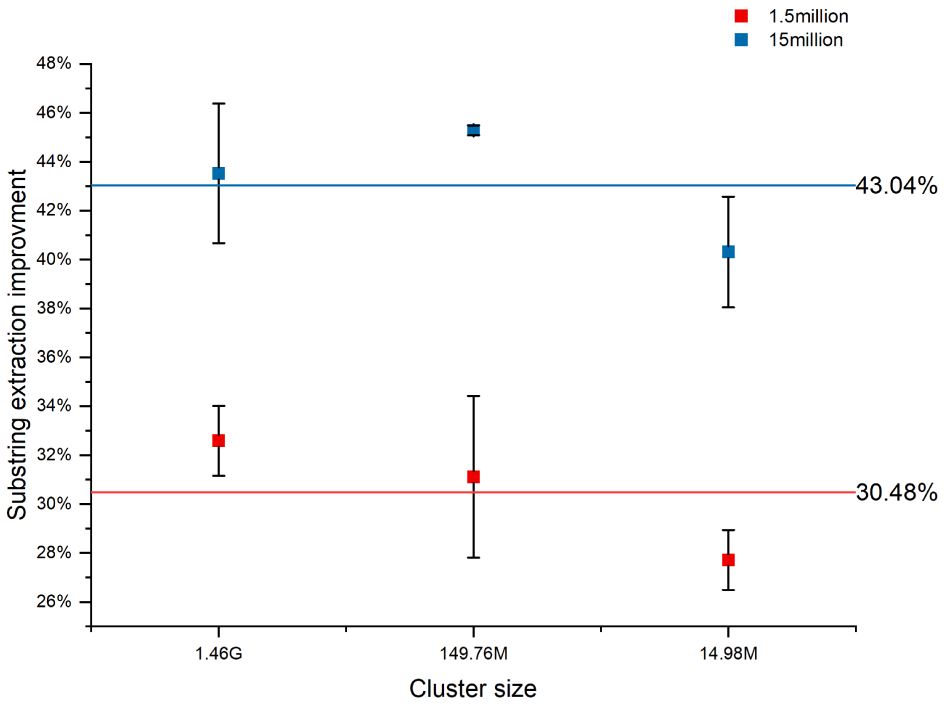

Figure 4.6: Substring extraction using block wavelet trees of size 1.5 million and 15 million with different cluster sizes on HDD.

size was reduced to 5000 bases due to the Dell laptop's less powerful performance. Since it was unclear whether the block size affects performance on the HDD laptop to the same extent as it does on the SSD laptop, substring extraction needed to be re-conducted on the tested block size. However, block sizes greater than 15 million were not prioritized due to their relatively large size compared to the Douglas Fir's size. Meanwhile, block sizes ranging from 1500 bases to 150000 bases were still not considered for HDD substring extractions as they contained too many universal tables, which leads to space explosion. The first two block sizes tested with the HDD were 1.5 million and 15 million, as they were the smallest and largest reasonable block sizes that have been investigated.

As shown in Figure 4.6, the results show that the HDD offers much better speedups compared to SSD. The use of the block wavelet tree now yields an improvement of around 30% to 40%, with a maximum of over 45% with the 15 million base block size. Such outstanding speedups confirmed the anticipation that an HDD takes significantly more time to accomplish one access compares to SSD, and with the number of disk

accesses caused by the single wavelet tree exceeding the block wavelet trees kept the same, the extraction gap grows in terms of time. In terms of the difference in speedups between the two block sizes, the 15 million bases block wavelet trees offer an average speedup of 43.03% and the 1.5 million bases block wavelet trees offer an average speedup of 30.48%, which confirmed that the optimal block size is around 15 million bases. The graph also includes the standard deviations of the average improvement of the extraction, which show that the speedups are relatively stable at 40% and 30% for the 15 million block size and 1.5 million block size, respectively. However, there is still no clear pattern of how the cluster sizes affect substring extraction improvement. The 1.5 million block size shows a decreasing trend, while the 15 million block size has a peak point in the middle cluster size. Nevertheless, more block sizes were tested to find a more significant substring extraction speedup from applying the optimized BWT layout.

Since no clear pattern has emerged in finding a better cluster size for substring extraction, whether the METIS clustering helped with the speedups or the improvement was solely due to block partitioning the BWT, to address this question, it was decided to compare the trends of speedup with different block sizes, but the same set of cluster sizes, to determine if the speedups demonstrated a pattern with respect to cluster sizes. Table 4.1 lists the new block sizes that were tested, covering the majority of block sizes between 1.5 million and 15 million in increments of 1 million. If no similar pattern of improvement changes were observed among the cluster sizes in at least half of the block sizes tested, there would be concerns regarding the effectiveness of the METIS clustering.

Block size	2000000	4000000	5000000	6000000
Block number	7852	3926	3141	2618
7000000	8000000	9000000	10000000	11000000
2244	1963	1745	1571	1428

Table 4.1: New block sizes for the substring extractions

Figure 4.7 depicts all the substring extraction experiments that were conducted on block sizes between 1.5 million and 15 million. As observed from the plots, six out of the 11 plots show a single peak at either cluster size 149.70MB or 29.95MB, while the remaining five plots show the lowest data point at either cluster size 149.70MB

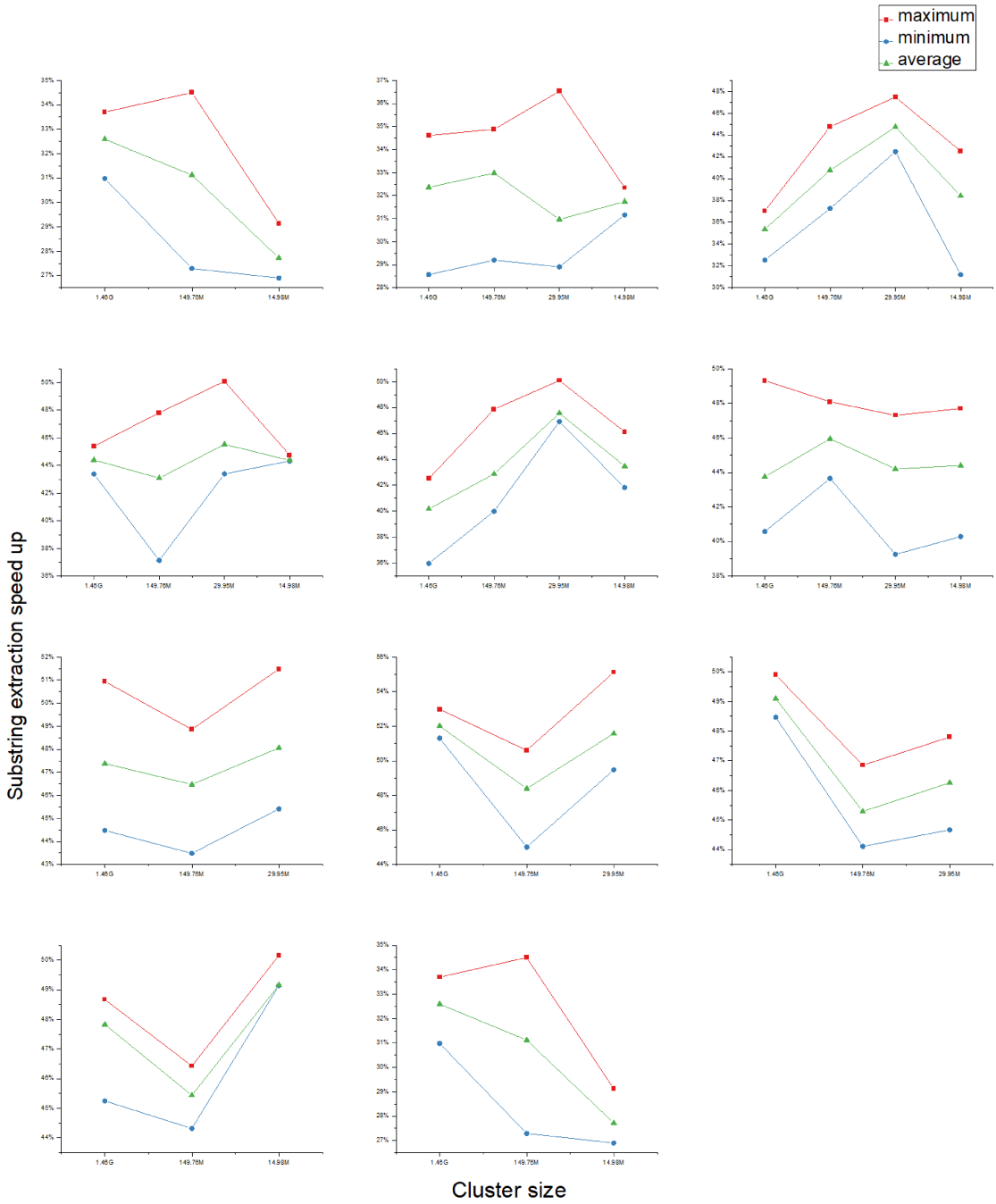

Figure 4.7: Substring extraction using more different block sizes wavelet trees between 1.5 million and 15 million. Notice that the rightmost 3 points in the last graph show speedups between 27% and 29% with essentially equal block and cluster sizes.

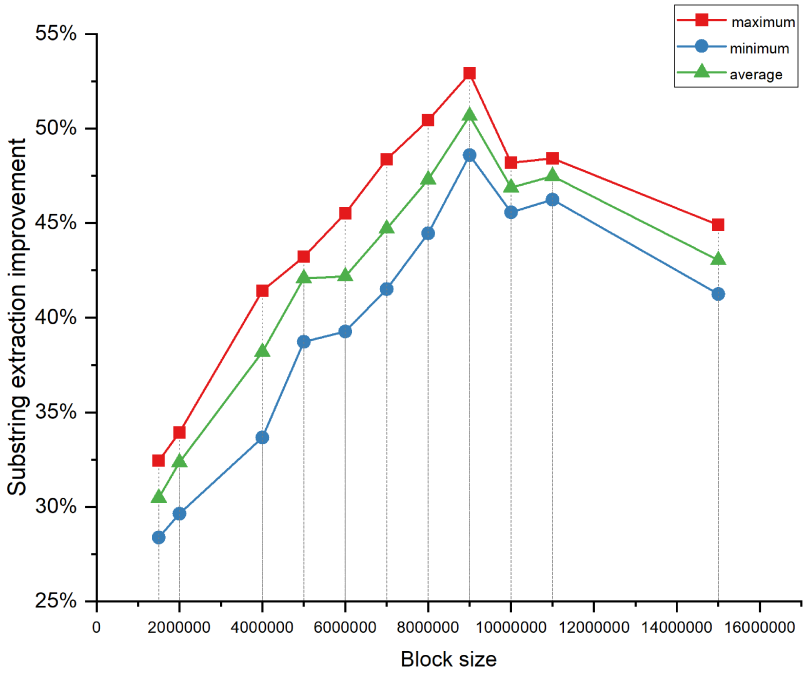

Figure 4.8: The overall performances of substring extraction between 1.5 million to 15 million block sizes.

or 29.95MB. None of the 11 plots exhibit a clear increase or decrease trend, and they are almost equally divided into two groups showing completely opposite trends. Consequently, no discernible pattern was observed among the different cluster sizes in terms of the speedup for substring extraction. More importantly, we observed that the block wavelet trees still demonstrated a 27% to 29% speedup with essentially equal block and cluster sizes of 15 million bases and 14.98 MB, which should be no speedup since no clustering exist. Such a scenario raised doubts about whether the METIS clustering contributes to the efficiency of substring extraction.

As the cluster sizes did not appear to make any difference and sufficient substring extractions have been conducted with different block sizes, it was decided to collect all the performances of these substring extractions and investigate how the block sizes affect them and whether there is a pattern of changes concerning the block sizes. As shown in Figure 4.8, the results indicate that the speedups of substring extractions using different block sizes demonstrate a clear trend along with the increase of block sizes until blocks of 9 million bases. Specifically, there is a significant speedup of 20%

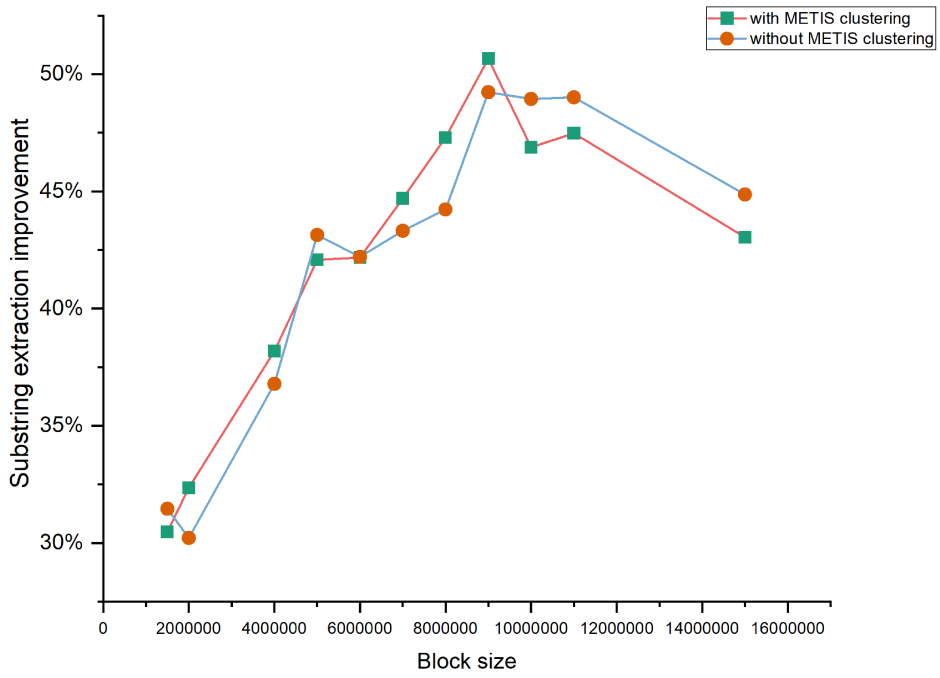

Figure 4.9: Comparing substring extraction with and without METIS clustering.

in performance from the block size of 1.5 million bases to 9 million bases, and the peak speedup of 50.66% is achieved at the block size of 9 million. After the peak, the speedups gradually decrease to 43.04% when blocks increase to 15 million bases. Therefore, the block size has a significant impact on the performance of substring extractions, and the optimal block size should fall within the range of 9 to 10 million bases, providing a 50% improvement in extraction speed compared to conducting substring extractions using the single wavelet tree.

To investigate whether the METIS cluster offers any help regarding substring extraction speed, more substring extractions were performed with only the block partitioning approach. The results of these substring extractions were then compared with the substring extractions that were conducted using the same set of block sizes but used METIS clustering. Figure 4.9, indicates that the METIS clustering does not offer any additional speedups for the substring extractions on top of the speedups gained from block partitioning the uncompressed BWT. This is in line with the previous conjecture drawn from Figure 4.7. Therefore, it appears that the substring

extractions gain a 40% to 50% speedup just by block partitioning the uncompressed BWT, which is not consistent with the previous assumption of how the layout of the BWT would affect the speed of backward stepping of the BWT, but it is still a noticeable increment of the backward search speed. Moreover, the current optimal block size was bigger than anticipated, and the non-sharing universal table issue of the block wavelet tree prevented the exploration of smaller block sizes. Therefore, an alternative approach was needed to produce the block wavelet trees in a different style so that the universal table is shared between them. Fixed block compression boosting (FBB), another approach that employs block partitioning to building wavelet trees that were implemented by Gog et al., was proposed to solve the issue of non-shareable universal tables.

4.2.3 BWT Substring Extraction Using Fixed Block Compression Boosting

Due to the renunciation of METIS clustering and the hypothesis that small block sizes would provide more significant improvements, FBB wavelet trees were applied to replace the block wavelet trees built from our data structure. The author of SDSL Gog et al. implemented the FBB technique, which is a simpler and faster alternative to optimal compression boosting and implicit compression boosting used in previous FM-indexes, such as the RRR-compression used for both the single wavelet tree and the block wavelet tree in the BWT inversion experiments [11]. More importantly, this technique is compatible with SDSL and solves the issue of non-sharing universal tables among a set of wavelet trees that use the same alphabet. Therefore, FBB is considered an innovative technique for generating block wavelet trees, especially for smaller block sizes wavelet trees that the previous substring extractions have not covered yet. Its performance was compared with the single wavelet tree that is Huffman-shaped and RRR compressed, the Huffman-shaped single wavelet tree but uncompressed, and with block wavelet trees that are Huffman-shaped and RRR compressed with the optimal block size of 9 million bases based on the previous experimental results. FBB wavelet trees were compared with these three types of wavelet trees because they were all employed during the inversion experiments that have been conducted. All substring extractions were performed on the HDD laptop

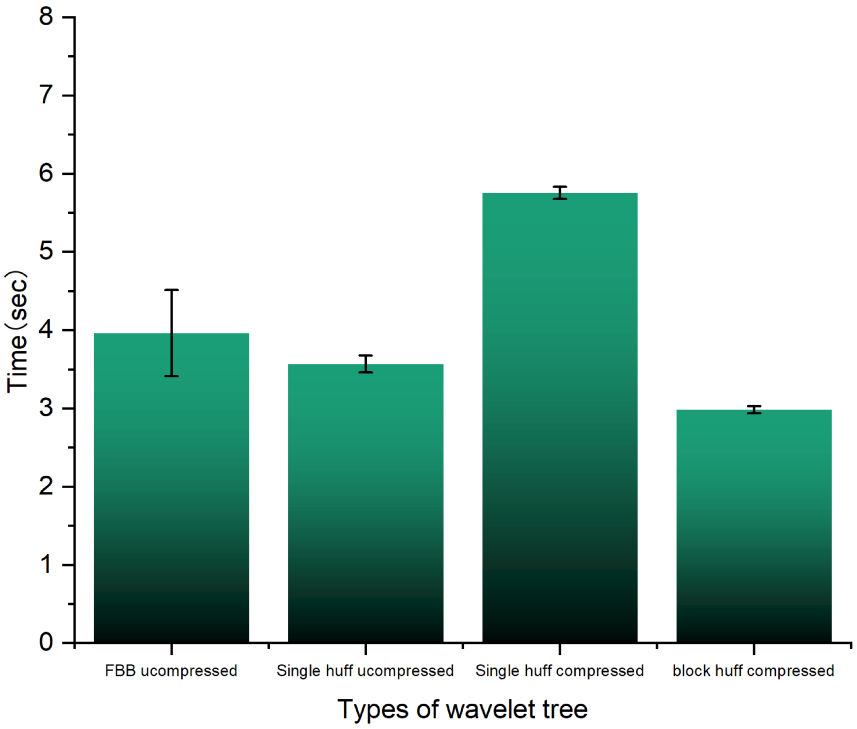

Figure 4.10: Substring extractions using FBB wavelet trees and other wavelet trees

Dell Inspiron 14-7472 with a substring length of 5000 bases, 1 GB of memory, and a swap file of 8 GB. Moreover, since it was shown that the METIS clustering does not improve substring extraction speed, only block partitioning was applied to the block wavelet trees involved in this stage of the experiment.

The FBB wavelet tree has four parameters, which consist of the type of underlying bit vector structure, the type of rank query, the logarithm with base 2 of the block size, and the logarithm with base 2 of the superblock size. For the initial tests, all four parameters were set to default values, where the vector structure was hybrid, the rank query was type 1, the block size was 2^{12} and the superblock size was 2^{20}, as reported by Gog et al.. Moreover, Gog et al. mentioned that if the size of the dataset exceeds 2^{32}, the dataset is divided into hyperblocks of size 2^{32} [11], and since the Douglas Fir dataset is larger than 2^{32}, this technique is used for the experiments. As shown in Figure 4.10, the slowest substring extractions are observed using the Huffman-shaped compressed single wavelet tree as it yields an average extraction time of 5.754 seconds for 5000 bases, which corresponds to the expectation because it is uncompressed and without block partitioning. Surprisingly, the second slowest performances are from the

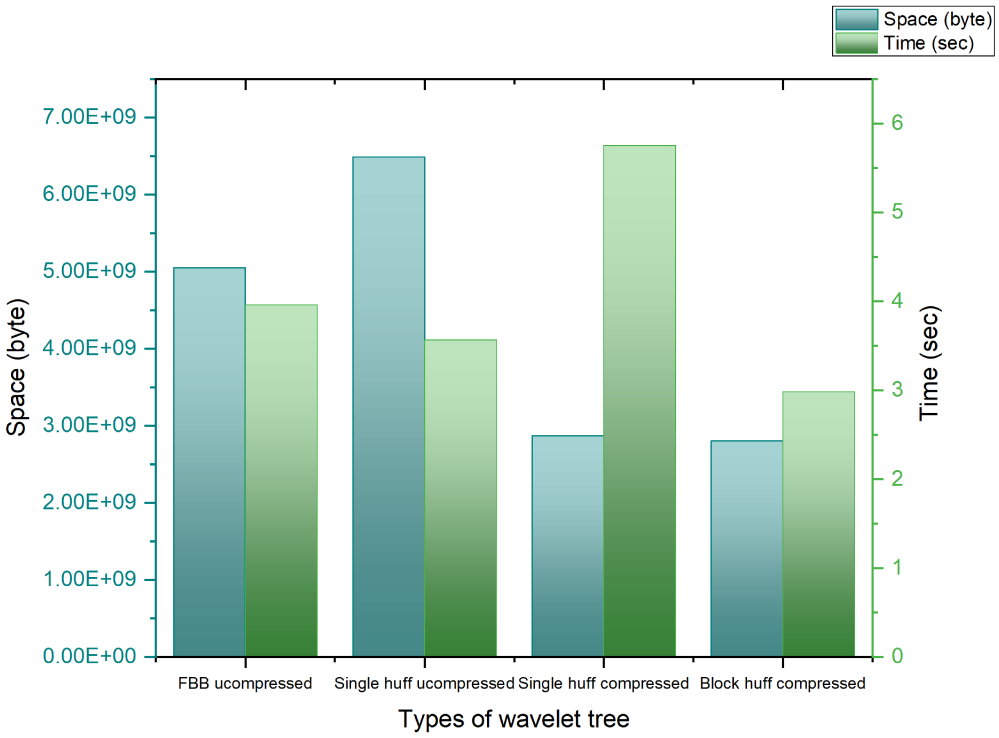

Figure 4.11: The time/space trade-off of substring extractions using FBB wavelet trees and other wavelet trees

FBB wavelet trees, which demonstrate an average extraction time of 3.963 seconds. However, since the block and superblock sizes were default numbers and the standard deviation of the substring extraction time was noticeably larger than other types of wavelet trees, it was expected that the FBB wavelet trees would perform better when the block and superblock size is further optimized. Moreover, the Huffman-shaped compressed single wavelet tree provides an average extraction time of 3.568 seconds, which is only 9.97% faster than the FBB wavelet trees. The Huffman-shaped compressed block wavelet trees still provide a similar average extraction time as the previous experiments, which is 2.984 seconds.

Furthermore, as shown in Figure 4.11 the tradeoffs between time and space among these wavelet trees were mainly fitted to the expectation. For the two types of single wavelet trees, it takes 2.7 GB of space when it is RRR-compressed and 6.1 GB when it is not, and the compressed version is 37.99% slower than the uncompressed version. Such results seem a reasonable tradeoff between time and space. As for

the FBB wavelet trees, they take both more space and time compared to the compressed block wavelet trees, which are an extra 1.1 GB of space and an extra 0.979 seconds, respectively. Hence, the compression ratio of FBB is not as high as the RRR-compression compression and the block partitioning did not offer a speed-up as much as the block wavelet trees from the data structure. Even though it was expected that the extraction time of the FBB wavelet trees would decrease with further block and superblock size optimizations, the required space remains the same, which means the block wavelet trees provide the best time and space tradeoff unless the FBB wavelet trees show a dramatic decrease in extraction time.

Since the FBB wavelet tree did not demonstrate better performance than the block wavelet trees with default settings, the next round of the experiment involved testing more block and superblock sizes in order to find a combination that offers better extraction speedup. According to Gog et al. [11], the maximum block size is 2^{16} and the maximum superblock size is 2^{24}. To account for any potential changes in the dataset, two relatively extreme block and superblock size combinations were initially tested: a block size of 2^6 with default superblock size and a superblock size of 2^{32} with default block size. Despite the warnings that occurred while building the FBB wavelet trees with these combinations, the wavelet trees were successfully built. However, both combinations were unsuccessful in extracting the substring, with the block size of 2^6 being killed by the system and the superblock size of 2^{32} resulting in a "segmentation fault (cored dumped)" error. More block and superblock sizes were tested in the range of 2^{11} to 2^{22} and 2^{20} to 2^{24} respectively, and extractions were completed without any errors or exceptions when the block size exceeded 2^{16}, but they actually had problems that have not been noticed. The remaining block and superblock combinations that were tested are listed with their extraction time in Table 4.2, and their extractions time are recorded in Figure 4.12.

log base 2 block size	11	16	16	19	19	20	20	20	21	22
log base 2 superblock size	24	20	24	21	24	20	21	24	24	24
Extraction time (seconds)	3.2	3.5	3.1	0.7	0.7	0.9	0.8	0.6	0.50	0.7

Table 4.2: Block and superblock sizes of FBB substring extractions

As shown in the plot, the altered block and superblock sizes result in better performances than the default combination. Even the worst combination provides an

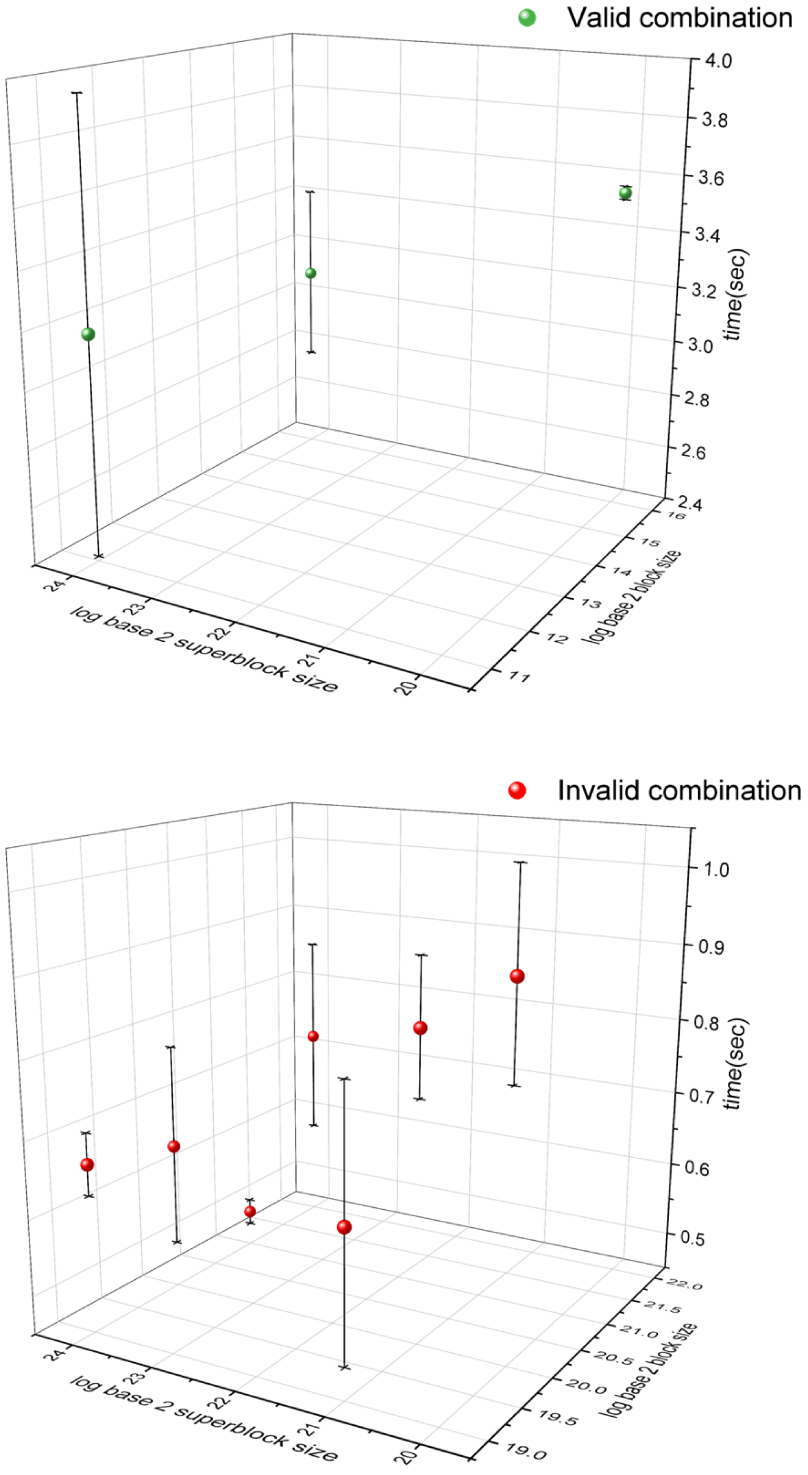

Figure 4.12: Substring extractions using FBB wavelet trees with different block and superblock size combinations

extraction time of 3.544 seconds, faster than the uncompressed Huffman-shaped single wavelet tree that previously outperformed the FBB wavelet trees. The substring extractions become considerably faster when the block size exceeds 2^{18}; all combinations with block size larger than 2^{18} can extract the substring of 5000 bases in under 1 second. The fast combination of block size 2^{21} and superblock size 2^{24} can extract the substring using just 0.49786 seconds with a small standard deviation of 0.01771. However, upon verifying the correctness of the extractions using FBB wavelet trees, it was found that the substrings were all incorrectly extracted when the block size exceeds 2^{16}, indicating that the maximum limit of the block size 2^{16} was still applicable for the Douglas Fir dataset. It was meaningless to search for the optimized block size beyond 2^{16}, some expectations would work though. Therefore, the most rapid extraction using FBB wavelet trees is actually 3.090 seconds, which is slower than the block wavelet trees of 2.984 seconds, but just by a small percentage. Moreover, the most rapid FBB substring extraction used the block size 2^{16} and the superblock size 2^{24}, which are the allowed maximum size of the block and superblock of the FBB wavelet trees. The performance of the substring extraction decreases along with the block and superblock size according to the tested combinations. Thus, 3.090 seconds should be the best result that can be produced with the FBB wavelet trees for 5000 bases.

At this point, all the experiments for optimizing the layout of the uncompressed BWT were finished, and the best result out of all the experiments is the 50% speedup with the block wavelet trees by block partitioning the BWT under circumstances. All the results will be discussed further in the following section.

4.3 Discussion of BWT Experiment

The primary objective of the BWT experiment was to investigate whether an optimized layout of the BWT could offer better access locality over the sequential layout, similar to the RLBWT experiment. The approach taken to achieve this objective was to compare the inversion times of the BWT using different wavelet trees constructed from both the sequential and optimized layouts of the BWT. The experiment was conducted on three datasets using various techniques to optimize the layout of the BWT. The results showed a notable decrease in the inversion time of the BWT when

an optimized layout was used; however, the reasons for this improvement did not quite align with the hypotheses before the experiment.

The first dataset used in the BWT inversion experiment was the Hamlet dataset, and the same techniques used in the RLBWT experiment were applied, which included block partitioning the BWT and clustering the blocks using METIS. Although no inversions were conducted on the Hamlet dataset, block partitioning and clustering the BWT still generated better results in terms of the sum of edge weights and distances, where the sum of edge weights decreased by a factor of 2.17 and the sum of jump distances decreased by a factor of 1.99. Based on these results, it was expected that the number of cache misses during the inversion would also be decreased. Furthermore, it was also expected that even better results could be achieved if the block orders within clusters are optimized, two techniques were used for this optimization with the same objective of reducing the product of all the edge weights and distances. First, the problem was treated as an MLA problem and solved using an ILP system by Gurobi; second, the problem was solved using the SA algorithm. Both techniques showed better results compared to sequential block orders within clusters, but the ILP gave a more significant decrease of a factor of 3 in the product of edge weights and distances. As the result, the Gurobi optimizer was chosen as the technique to permute the block orders within clusters.

After a solid data structure was implemented for the BWT inversion that uses a lookup table built by the optimized layout of BWT, the backward searches of the BWT were conducted in order to verify whether the optimized layout of BWT offers more cache or memory efficiency, which included inversions and substring extractions. The inversion tests were first conducted on portions of the HRG using the optimized layout with Gurobi-permuted block orders within clusters. However, the inversion time remained the same as the inversion with a just blocked partitioned and clustered layout, which led to the conclusion that the Gurobi optimization did not offer a better inversion speed of the BWT, despite effectively decreasing the product of the sum of edge weights and distances. This was contrary to the expectations from optimizing the block orders within each cluster. It could be due to the decrease in the average distance of LF steps does not imply an improvement of the inversion speed in practice, or the decrease is not significant enough to make a difference in terms of the inversion

speed. Either way, the Gurobi optimizer with the current setting was not helpful for achieving a better inversion speed and was suspended for future inversion tests. However, it is still believed that fully optimized block orders within each cluster would provide a better inversion time of the BWT.

Due to the suspension of Gurobi, the length of the inversion dataset was no longer limited. Therefore, the BWT of the whole HRG was used as the dataset for the inversions. Based on the inversion results of the HRG, it was also discovered that larger datasets and smaller memory setups are required to demonstrate the advantages of block wavelet trees in terms of memory efficiency. The Douglas Fir genome was chosen as the dataset, and a virtual machine with different memory sizes was used for future tests. As the correctness of the inversion data structure was proven, and it takes a huge amount of time to perform an inversion on an entire BWT, substring extractions were used to replace the inversions for time efficiency. Various combinations of cluster and block sizes were tested using the BWT of Douglas Fir with different memory sizes, and the block wavelet tree started to outperform the single wavelet when the memory size is 3G or smaller. However, the speedup is rather modest, reaching only about 14% when the block size is 10 million bases. Then, it was discovered that all the tests so far were conducted on a laptop with SSD and non-HDD, which may have caused the relatively small speedup. To address this, a new virtual machine was built on a laptop that offered HDD, and the experiments were re-conducted using the same range of block and cluster sizes. As expected, the speedup of the block wavelet trees was found to be bigger on the virtual machine with HDD. The most significant improvement of 50% occurred with a block size of 9 million bases with 1 GB available memory.

However, different cluster sizes did not exhibit a uniform trend in terms of speedup changes along with changes in block sizes. This raised questions about the validity of METIS clustering. A validation check was performed between block wavelet trees with and without METIS clustering. It was found that they almost shared the exact same trend, indicating that METIS clustering was not effective. The reason for this could be the use of incorrect parameters to estimate the decrease of the number of cache misses or at least an incorrect ratio of decrease. For instance, assuming that the decrease in the number of cache misses and the decrease of the sum of edge weights has

the same proportion. It could also be that the graph built from the block partitioned BWT is not as clusterable as expected, which means that the idea of clustering the graph is not an efficient process to optimize the layout of BWT. Moreover, METIS may not be the best tool for this particular graph clustering, and other existing graph clustering tools may give better performances.

Based on the results of previous experiments, it was clear that applying block wavelet trees offer speedup of the backward searches by simply block partitioning the BWT. However, the block sizes that were tested did not cover smaller block sizes that less than 1 million bases. This was due to the issue that the total size of wavelet trees would be extremely big when the block sizes were relatively too small compared to the length of BWT because the universal tables of the wavelet trees were not shared. To address this issue, the FBB technique implemented by Gog et al. was applied to build the block wavelet trees [11]. The FBB wavelet trees have two size parameters: block size and superblock size, which have maximum limits of 2^{16} and 2^{20} respectively. The dataset would also be divided into hyperblocks if its length exceeds 2^{32}. After trying different combinations of block and superblock sizes, the best correct substring extraction speed of 3.090 seconds was achieved by using a block size of 2^{16} and a superblock size of 2^{20}, resulting in a tiny disparity of 3.4% compared to the best extraction result of the block wavelet tree built from the data structure with a block size of 9 million bases. Since both block and superblock sizes have reached the maximum limits and the speedups of substring extractions grew along with the block and superblock sizes, it concluded that the 3.090 seconds would be the best result from FBB wavelet trees unless the maximum limit of block and superblock sizes increase.

Based on the extraction times collected from the FBB wavelet trees, it was suggested that smaller block sizes (i.e., smaller than 1 million bases) do not provide better performance over block sizes of millions for the Douglas Fir dataset. Therefore, due to the maximum limit of the block and superblock sizes, FBB wavelet trees could not outperform the block wavelet tree with a block size of 9 million bases built from the data structure. However, the extraction time of 3.090 seconds still outperformed the block wavelet trees with block sizes that are smaller than 7 million bases, according to the results of previous extractions. Therefore, if the FBB technique is

further optimized to accept bigger block sizes, such as 2^{21}, it is likely to show a better speedup than the block wavelet trees constructed by the data structure. On the other hand, limited maximum block and superblock sizes could be the nature of the FBB technique, which means the non-shareable universal table problem has to be solved by other approaches.

Chapter 5

Conclusion

The project was divided into two major experiments. The primary objective of this research was to reduce the processing time of BWT by optimizing its layout. The results of the experiments revealed that the optimized BWT representation was 50% better than the traditional BWT representation in terms of the time of BWT inversions with only block partitioning under the constraints of 1 GB memory, HDD for disk equipment, a genome of 14.67 GB, and a block size of 9 million bases.

Our original hypothesis was to improve the performance of RLBWT of pangenomic datasets because of the practical application value of pan-genome. Specifically, we employed inter-cluster edge weights as a proxy measure of the cache misses and decrease the edge weights using block partitioning and clustering. However, it was observed that the run length compression feature did not contribute to providing lower inter-cluster edge weights rather than causing extremely large clusters and negatively affecting the efficiency of clustering. On the other hand, after applying METIS clustering, new RLBWT layouts offered a notably smaller sum of edge weights compared to the traditional layout of the BWT. As a result, the run-length compression was discarded and block partitioning and clustering were applied on uncompressed BWT to seek a layout that provides better access locality. Furthermore, an ILP system with the Gurobi optimizer was applied to optimize the block order within clusters in order to gain a better layout since the cluster sizes were relatively big and blocks inside clusters were still sequentially stored. A data structure was implemented to apply the optimized BWT layout by using wavelet trees built from each block of the BWT with a rank header to form a look-up table to track the position of each block in the optimized layout. Additionally, a single wavelet tree was built from the entire BWT of the dataset as a comparison with the block wavelet trees with an expectation that block wavelet trees would outrun the single wavelet tree during backward searches, so it proves the optimized layout offers better access locality.

The backward searches were conducted on virtual machines with limited amounts of memory, and the virtual machines were hosted on two laptops, one with an SSD and the other with an HDD. The block wavelet trees showed more rapid backward search speeds than the single wavelet tree on both virtual machines. This indicated that the optimized BWT layout was causing fewer disk accesses compared to the sequential layout, making it a more access-friendly layout. A speedup of 14% was observed on the SSD laptop, while the HDD laptop showed a 50% improvement, with similar block sizes and setups. These results were consistent with expectations since accessing an HDD takes much longer time than accessing an SSD. However, the analysis showed that neither the Gurobi optimization nor the METIS clustering provided any help in reducing the inversion time of the BWT. Hence, the speedups were solely achieved by block partitioning the BWT, which was unexpected. In order to cover block sizes smaller than 1 million bases, which could not be tested using the block wavelet tree built by the data structure due to the non-shareable universal table, the FBB wavelet tree was introduced. It almost provided the same amount of improvement as the block wavelet tree with the optimal block size that was found. However, since the maximum limit of the block size of FBB is relatively small compared to the size of Douglas Fir, it could not eventually outperform the block wavelet trees.

The primary limitation of this project is that two of the main techniques that were proposed to optimize the layout of BWT were invalid. The reasons behind these invalidations could be that permuting the BWT by blocks is ineffective, but it is more likely that the layout is not optimized enough to make a visible difference when it comes to the inversion time. Furthermore, the Gurobi optimization was preset to abort and show the current result if the process could not finish within a reasonable time. Therefore, it is believed that there is still much further optimization that can be done on the optimization of the block order. Besides, due to the limited time for conducting experiments, some block sizes still have not yet been tested. For example, block sizes from 100 thousand bases to 1 million bases were too small for building block wavelet trees using the data structure and SDSL because of the issue of non-shareable universal tables. Also, they were too big for the FBB wavelet tree due to the maximum limit of its block size. Hence, the problem of non-shareable universal tables in the block wavelet tree and the limited block size of the FBB wavelet tree may also

limit the speedups. If the block wavelet tree can be built with a shared universal table, its overhead would be reduced dramatically, leading to an expectation of an even more rapid inversion process. As for the FBB wavelet trees, they almost outperformed the block wavelet trees with a much bigger block size. If its block size could be set to over 2^{16} bases such as 2^{21}, it would provide even more significant speedups.

This thesis has demonstrated that block partitioning a BWT and constructing a set of small block wavelet trees can lead to a better access locality of the BWT with limited memory, a genome sequence that can build a wavelet tree that exceeds the size of memory, and an HDD for disk equipment. However, there is still room for improvement in the performance of BWT backward searches even with the block partitioned representation. Future research could focus on solving the issue of non-shareable universal tables of the block wavelet trees or the maximum block limit of FBB wavelet trees. It is expected that solving either of these two problems will result in a more significant improvement of the inversion speed gained from block partitioning the BWT. In addition to the improvement gained from block partitioning the BWT, permuting the blocks within each cluster for a better layout is still a research direction that deserves more attention. It has the potential to provide a valid improvement in access locality with sufficient optimization. Exploring other software for graph clustering and block order optimization would be a good starting point. Furthermore, a more intuitive approach for estimating the number of cache misses could be an excellent alternative to replace the measures used to estimate the number of cache misses.

The data structure built in this thesis, which employs BWT block partitioning and the look-up table, can perform BWT inversion and substring extraction in a more cache-friendly manner. This creates a strong foundation for implementing a truly cache-friendly BWT. Furthermore, the challenges and invalidations encountered during this project provide valuable experiences for researchers intending to implement similar features on BWT or its other forms.

Bibliography

[1] Nathaniel Brown. Interval mapping of bwt-runs to efficiently compute lf-mapping in o (r) space. *16th Workshop on Compression, Text and Algorithms*, Oct 2021.

[2] Nathaniel Brown. *BWT-RUNS COMPRESSED DATA STRUCTURES FOR PAN-GENOMIC TEXT INDEXING*. Msc thesis, Dalhousie University, Dalhousie University, Apr 2023.

[3] Nathaniel K. Brown, Travis Gagie, and Massimiliano Rossi. Rlbwt tricks. *2022 Data Compression Conference (DCC)*, 2022.

[4] Nicos Christofides. Worst-case analysis of a new heuristic for the travelling salesman problem. *Operations Research Forum*, 3(1), 2022.

[5] George Bernard Dantzig. *Linear programming and extensions*. Princeton Univ. Pr, 1974.

[6] P. Ferragina and G. Manzini. Opportunistic data structures with applications. *Proceedings 41st Annual Symposium on Foundations of Computer Science*, 2000.

[7] Paolo Ferragina and Giovanni Manzini. Indexing compressed text. *Journal of the ACM (JACM)*, 52(4):552–581, 2005.

[8] Sven Fiergolla and Petra Wolf. Improving run length encoding by preprocessing. *2021 Data Compression Conference (DCC)*, 2021.

[9] Travis Gagie, Gonzalo Navarro, and Nicola Prezza. Fully functional suffix trees and optimal text searching in bwt-runs bounded space. *J. ACM*, 67(1), jan 2020.

[10] Simon Gog, Timo Beller, Alistair Moffat, and Matthias Petri. From theory to practice: Plug and play with succinct data structures. In *13th International Symposium on Experimental Algorithms, (SEA 2014)*, pages 326–337, 2014.

[11] Simon Gog, Juha Kärkkäinen, Dominik Kempa, Matthias Petri, and Simon J. Puglisi. Fixed block compression boosting in fm-indexes: Theory and practice. *Algorithmica*, 81(4):1370–1391, 2018.

[12] Roberto Grossi, Ankur Gupta, and Jeffrey Scott Vitter. High-order entropy-compressed text indexes. In *Proceedings of the Fourteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '03, page 841–850, USA, 2003. Society for Industrial and Applied Mathematics.

[13] Gurobi Optimization, LLC. Gurobi Optimizer Reference Manual, 2023.

[14] Michael Held and Richard M. Karp. A dynamic programming approach to sequencing problems. *Journal of the Society for Industrial and Applied Mathematics*, 10(1):196–210, 1962.

[15] G. Jacobson. Space-efficient static trees and graphs. *30th Annual Symposium on Foundations of Computer Science*, 1989.

[16] Anil K. Jain and Richard C. Dubes. *Algorithms for Clustering Data*. Prentice-Hall, Inc., USA, 1988.

[17] Xiaofang Jiang, Qinghui Liu, N. Parthiban, and R. Sundara Rajan. A note on minimum linear arrangement for bc graphs. *Discrete Mathematics, Algorithms and Applications*, 10(02):1850023, 2018.

[18] George Karypis and Vipin Kumar. A fast and high-quality multilevel scheme for partitioning irregular graphs. *SIAM Journal on Scientific Computing*, 20(1):359–392, 1998.

[19] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. Optimization by simulated annealing. *Science*, 220(4598):671–680, 1983.

[20] Ben Langmead and Steven L Salzberg. Fast gapped-read alignment with bowtie 2. *Nature Methods*, 9(4):357–359, Mar 2012.

[21] Heng Li and Richard Durbin. Fast and accurate short read alignment with Burrows–Wheeler transform. *Bioinformatics*, 25(14):1754–1760, 05 2009.

[22] Linux. Ulimit. `https://ss64.com/bash/ulimit.html`.

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

[24] Giovanni Manzini. An analysis of the burrows—wheeler transform. *J. ACM*, 48(3):407–430, may 2001.

[25] Alex Murillo, J. Fernando Vera, and Willem J. Heiser. A permutation-translation simulated annealing algorithm for l1 and l2 unidimensional scaling. *Journal of Classification*, 22(1):119–138, 2005.

[26] Gonzalo Navarro and Veli Mäkinen. Compressed full-text indexes. *ACM Computing Surveys*, 39(1):2, 2007.

[27] NCBI. Grch38.p14 - hg38 - genome - assembly - ncbi. `https://www.ncbi.nlm.nih.gov/assembly/GCA_000001405.29`.

[28] David B Neale, Patrick E McGuire, Nicholas C Wheeler, Kristian A Stevens, Marc W Crepeau, Charis Cardeno, Aleksey V Zimin, Daniela Puiu, Geo M Pertea, U Uzay Sezen, and et al. The douglas-fir genome sequence reveals specialization of the photosynthetic apparatus in pinaceae. *G3 Genes—Genomes—Genetics*, 7:3157–3167, 2017.

[29] Takaaki Nishimoto and Yasuo Tabei. Optimal-time queries on bwt-runs compressed indexes. *CoRR*, abs/2006.05104, 2020.

[30] Rajeev Raman, Venkatesh Raman, and S. Srinivasa Rao. Succinct indexable dictionaries with applications to encoding k-ary trees and multisets. In *Proceedings of the Thirteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '02, page 233–242, USA, 2002. Society for Industrial and Applied Mathematics.

[31] Ilya Safro, Dorit Ron, and Achi Brandt. Graph minimum linear arrangement by multilevel weighted edge contractions. *Journal of Algorithms*, 60(1):24–41, 2006.

[32] M. Schindler. A fast block-sorting algorithm for lossless data compression. *Proceedings DCC '97. Data Compression Conference*, 1997.

[33] William Shakespeare. Hamlet. `https://github.com/second12138/Master-s-thesis/blob/main/hamlet_short.txt`.

[34] William Shakespeare. Hamlet. `https://www.gutenberg.org/files/27761/27761-0.txt`.

[35] A. M. Shrestha, M. C. Frith, and P. Horton. A bioinformatician's guide to the forefront of suffix array construction algorithms. *Briefings in Bioinformatics*, 15(2):138–154, 2014.

[36] Jouni Sirén, Erik Garrison, Adam M Novak, Benedict Paten, and Richard Durbin. Haplotype-aware graph indexes. *Bioinformatics*, 36(2):400–407, 2019.

[37] William M. Springer. Review of the traveling salesman problem: A computational study by applegate, bixby, chvátal, and cook (princeton university press). *ACM SIGACT News*, 40(2):30–32, 2009.

[38] Chvatal Vasek. *Linear Programming*. W.H. Freeman and Company, 2002.

[39] Huanchen Zhang, Hyeontaek Lim, Viktor Leis, David G. Andersen, Michael Kaminsky, Kimberly Keeton, and Andrew Pavlo. Surf: Practical range query filtering with fast succinct tries. *Proceedings of the 2018 International Conference on Management of Data*, 2018.