https://www.overleaf.com/project/6376a4c54bac8f9d6266c44c

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
April 2023

Optionally, the thesis can be dedicated to someone, and the student can enter the dedication content here.

Contents

List of Tables

List of Figures

Abstract

Burrows-Wheeler Transform (BWT) is a widely used succinct data structure in bioinformatics, particularly for compressing datasets, aligning sequences, and matching patterns. However, one of the main concerns that researchers still face when using BWT is its high randomness during backward searches. Although a BWT employs the (Last to First) LF mapping property 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 or even several.

Our project aims to reduce the number of cache misses caused by conducting backward searches of a BWT to improve memory locality. This objective is achieved by blocking, clustering, and permuting the layout of the BWT. Although some of the techniques we applied did not effectively enhance the speed of backward searches, the optimized layout still resulted in a 50% improvement in the speed of backward searches.

Acknowledgements

Nothing yet.

Chapter 1

Introduction

Compressing large highly competitive strings such as the human genome is an essential aspect of research in the field of bioinformatics. According to the National Centre of Biotechnology Information (NCBI), human chromosome 19 alone has almost 59 million bases [26]. Succinct data structures are one of the main approaches for compressing large competitive datasets, and the Burrows-Wheeler Transform (BWT) is one of the most popular techniques. BWT is a lossless data compression algorithm that restructures the input string [2]. Technically, a string of characters is transformed to its BWT by listing all the permutations in lexicographical order, which is formed as the Burrows-Wheeler Matrix (BWM), then 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 characters that have the same data value occurring in consecutive order [28].

Both BWT and RLBWT can be reversed to their original strings using a key property called Last-to-First (LF) Mapping. The ith occurrence of a character c in the last column of the BWM and the ith occurrence of the same character c in the first column correspond to the same occurrence in the input string [32]. However, reversing BWT and RLBWT to their original strings is practically slow, despite the algorithm's fast implementation, due to the randomness of the backward steps of LF mapping. As a result, the randomness of LF mapping leads to an extremely high number of cache misses, with almost every backward step causing several cache misses during the backward searches [7].

Sirén et al. proposed an approach in 2020 by rearranging BWT to achieve a better memory locality for the graph extension of the positional BWT [36], and it inspired us to consider applying such a method on either RLBWT or regular BWT to improve memory locality and reduce the number of cache misses, leading to a faster backward search speed. Therefore, the ultimate goal of this project is to reduce the number of

cache misses during the reversion of either regular BWT or RLBWT using Sirén et al.'s algorithm, with the expectation of reducing several cache misses per LF jump to one cache miss for several LF jumps. To achieve this objective, a data structure will be constructed that supports *access* and *rank* queries of a BWT or RLBWT. It contains a table built from blocked, clustered, and permutated BWT, ensuring that indices of BWT that likely jump to each other have stayed as close as possible. Both RLBWT and BWT are potential targets for layout optimization, but since Nishimoto et al.'s data structure has recently made progress on computing the LF-mapping of RLBWT, we will first focus on optimizing the layout of RLBWT to see if we can improve its performance.

As the current state-of-the-art method for computing the LF mapping of the RLBWT, Nishimoto et al.'s data structure achieves constant-time LF mapping using a lookup table with $O(r)$ space, where r is the number of runs in RLBWT [28]. However, due to the size of bioinformatics datasets, the $O(r)$ lookup table still needs to be stored in memory or partially on disk, which means Nishimoto's lookup table does not provide better memory locality than traditional data structures when it comes to reversing RLBWT. This can still result in a large number of cache misses. Therefore, the first experiment aims to determine whether applying Sirén et al.'s algorithm for BWT rearrangement to Nishimoto et al.'s lookup table can provide better memory locality for RLBWT reversal while maintaining constant running time.

If the RLBWT layout optimization does not result in a satisfactory reduction in cache misses, a second experiment will be conducted that will focus solely on the uncompressed BWT, discarding the run-length compressed feature. Since no look-up table similar to the RLBWT table proposed by Nishimoto et al. currently exists, the uncompressed BWT will be built from scratch. One of the best tools for regular BWT construction is the Succinct Data Structure Library (SDSL), developed by Gog et al. [9], which can compute the BWT of a text string in $O(\log n)$ time. Therefore, SDSL will be used to construct the BWT of an input string and then apply the same blocking, graphing, and clustering procedure used in the RLBWT experiment to see if it leads to better memory locality.

For the structure of the rest of the thesis, chapter 2 will provide a comprehensive background and an extensive overview of the related topics for the two experiments

conducted in this thesis. In Chapter 3, the RLBWT experiment will be presented, along with its results and discussions. Chapter 4 will focus on the BWT experiment, which will be conducted based on the findings of the previous experiment, and will also contain the results and discussions of the BWT experiment. Finally, the thesis will conclude with a summary of the results from both experiments, along with directions for future research.

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. Jacobson et al. introduced these data structures for the first time in 1989; they are especially beneficial for applications that require data to be processed in real-time and stored in memory for faster access [13].

There are various instances of succinct data structure have been developed, and one of the most widely used examples is the Succinct Trie, the Succinct 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 [40]. The Succinct Trie enables effective string searches, string insertion and deletion within the collection. Moreover, succinct Arrays and Succinct Bitvectors are also considered as popular succinct data structures. Whereas, the Succinct Array is a compact variant of an array that permits rapid access to individual array elements [9]. As the Succinct Bitvector, it permits rapid access to individual bits, as well as operations such as rank and select, which are used to count the number of bits set to 1 within a given range of the vector [9]. Both are commonly exploited in applications for data compression and retrieval.

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 [9]. SDSL is designed for handling data structures that could be represented succinctly or compactly, allowing for more effective memory utilization and faster processing times. SDSL is applicable to a variety of applications, including text processing and data compression. For the majority of this project, SDSL is used to create wavelet trees BWT and suffix arrays, which are discussed in the following sections.

2.2 The Burrows-Wheeler Transform

BWT is a well-known text compression and sequential analysis tool that has become commonly recognized since Michael Burrows and David Wheeler initially introduced it in 1994 [32]. The central concept of 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 [2]. 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 BWT particularly helpful in bioinformatics fields such as DNA or protein sequencing [2]. If the input string is generated as $S = s_1, s_2, ..., s_n$, its BWT matrix (i.e. cyclic rotations of S) can be written as $S_0 = s_1, s_2, ..., s_n, S_1 = s_n, s_1, s_2, ..., s_{n-1}, S_{n-1} = s_2, ..., s_n, s_1$ Thus the BWT of S is the final column of the BWM, represented as $BWT(S) = (e_1, e_2...e_n)$, where $e_i = s_i$ for each i.

The BWT can be reversed to the original input string, and the reversibility of the BWT is provided using a technique known as LF mapping because each character in the first column has the same rank as each character in the last column in the BWM [32]. Yet, one of the disadvantages of BWT is the randomness 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 leads to poor memory locality in BWT, which is one of the reasons BWT has a high computational cost; hence, the BWT layout requires further improvement.

2.3 The Run-length encoding

Similar to the BWT, Run-length encoding (RLE) is another approach for lossless data compression but relies on building a coding scheme [8]. An input string is encoded by runs, and each run is defined as a sequence of consecutive identical characters as long as it lasts. Usually, σ_i is used to represent a run, where σ is a character from the alphabet of the input string and i is its number of occurrences in the current consecutive sequence [8].

For instance, a short DNA sequence read is formed as "$AAGGGGGGGTTTTCCC$"

and can be encoded to "$A_2G_7T_4C_2$". For the character of a run, there is a fixed number of bits d assigned to it for encoding and storing its binary representation, and the length of the current run is also encoded and stored in binary format using fixed e bits [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. Since $e = d = 8$ bytes, and the run-length compressed string "G_4T_4 only need 4 bytes. However, if the input string has little or no consecutive repetitions but mixtures of different characters, then the RLE would become a drawback. For example, "$GTGTGTGT$" shares the same alphabet and length with the previous string but it takes 16 bytes to store its run-length compressed form "$G_1T_1G_1T_1G_1T_1G_1T_1$" for $e = d = 8$ [8].

The number of repetitions in the input string plays an essential aspect in whether the run-length compressed string is going to explode in terms of the size, which is the least expectation from a string compression algorithm. Therefore, the input data has to be fitted for RLE as a compression scheme [8]. Since BWT tends to compress the input string to a more consecutive style, combining RLE as a compression scheme for BWT gains an even better compression than regular BWT, which is RLBWT.

2.4 Suffix Array

The suffix array is a data structure that provides a high compression ratio of a string's suffixes and is arranged in lexicographic order [22]. A suffix array of a string S is an array of integers containing the starting positions of the sorted suffixes of S. The length of S is denoted by n and the suffix array is an array of length n. Meanwhile, the ith element of the suffix array represents the starting point of the ith sorted array [22]. Moreover, the fundamental advantage of suffix array over other traditional string search approaches, such as suffix tree, is its capacity for carrying out efficient online string searches. Besides, the suffix array is also more space-efficient and easier to design. Specifically, the suffix array is implementable in linear time $O(n)$ and linear space $O(n \log n)$ [22].

Suffix array has been applied in many different fields of string searches, and one of the primary applications of 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, this can be achieved in linear time $O(m + \log n)$, in which m is the length of P and n is the length of T [22]. This is accomplished by conducting a binary search on the suffix array to find the domain of suffixes containing P as a prefix [22]. Moreover, the suffix array can be used to compress strings by identifying patterns that occur repeatedly, similar to the RLE that has been mentioned. Such that the text can be compressed by replacing all occurrences of repeated patterns with references to a specific instance of the pattern [22].

Suffix array has also become a common bioinformatics tool like BWT, especially for DNA or protein sequence alignment and pattern matching across numerous sequences [34]. In addition, suffix array is employed in the sequence alignment to rapidly select the best match between two sequences, as well as in pattern matching to locate all instances of a particular pattern in a DNA or protein sequence, using software such as Bowtie [19].

2.5 Wavelet Tree

Wavelet tree is a type of tree-shaped succinct data structure that has been extensively explored and employed for the effective processing and retrieval of vast quantities of data [11]. It was first proposed and implemented by Grossi and Vitter in 2003, and it has been intensively explored and modified since then [11]. Numerous efficient techniques and data structures have been created for the construction and application of the wavelet tree. In terms of its format, it 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 operations for data operations [11].

Wavelet tree leverages a binary tree structure to represent data, making it possible to produce condensed representations of vast data sets that can be processed efficiently. Consequently, it facilitates the preservation of compact representations while permitting a wide variety of operations [11]. The wavelet tree has also been utilized in a variety of disciplines, including text processing, image compression, and range searching. For instance, the wavelet tree is applied in text processing for efficient pattern and string matching [24]. The wavelet tree has been applied in image

compression for the efficient representation and compression of images [4]. In addition, the wavelet tree has been deployed to efficiently query and interpret big datasets during range searching [38].

Despite its benefits, the wavelet tree has its own disadvantages. It is not always worthwhile to build a wavelet tree of the datasets, especially for smaller ones, due to the fact that it might be complicated and difficult to construct because of the overheads such as the universal table [11]. 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 dataset 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 individual requirements. Although, differently shaped wavelet trees are employed for access and ranking queries in this thesis.

2.6 Sirén's algorithm

Sirén improves the memory locality of the graph extension of the positional BWT (pBWT) in his Graph BWT implementation by reducing the number of cache misses. Based on pBWT and its graph extension, the GBWT is implemented as a Ferragina-Manzini (FM) index of multiple texts over an integer alphabet V that answers find, locate, and extract queries. For each vertex $v \in V$, a local alphabet is defined as $\sum v = w \in V|(v, w) \in E)$. \$, which is added to $\sum v$ if v is the last vertex of a path, and $\sum \$$ is defined as the initial vertex of each path [36]. A BWT is partitioned to substrings $\mathrm{BWT_v}$ corresponding to the prefixes ending with v, and each $\mathrm{BWT_v}$ is encoded using $\sum v$. Let $\sum v(w) = k$ when $w \in \sum v$ is the kth character in $\sum v$ in sorted order [36]. There is a record corresponding to each $v \in V$ and end-mark \$, which is composed of a head and a body. The records for adjacent vertices are stored close to each other to decrease the number of cache misses caused by the jumps of LF-mapping [36]. Meanwhile, it is assumed that the graph layout is cache-friendly such that each vertex $v \in V$ has a low out-degree, is almost linear, and is topographically sorted. These are reasonable assumptions for a large set of biological haplotype sequences over a variation graph [36].

The GBWT construction algorithm is based on a dynamic FM index and uses the

Biconjugate Residual (BCR) algorithm to insert multiple texts into the index in a single batch [36]. When construction using a single batch is too slow, the dataset is partitioned into super-batches and separate GBWT indexes are built for each super-batch. Next, the indexes are merged using the BWT-merge algorithm, and the merge algorithm can also be reversed for removing texts from the index [35]. After that, the haplotype information in the GBWT can be used to simplify the variation graph (VG) for k-mer indexing with a series of pruning heuristics, which means the k-mers from the recombinations are pruned, and k-mers from the haplotypes are kept in the index [36].

2.7 Nishimoto's data structure

In order to store an arbitrary permutation P of the string $S = 0, ..., n-1$ with a property that the sequence $P(0), P(1), P(2), ..., P(n-1)$ is composed of a limited number b of unbroken incrementing subsequences. A $O(b)$ space-compressed sparse bit vector was widely used, which requires $O(\log \log n)$ time and $O(r)$ space, where r is the number of runs in the BWT [2]. 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 $O(r)$[28]. Specifically, if an array T is used to represent an RLBWT table of length r that was built from a BWT with size n, for each subsequence head p, there is a row of the table T_i that corresponds to it, which is a quadruple that consists of p, the length of the subsequence $|p|$, the interval $P(p)$, and the offset of $P(p)$[2].

The reversal of RLBWT is achieved by jumping back and forth in each row of the lookup table, starting from the row of the end mark \$ [28]. For each row T_i, its index is the starting position of the LF mapping jump, $P(p)$ is the ending position of the jump, and the subsequence head p in each row is the character that is reversed from the current jump [28]. There exists a particular case of LF jumping, which is crossing the boundary. When the offset is larger than the length of the subsequence $|p|$, the boundaries of the rows are crossed to find the ending position of the current jump, rather than just jumping to the row specified by the interval $P(p)$. However, Brown et al. have proved that this boundary-crossing situation rarely happens [2].

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. Therefore, using the lookup table to compute the LF mapping of RLBWT will still cause a significant number of cache misses.

2.8 Graph clustering and METIS

Graph clustering is the technique of separating a graph into subgraphs or clusters based on similarity criteria [16]. 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. There are a number of techniques for clustering graphs, including hierarchical clustering, k-means clustering, and spectral clustering [14].

An algorithm of dividing irregular graphs into smaller sub-graphs that was proposed by Karypis et al. in 1994 indeed contributed to the field of graph clustering [16]. The approach is based on a multilevel system that employs coarsening, initial partitioning, and refining to generate partitions of superior quality. The multilevel approach 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. The coarsening phase is performed until the graph is small enough to be partitioned with an initial partitioning technique. Next, 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. The examination of the technique on numerous benchmark graphs reveals that it creates high-quality partitions of graphs in a reasonable amount of time. Moreover, it is proved that the technique outperforms various well-known graph partitioning algorithms in terms of partition quality and execution time [16].

In addition, METIS, a graph partitioning software package, was developed using this concept. METIS is frequently used to break large-scale graphs and hypergraphs into smaller subgraphs for applications such as parallel computing, circuit design, and mesh generation [16]. 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 an arrangement of the vertices of a complete graph that ensures that the total weight of the edges is minimised [15]. Given an undirected graph $G = (V, E)$, where V is the set of vertices and E is the set of edges, find a permutation pi of the V such that the total weight of the E is reduced, where the weight of an edge (u, v) is defined as the distance between the locations of u and v in permutation pi [15].

Many techniques have been developed to address the problem including heuristics algorithms, approximation algorithms, and exact algorithms [37]. With heuristics 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 [3]. For instance, the nearest neighbour algorithm selects the closest unordered vertex to the last ordered vertex as the next vertex in the ordering [15]. With approximation methods such as the Christofides algorithm, they attempt to provide a solution that is within a specified ratio of the ideal answer. The Christofides algorithm, for instance, provides a 3/2 approximation for the MLA problem [3]. Whereas the Exact algorithms such as branch and bound algorithms and branch and cut algorithms, guarantee finding the optimal solution [12]. These algorithms function by systematically examining the solution space and eliminating any branches that cannot lead to a better solution [12].

The approaches that have been mentioned were all proposed a while ago, and subsequent studies have been based on them. In recent years, a number of 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 tackle issues involving the optimization of a linear objective function constrained by linear equations or inequalities [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 some or all decision variables are limited to integer values [39].

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

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

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, $b_1, b_2, ..., b_m$ are the right-hand side values of the constraints, and ≥ 0 indicates non-negativity of the variables [5]. In addition, if it is an ILPproblem, then some or all decision variables would be restricted to integer values, which means rather than $x_1 \geq 0, x_2 \geq 0, ..., x_n \geq 0$, $x_1, x_2, ..., x_n$ would be integers.

Both LP and ILP have been implemented across diverse fields. The applications of LP include resource allocation, portfolio optimization, and transportation planning. Specifically, the LP is employed in resource allocation to allocate resources so as to maximize the target function while satisfying constraints [5]. For Portfolio optimization in finance, it employs LP to enhance investment portfolios [5]. In

transportation planning, LP is used to determine the optimal distribution of transportation resources such as trucks, trains, and ships [5]. In contrast, ILP applications are generally concerned with discrete decisions, such as scheduling, planning, and telecommunications [39]. Specifically, the ILP can be used in scheduling to identify the optimal schedule for a collection of assignments according to constraints such as resource availability and time constraints [39]. Meanwhile, the ILP can be used in planning to determine the optimal plan for a series of activities subject to constraints such as budgetary constraints and resource availability [39]. Besides, the ILP can be used in telecommunications to determine the optimal utilization of communication capabilities like bandwidth and frequencies [39]. Moreover, with the development of computing technology and the availability of specialized software such as Gurobi, LP and ILP have become vital tools in the field of operations research and continue to play a significant role in the decision-making processes of numerous industries. For this project, an ILP model is built to further improve the block ordering within a cluster.

2.11 Simulated Annealing and Unidimensional Scaling

The robust optimization approach Simulated Annealing (SA) has been successfully applied to tackle challenging combinatorial optimization issues [17]. It is a metaheuristic approach that imitates the cooling process of metals. Initially, SA starts with a predefined solution featuring the starting values of its parameters. 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 seek additional areas of the solution space and escape local minima, possibly resulting in a global optimum [17].

In terms of applications of SA, Murillo et al. present a novel SA algorithm for the unidimensional scaling (UDS) problem [23]. UDS is a data analysis technique used for reducing the dimensionality of a dataset while preserving its intrinsic structure [18]. The UDS problem is usually solved by minimizing the stress function, which measures

the difference between the original dissimilarities and the Euclidean distances in the reduced space [18]. Differently, the SA algorithm proposed by Murillo et al. performs UDS by simultaneously permuting and translating the coordinates of the points in the reduced space [23]. The algorithm solves the UDS problem in $O(n^2 \log n)$ time, where n is the number of points, and it outperforms other existing SA algorithms and heuristics in terms of the solution quality and computation time. Murillo et al. conducted extensive experiments on various datasets, including the classical multivariate normal, Swiss Roll, and Friedman's tensor product grid, and showed that the proposed algorithm is more accurate and efficient than other existing UDS methods. Inspired by this paper, SA was also applied in this project for optimizing block orders.

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 SA to facilitate rapid and effective pattern matching, substring search, and other text processing operations [6]. In particular, the FM-Index maintains the BWT, the SA, 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. 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, FM-Index has been extensively employed for tasks such as genome assembly. It has been demonstrated that they are highly efficient because of their capacity to manage enormous volumes of data and perform rapid pattern matching and substring search [25].

Fixed block compression boosting (FBB) by Gog et al. is one of the most cutting-edge techniques for optimizing the time and space efficiency of FM-Indexes further [10]. 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 duration. 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 [10]. Gog et al. shown that FBB can drastically reduce the size of the FM-Index, often by more than 50%, while preserving the index's rapid search performance [10]. As a result, FBB decreases the size of the FM-Index while keeping its functionality, such as the shared universal table of the blocks, by breaking the BWT into blocks and applying lossless compression to each block.

2.13 A cache-friendly layout proofing of the graph built from Nishimoto's look-up table

The feasibility of implementing a cache-friendly feature onto RLBWT is based on the assumption that the graph built from Nishimoto's look-up table has a cache-friendly layout, and one of the essential aspects is that each node has a low out-degree. Brown et al. proved that the occurrence of boundary crossing is low in practice for pangenomes, which indicates that each node in the graph has a low degree because the graph is undirected, and the edges represent the jumps spanned in different blocks. In addition, my superior Dr.Travis Gagie proposes the following theorem with proofing.

Theorem 1. *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 about $d + \sigma$, 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 3rd 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. Brown's paper shows that d is small in practice [1].

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 $\sigma(d/\sigma + 1) = d + \sigma$. □

Furthermore, we believe that the graphs built from uncompressed BWT also hold the same characteristics.

Chapter 3

RLBWT reversion

3.1 RLBWT reversion methodology

3.1.1 Graphing and clustering using RLBWT Table

At the beginning of the project, the aim was to build graphs and cluster them from the RLBWT tables that were proposed by Nishimoto et al.[28]. A preliminary experiment of graph building and clustering was conducted using a short string example to demonstrate the feasibility of the blocked and clustered BWT layout. Furthermore, the performance of the graph building and clustering was then assessed by calculating the total weight of inter-cluster edges using real genome datasets.

Preliminary graph building and MTEIS clustering

Before conducting experiments on real genome datasets, a small string example was used to demonstrate the feasibility of implementing the cache-friendly feature onto RLBWT. A lookup table was derived from the string "bananaband$" as shown in Table 3.1, which was collected from Brown et al in WCTA 2021 [35], who implemented Nishimoto's lookup table from theory.

Index	Character	Length	Interval	Offset
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 3.1: RLBWT Look-up table of "bananaband$"

Firstly, the table is divided into four blocks, each containing two rows, which seems to be an appropriate block size based on the string size of 11. Next, the blocks

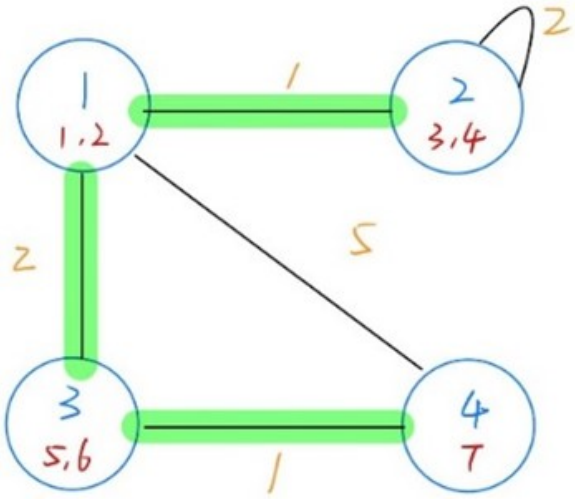

Figure 3.1: Preliminary graph clustering: step 1.

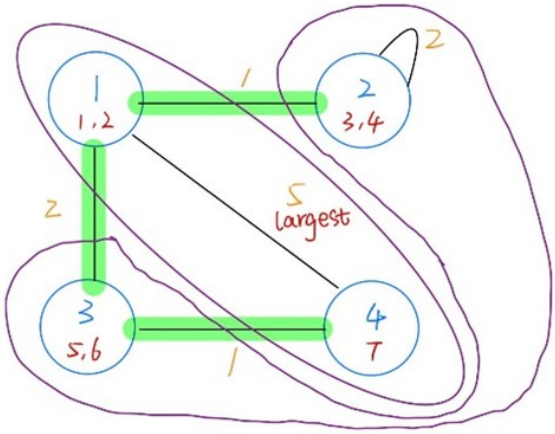

Figure 3.2: Preliminary graph clustering: step 2.

are used as nodes, and the sums of the lengths of the inter-block jumps are used as edges to construct a graph, as shown in the Figure 3.1 .

After building the graph, the self-connected edges in each block are ignored as they are considered meaningless for the implementation. Next, the graph is clustered using an algorithm that takes inspiration from Sirén's method of rearranging pBWT. The most weighted edges between two un-clustered nodes are chosen and clustered into the same group. This process 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 between block

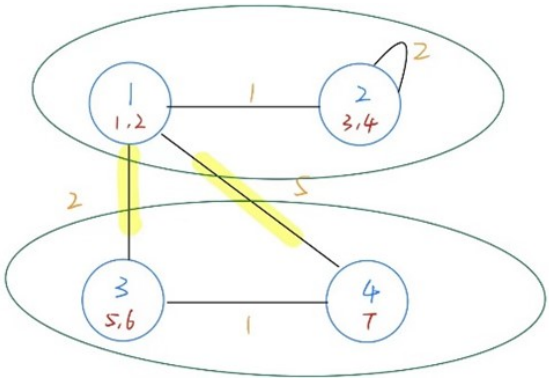

Figure 3.3: Preliminary graph clustering: step 3.

1 and block 4 has been removed, while the other three edges have been retained. Furthermore, block 1 and block 4 have been grouped together, and block 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.

Subsequently, the blocks are clustered sequentially as a simulation of the current LF mapping processing approach, and the total weight of inter-cluster edges is computed, as illustrated in Figure In Figure 3.3, the sum of the inter-cluster edges increases to 7. This small dataset suggests that implementing the cache-friendly feature onto RLWBT by rearranging the look-up table would work. 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.

Real genome graph building and METIS Clustering

The RLBWT tables that are built from real genomes were provided by Brown et al.. After reading the look-up table from a binary file, the procedure mentioned in preliminary clustering is followed to cut it into blocks based on the $L1$ cache size. For this particular experiment, 1024 rows were used, which is small enough to be stored in common $L1$ cache of $64K$. The next step involves building the graph. The procedure can be described as follows: the blocks are used as vertices $v \in V$, and the jump of LF mapping 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 total length of runs between two vertices $(v, u) \in V$, and edges that connect to the same block are ignored since they do not affect anything. The graph construction for genome

datasets is implemented using C++.

After building the graph using the RLBWT table, it is clustered using METIS with different numbers of clusters to observe the differences in the total weights of inter-cluster edges. The same clustering algorithm as the one used in the preliminary experiment is applied in METIS. This algorithm involves finding $e = (v, u)$ with the largest w, placing v and u into the same cluster, and continuing to cluster e with the next largest w until reaching the size of a cluster. Once all the blocks have been partitioned into clusters, the total weight W_{r} of inter-cluster edges $e_{\mathrm{t}} \in E_{\mathrm{t}}$ is calculated.

As a comparison to the METIS clustering approach, the graph G is also clustered sequentially with the same number of clusters. Since the graph is clustered sequentially, the clustering is performed based on the index of the blocks in the table, and all clusters have similar sizes. The total weight of the inter-cluster 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 reversion results

3.2.1 Salmonella dataset

In the first experiment, the RLBWT table was built from the Salmonella genome sequence, and some statistics of this dataset are provided. The sequence length is 145,595,456, and the number of runs is 12,823,516, resulting in an n/r ratio 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 3.2. 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.

In addition, a scatter graph is used to virtualize the difference between MEITS clustering and sequential clustering. As shown in Figure 3.4, until 100 clusters there is a clear difference in weights between METIS clustering and sequential clustering. The METIS is still 2 times smaller than sequential clustering when the cluster number is

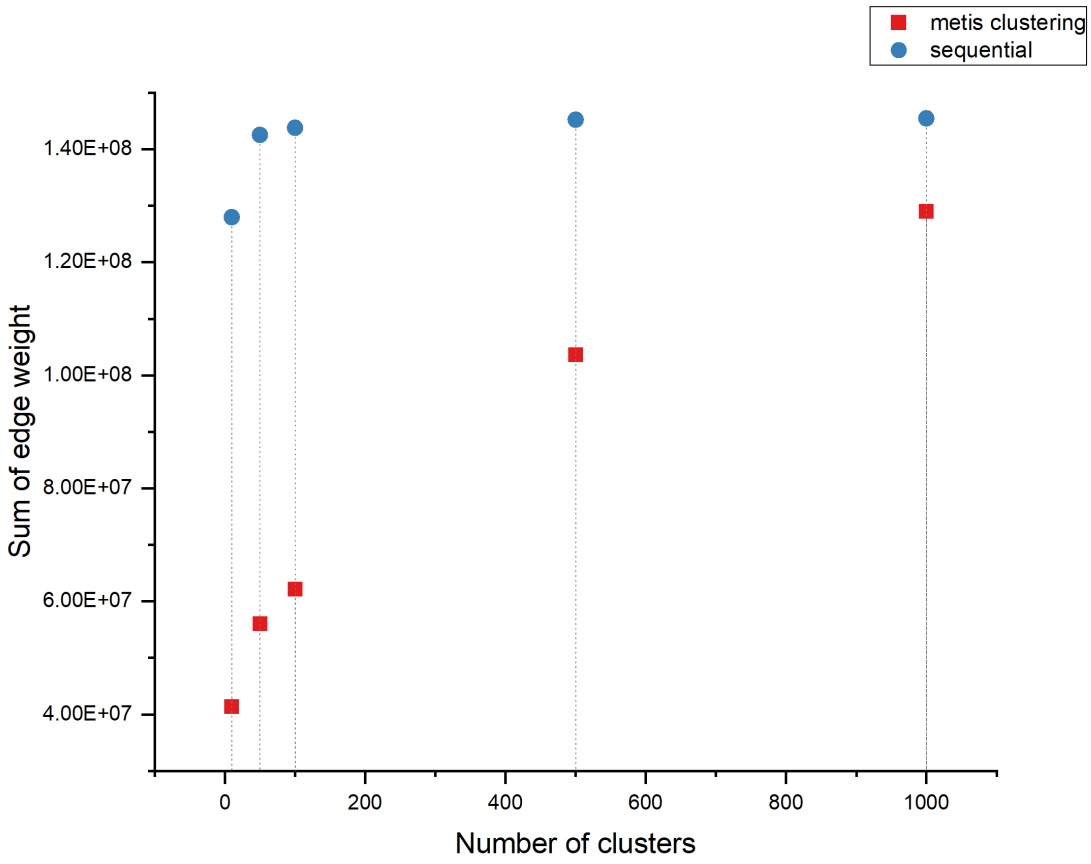

Figure 3.4: Salmonella weight change before and after METIS clustering.

#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 3.2: RLBWT Look-up table of "bananaband$"

100. Once the cluster number is over 100, the weight difference shrinks significantly.

As a result, the salmonella dataset demonstrated a decrease in cache misses by a factor of 3 when the cluster number was 10. However, since the n/r ratio was not significantly large in this dataset, the input string was not super consecutively competitive. Therefore, we also conducted experiments on human chromosome 19 to determine if it provides better results due to the human genome is the 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. 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 n/r ratio of 222.236. This graph has 33,256 nodes and 166,104 edges. The total weight of the edge is 7,142,005,530. As shown in Table 3.4, 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 3.3: Human chromosome 19 dataset clustering results

Again, let's also virtualize the result using a scatter graph. 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

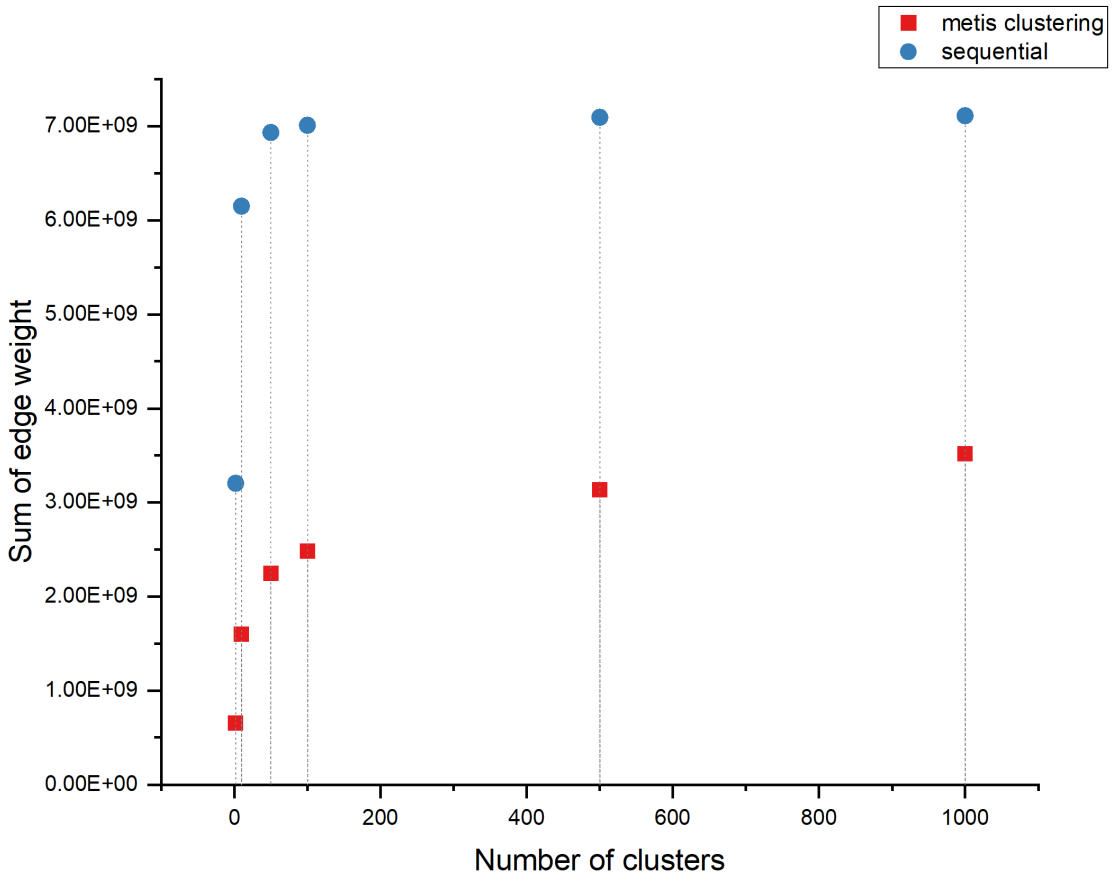

Figure 3.5: Human chromosome 19 weight change before and after METIS clustering.

the Salmonella dataset when the cluster number is over 100. However, the rate of increase in the weight of METIS clustering still 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, it was expected that the n/r ratio would increase by 2 times, and the weight of the inter-cluster edge would also increase by 2 times. However, dividing both weights by their number of copies showed that the two sets of weights of the inter-cluster edges almost draw the same line. Therefore, increasing the number of chromosome copies did not improve the results but instead increased the weight of each edge in the graph. The best result obtained from chromosome 19 was a decrease in the number of cache misses by a factor of 4 when the cluster number is 10.

3.2.3 Noisy string dataset

To investigate the relationship between the number of cache misses and the n/r ratio of the input sequence further, an experiment was conducted to cluster artificial sequences with different numbers of flipped characters (i.e., $A \longleftrightarrow G$ and $C \longleftrightarrow T$). 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. The sequence $noisy_0$ means that all characters were not flipped, $noisy_1$ means that every character was flipped, $noisy_{10}$ means that every 10 characters were flipped, and so on until $noisy_{10000}$. Fewer flipped characters mean longer runs, which results in a higher n/r ratio. Figure 3.6 shows the METIS clustering result of six input sequences. As shown in the figure, the number of cache misses increases as the n/r ratio grows. However, when the n/r ratio exceeds a certain threshold, the number of cache misses starts to plateau, indicating that the sequence is no longer consecutively competitive. Figure 3.6.

The results of the noisy strings experiment showed the opposite of the expected outcome, which was surprising. After discussing the matter with my supervisors, the conclusion was reached that flipped characters actually promote cache misses because they serve to divide runs. This means that what used to be a single run (i.e., a single row in the look-up table) now becomes multiple shorter runs, which results

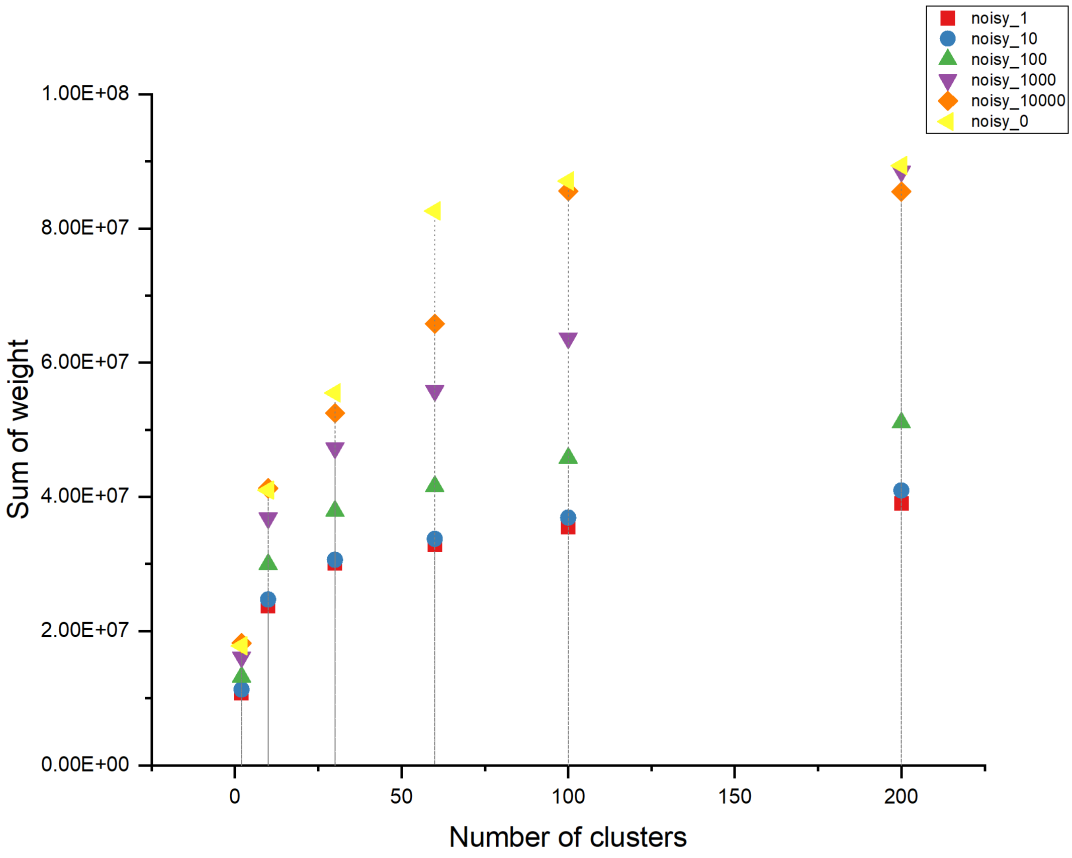

Figure 3.6: Noisy strings weight change before and after METIS clustering.

in more blocks. Furthermore, based on the results of the noisy strings experiment, it was determined that the BWT with run-length compression feature could also cause more cache misses compared to the uncompressed BWT. As a result, the unexpected outcomes of the noisy strings experiment have altered the future direction of the project to some extent.

3.3 Discussion of RLBWT experiment

There are three types of datasets that have been tested in this experiment, which are the Salmonella reference genome, 128 copies and 256 copies of the human chromosome 19, and the noisy strings with different numbers of flipped. Based on the results obtained from the Salmonella dataset and the human chromosome 19, we can say that METIS clustering provides significantly 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 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 issues when implementing the cache-friendly feature. This means that run-length compression may not be the best compression algorithm to apply on BWT when the optimization direction is focusing on memory locality.

In the next chapter, the focus of the experiment will shift to developing a cache-friendly layout using regular BWT instead of RLBWT. This experiment will be divided into two main aspects: firstly, building the graph and then optimizing the permutation of BWT blocks (i.e. the vertices in the graph that are built from the block BWT) using blocking, clustering, and other techniques. secondly, building a data structure with this permutation to enable a more cache-friendly BWT layout when performing the BWT reversions using LF mapping. Meanwhile, the results of

this experiment will be thoroughly discussed in the next chapter too.

Chapter 4

BWT reversion

4.1 BWT reversion methodology

Due to the result of the noisy string experiment, it was found that run-length compressing is not in favour of the objective. Hence, the second stage of the experiment was conducted on regular BWT. The methodology of this experiment was composed of four aspects. First, a graph for METIS clustering was built using a BWT that was built from scratch. Then, a data structure was built that supports access and rank queries using the block permutation from METIS clustering and its performance was checked by calculating the sum of the jump distance of the LF mapping steps and the sum of the weight of inter-cluster edges. Next, the permutation of blocks in each cluster was optimized by applying some algorithm such as SA. Finally, the BWT reversion was performed on different datasets to check if the data structure actually performs better than traditional LF mapping.

4.1.1 build the BWT and clustering it

The first dataset used for METIS clustering is hamlet from Shakespeare [33], where two versions of the text were found, one with 22.8 thousand characters and another one with 16.3 thousand characters. Initially, a BWT of hamlet generated by my colleague Nate Brown with a block size of 100 characters and a cluster size of 100 blocks is used to build the graph for METIS clustering. However, the result showed that the sum of the weight of inter-cluster edges after clustering was 15 times better than without clustering, indicating that the BWT itself was not reliable enough since the result was too good to be true.

Next, a new BWT was generated using a preliminary BWT generation code, built by Travis, with time complexity of $O(n^2)$. With that piece of code, an accurate BWT of the hamlet was generated. As a result, the clustered graph of the BWT of the short

version hamlet with 162,850 characters gives a sum of the edge weight of 75,116 with a block size of 100 characters and a cluster size of 100 blocks and the unblocked and unclustered give 162,848, which seems a legitimate decrease compared to the results from the previous RLBWT experiment. However, $O(n^2)$ is not fast enough for larger datasets like the human genome. Therefore, finding a more effective BWT calculation algorithm is essential. Travis suggests using the SDSL library, with which the BWT of a dataset can be constructed in $O(nlog_n)$ time using the compressed suffix array (CSA).

Except for constructing the BWT, the CSA serves as the basis for building the graph of METIS clustering. To build the graph, an alphabet of the dataset is first built by looping through the BWT of the dataset. Next, the LF positions are calculated using the inverse suffix array (ISA) that is built under the CSA of SDSL. Specifically, for a dataset with length n, the values of LF are computed as $LF[csa.isa[i]] = csa.isa[i-1]$ for $i = 1...n-1$, and $LF[csa.isa[0]] = csa.isa[n-1]$. After that, the LF positions are converted to block positions using the block size, which is $P[i] = LF[i]/BLOCK$, where $P[i]$ is the block position and $BLOCK$ is the block size. Meanwhile, the number of appearances of each block position $P[i]$ for all the blocks are recorded. The edges and vertices that form the graph for METIS clustering are then built using the same procedure that was applied during the RLBWT experiment.

Another parameter is introduced to represent the efficiency of BWT queries, which is the jump distance of the LF-mapping steps, i.e., the difference between the new position and the old position on the BWM of each LF jump. For the short version hamlet dataset, the sum of the jump distance of the LF steps before METIS clustering is 8,494,543,292, and after the METIS clustering, it is 4,271,734,940, which is about 1.99 times better than the unblocked and unclustered BWT. Hence, combining the improvement on the sum of the weight of the inter-cluster edges, we hypothetically have a (1.99+2.17)/2=2.08 times improvement from blocking and clustering the BWT for the reversion using LF mapping.

4.1.2 build the data structure

The weights of inter-cluster edges and jump distances of the LF steps are considered as parameters for estimating the final performance of the permuted BWT layout.

In order to observe the real BWT reversion performance after replacing the regular BWT layout with the cache-friendly layout, a data structure needs to be built to test the speed of backward searches. Since the focus is primarily on the performance of genome reversion, the dataset was changed to the human reference genome (GRCh38 version) [26]. The data structure is built to support *access* and *rank* queries on any random string, but mainly focusing on the BWT of genomes. To compare the data structure, a wavelet tree of the input dataset is built at the same time using SDSL, which also supports *access* and *rank* queries.

To illustrate the key feature of the data structure, a table is created where each entry of the table is considered as one BWT block of the input dataset. For each BWT block in the table, a rank header is attached that contains the ranks of each character in the alphabet of the dataset until the end of the last block. All the blocks of the BWT are permuted according to the permutation of BWT blocks from the previous METIS clustering. With the help of the table, *access* and *rank* queries on the BWT of the dataset can now be built. Furthermore, since finding out the position of a character inside the BWT block still requires a traverse, a decision is made to replace it with a wavelet tree built by SDSL, which supports *access* and *rank* queries more effectively.

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 R. First, the BWT block in which C is located is found by $C_B = R/B$, and the offset of C inside that BWT block is found by $C_O = R/B$. If C is located in the first block ($C_B = 0$), then the rank of C at R can be found by $T[P[C_B]].block.rank(C_O, C)$, where *rank* is calling the rank query of the wavelet from SDSL. If C is located in the other blocks ($C_B \neq 0$), then the rank of C at R can be found by $T[P[C_B]].second[A[R]] + T[P[C_B]].block.rank(C_O, R)$, where A is the alphabet of the BWT.

After running some test cases, it was discovered that the running time of the data structure was longer than expected. After reviewing the code, it was identified that

the slowest part was the building of the table of headers and the wavelet trees of BWT blocks. To improve the performance, the integer sequence wavelet tree was replaced with a byte sequence wavelet tree; moreover, it was further transformed into a Huffman-shaped tree for better data compression. After these modifications, the building time of the table reduced and became normally proportional to the size of the dataset. To verify the correctness of the *access* and *rank* queries of the data structure, an inversion of the input BWT was performed using the data structure and the wavelet tree of the BWT simultaneously. It was observed that both approaches reported the exact same inversion result, indicating that the data structure was functional. The next step is to optimize the permutation of BWT blocks to further improve the speed of BWT inversion using the data structure.

4.1.3 optimize the block permutation

Two aspects can be focused on for further optimizing the permutation of the BWT, which are the order of blocks inside a cluster and the order of the clusters. As the number of clusters is rather small, there is not much improvement that can be made in the order of the clusters. Thus, the focus is on optimizing the order of blocks within a cluster to see if it gives better performance.

The first approach that was tried is hierarchical clustering. For the short version of Hamlet, it can be hierarchically clustered seven times from 813 clusters to 10 clusters, where the total weight of inter-cluster edges decreases from 162,830 to 58,866. If the short version of Hamlet is directly clustered into 10 clusters, then the total weight of inter-cluster edges would be 72,127. Hierarchical clustering provides better results in terms of the sum of the weight of inter-cluster edges by 22.5%. However, when the dataset is larger, such as the human reference genome, then the hierarchical clustering procedure becomes complicated, as the graph needs to be rebuilt for each round of clustering and the improvement in performance is not enough, given the large dataset and relatively small block size.

The second approach tried is random permutations. This 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. 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 distance of LF mapping. The original sum of jump distances without any permutation of the long version of Hamlet is 417,406,828 when the inter-cluster jumps are filtered out. When using the same random permutation on all the clusters, the best result out of 50 random permutations is 417,327,446, which is just an improvement of 0.27%. To seek a better performance using random permutation, the best-performing permutation was also found for each individual cluster, and the sum of jump distance was calculated using these best-performed 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 improve performance, and more approaches need to be tried to find a better one.

The two methods previously tried were only preliminary approaches expected to give better BWT reversion results in terms of the sum of jump distances. Both approaches did yield improved results compared to using the direct clustering result as the order of BWT blocks, but the improvements were not significant. A more fundamental solution is needed to address the problem. The optimization of the BWT block order is evaluated based on the sum of jump distance of the LF steps and the sum of the weight of the inter-cluster edges. Dr. Norbert Zeh, my supervisor, suggested that there is a similar problem in the field of algorithms known as the MLA problem. In 2004, Safro et al. proposed a linear-time solution to this problem [31]. Norbert drew inspiration from this paper and helped me develop an ILP solution to further optimize the arrangement of the BWT blocks in each cluster. The objective function of the ILP is to minimize the product of the sum of the weight and the sum of the distance of the edges of the graph that was built from the clustered BWT. Therefore, the objective function can be formulated as follows:

$$Minimize \sum_{(u,v)\in E} w_{u,v} d_{u,v} \tag{4.1}$$

where 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). Next, A constraint is needed to calculate the distances between endpoints, which can be defined as the absolute value of the difference between two points. Hence, it can be formed as

$$d_{u,v} = |p_u - p_v| \quad \forall (u,v) \in E \tag{4.2}$$

However, this constraint is not yet linear, so it requires some transformations to convert the problem into a linear form. One way to achieve this is by splitting the two sides of the absolute value into two separate constraints. As a result, the previous constraint is transformed to

$$\begin{aligned} d_{u,v} &\geq p_u - p_v \quad \forall (u,v) \in E \\ d_{u,v} &\geq p_v - p_u \quad \forall (u,v) \in E \end{aligned} \tag{4.3}$$

Besides, we also need to translate the permutation points to vertices in the graph, so

$$p_v = \sum_{i=1}^{n} i x_{v,i} \quad \forall v \in V \tag{4.4}$$

For the vertices, we also need to make sure that they only appear once in the permutation, so

$$\begin{aligned} \sum_{i=1}^{n} x_{v,i} &= 1 \quad \forall v \in V \\ \sum_{v \in V} x_{v,i} &= 1 \quad \forall 1 \leq i \leq n \end{aligned} \tag{4.5}$$

In order to formulate this problem as an ILP, it is necessary to ensure that x can only take integer values, and since x represents the appearance of a vertex, it can only be either 0 or 1. Therefore, the binary variable $x_{v,i}$ is defined as 1 if vertex v appears at position i of the permutation and 0 otherwise. As a result, the following constraint is added to ensure that x is a binary variable:

$$x_{v,i} \in \{0,1\} \quad \forall (v,i) \in V \tag{4.6}$$

Overall, the ILP that we constructed for minimizing the sum of the weights and the

sum of the distances of inter-cluster edges is formed as following

$$
\begin{aligned}
& Minimize \sum_{(u,v)\in E} w_{u,v} d_{u,v} \\
s.t. \quad & d_{u,v} \geq p_u - p_v \quad \forall (u,v) \in E \\
& d_{u,v} \geq p_v - p_u \quad \forall (u,v) \in E \\
& p_v = \sum_{i=1}^{n} i x_{v,i} \quad \forall v \in V \\
& \sum_{i=1}^{n} x_{v,i} = 1 \quad \forall v \in V \\
& \sum_{v\in V} x_{v,i} = 1 \quad \forall 1 \leq i \leq n \\
& x_{v,i} \in \{0,1\} \quad \forall (v,i) \in V
\end{aligned}
\tag{4.7}
$$

This ILP is clearly too complex to solve by hand, particularly when dealing with a large genome dataset. Hence, Dr. Norbert Zeh suggested using software called Gurobi to help solve the ILP [29]. Initially, there was concern about obtaining a license for Gurobi, but it was later discovered that Gurobi Ltd. provides free licenses to academics. A license was obtained for one year and Gurobi optimizer was installed on the school server, Waverley. After proper installation and testing, the ILP was implemented in Python, as suggested by Gurobi, we then conducted multiple weight-distance optimization experiments, with different parameters, to determine the best BWT layout.

To optimize the block permutation in each cluster, the blocks are treated as the input of ILP cluster by cluster. The optimizer seeks the upper bounds and lower bounds of the different solutions to narrow down the gap between them in order to find the optimal solution. The first experiment was conducted with a cluster size of 96, resulting in a matrix of nearly 10,000 variables, which posed a significant challenge for the Gurobi optimizer. After running the optimizer for over 48 hours on the block permutation of size 96, the gap remained over 90%. Under these circumstances, Travis suggested decreasing the cluster size to 20 blocks, which would result in a matrix with only 400 variables. As a result, 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 indicates that it does provide a

better block permutation of the current cluster by decreasing the production of the weights and distances of the edges by a factor of 3. If all clusters are optimized in this manner, it should be possible to form a more LF-mapping efficient layout of the BWT, which should be able to reverse the BWT in 1/3 time compared to the original layout. However, to obtain better performance of the reversions and meaningful clustering, block sizes are typically small compared to the entire length of the BWT. Therefore, if there are only 20 blocks per cluster, there will be a large number of clusters, and optimizing all of them using Gurobi will still be challenging. Thus, reducing the running time of the optimization procedure with a reasonable cluster size is still necessary. Manipulating different parameters was the first approach attempted. Many parameters were manipulated from different angles, 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"[29]. By default, the Gurobi Mixed-Integer Programming (MIP) Solver focuses on "finding new feasible solutions" while "proving the current solution is optimal." However, users can customize the solver's behavior by setting the $MIPFocus$ parameter[29]. If $MIPFocus = 1$, the optimizer will focus on finding new feasible solutions. Conversely, setting $MIPFocus = 2$ causes the optimizer to prioritize providing the optimal solution to the problem [29]. 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 [29]. As this third scenario addresses the issues encountered in this study, $MIPFocus$ was set to 3, and the optimizer was rerun with a cluster size of 20 blocks. 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.

Regarding $NoRelHeurTime$, this parameter "limits the amount of time (in seconds) spent in the NoRel heuristic." [29], where NoRel stands for No Relaxation. 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. If the optimization is stopped at this

point, the block permutation results are the same as those obtained after running the optimizer for 50 minutes. This suggests that finding the optimal objective lower bounds is not necessary for this particular ILP problem. Therefore, the focus was shifted to optimizing the upper bounds for larger cluster sizes. However, it still takes around 7,500 seconds to obtain the optimal objective upper bound for a cluster size of 100 blocks. After examining the trend of the objective upper bounds, 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 tried on one million bases and ten million bases of the reference genome, with different combinations of block and cluster sizes. With a block size of 500 and a cluster size of 50,000, the Gurobi optimizer was able to optimize the sum of $W * D$ by a factor of 3 for each cluster for the one million base reference genome sequence.

Since the goal of optimizing the block permutation has shifted from finding the optimal solution to finding an acceptable solution, other permutation optimization techniques were considered. SA was suggested as a possible technique by Travis. SA is a probabilistic method for finding approximate global optima of a given function [23]. After reading Murillo et al.'s paper on using the SA algorithm for L1 and L2 Unidimensional Scaling, a simpler version of the 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 initiation function, objective function, and optimization function.

The initiation 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 edge weight and the absolute difference of the edge members of the start and end of the edge, returning the cost. The optimization function takes the objective function, the starting point, 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 elements in the solution and calculates the new cost. If the new cost

is better than the current cost, it updates the current solution and cost; otherwise, it does not. Meanwhile, it uses a probability function to decide whether to accept the new solution or not. Finally, it returns the best solution 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 2000,0000 iterations and 1200 temperatures. However, the weight only dropped from 176108 to 162746, which is only 8%. This reduction is insignificant compared to the results obtained using the Gurobi optimization. Therefore, the Gurobi MIP optimizer was chosen as the final block permutation optimizing strategy.

Since the blocks have been clustered and the order of blocks inside each cluster has been optimized, the next step is to reverse the BWT back to its original string using LF mapping and the optimized layout to evaluate its performance.

4.1.4 The final BWT reversion

Human Reference Genome reversion

The reversion performance test was conducted on a reference genome sequence consisting of one million bases, due to the limited number of variables that Gurobi can handle. Secifically, even 100 blocks per cluster produced 10,000 variables in the optimization matrix, and even optimizing 100 clusters took 20,000 seconds if the optimization time was set to 200 seconds. Therefore, the BWT of the entire reference genome sequence could be used as the dataset for reversions. The BWT of this sequence was blocked and then clustered using METIS. Next, the order of blocks in each cluster was permutated using the Gurobi MIP optimizer. The METIS clustering was hypothesized to reduce the weight and jump distances of edges by a factor of 2, and Gurobi optimization was expected to reduce the product of the weight and distance of edges by a factor of 3, which would hypothetically decrease the reversion time of the BWT by at least a factor of 2.

However, the reversion running time remained the same after reversing the BWT of the 1 million bases snippet of the reference genome sequence using the permutation generated after applying METIS clustering and Gurobi optimization. To ensure the

reliability of this observation, different parameters were used for the Gurobi optimization, but the reversion times did not show any noticeable difference. Therefore, it was concluded that although the Gurobi MIP optimizer decreases the $weight * distance$ of edges by a factor of 3, it does not give better performance in terms of reversion speed. Hence, Gurobi optimization was not included in future BWT reversion experiments, and only METIS clustering was used as an approach to optimize the order of blocks. On the bright side, the size of the BWT was no longer limited by the number of variables in the optimization matrix of the ILP or the number of clusters in the dataset.

The reversion of the BWT of the reference human genome sequence was performed using two different approaches: a single Huffman-shaped wavelet tree with Ramen Ramen Rao (RRR) compressing algorithm [30] and the data structure that uses a table of block Huffman-shaped wavelet trees with RRR compressing algorithm. Both approaches were able to correctly reverse the BWT to its original reference genome sequence, but the reversion time using the data structure was slower than using the single wavelet tree, which indicates that the reversion did not take advantage of the permuted BWT layout when the memory was sufficient.

Travis proposed a theory that the Waverley server, which has a total memory of 94.132G, could potentially store the single wavelet tree of the BWT of the reference genome entirely in memory because it is supposed to be only less than 1 GB, which is 1/4 of the size of the BWT, given that the length of the alphabet of the human reference genome is 4 and its size is 3.2GB. Therefore, he suggested that a larger dataset with a limited amount of memory would require the single wavelet tree to be stored on disk or at least part of it, leading to disk accesses that would significantly slow down the reversion process. In contrast, the data structure only has block wavelet trees, which means that only some of the block wavelet trees will be stored in the memory at once, and they are permuted based on METIS clustering, potentially resulting in fewer disk accesses and achieving better reversion speed.

Douglas Fir Reference Genome reversion

To enable convenient genome reversion with limited memory, the Douglas Fir reference genome with 15,703,424,632 bases is chosen as the new dataset, which requires

around 14.62 GB of storage [27]. As the Douglas Fir reference genome shares the same alphabet size as the human reference genome, the expected size of the single wavelet tree of its BWT is 14.62 GB/4=3.66 GB. Next, 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. However, there are other important processes that also require memory while performing BWT reversion. It is anticipated that BWT reversion will experience increased disk access when conducted with 3 GB of memory or less since the whole wavelet can fit into memory. Meanwhile, the data structure that employs block wavelet trees will demonstrate a speed-up in comparison to the single wavelet tree due to less disk access.

The first attempt to limit the memory usage of the reversion program on the Waverley server was to use the *ulimit* command, which provides resource control for particular processes on systems that allow it [21]. 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. The hard limit acts as a ceiling for the soft limit" [21]. Initially, the memory limit was set to 10GB using $-Sv10000000$, which was smaller than the BWT itself but larger than the single wavelet tree. However, the reversion time using the single wavelet tree remained the same, indicating that the revision did not have more disk accesses than before. Then the memory limit was changed to a hard limit of 10G using $-Hv10000000$, but the reversion time remained the same. Next, the memory usage was reduced to 3G, which was just smaller than the size of the single wavelet tree. However, the reversion program threw a "bad_alloc" instance and aborted for either a soft limit or a hard limit. After reporting this situation to Norbert, he suggested that this could be because the Waverley server has enough memory to buffer the reversion process by swapping out the current memory block, even if it does not have control of that piece of memory. Therefore, the *ulimit* command could not be used to limit the memory usage of the reversion program.

The second command that was tried on the Waverley server was $time - v$, which can count the number of major and minor page faults caused by the process [20]. However, it was not available on the Waverley server, so it had to be installed using

the source code. After downloading and installing time-1.9, an error was encountered when attempting to run the revision program with $time - v$. There was not enough time to investigate the causes, so the idea of running the reversions of BWT on Waverley servers was abandoned.

As the reversion experiments cannot be conducted on the Waverley server, an alternative solution was proposed to run the experiments on a virtual machine where the memory size can be limited during the installation. 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 3.2GHz, 32MiB L3 cache, 1TB Solid-state drive (SSD) SKHynix-HF001 and 32 GiB of DDR4 main memory. The memory size of the virtual machine was varied from 8GB to 1GB with a decrement of 1GB each time to confirm the hypothesis that the single wavelet tree of the BWT will have more disk accesses when the memory size is 3GB or smaller. The swap file size was set to 8GB. Since the BWT was already reversed through the single wavelet tree and blocked wavelet trees, which means the correctness of the backward searches using either the single wavelet tree or the data structure is certain, the substring extraction method was used to replace the BWT reversion. The length of the substring was set to 100,000 bases based on the virtual machine's performance. Both the single wavelet tree and blocked wavelet trees were used for substring extraction experiments with different memory sizes. The block size was set to 1,500,000 bases, and the blocks were clustered by METIS with a cluster size of 10.

As shown in Figure 4.1 , the extraction time using the single wavelet tree significantly increased from 4GB to 3GB of memory, which is consistent with the earlier proposed hypothesis that the 3GB memory would not be sufficient to store the entire single wavelet tree after subtracting the memory used by system-critical components. Therefore, when the memory size is 3GB or less, the system needed to access the disk more frequently to complete the extraction. The substring extraction time using the blocked wavelet trees shows a similar trend to the single wavelet tree, taking slightly longer to finish when the memory is between 5GB and 8GB. When the memory is equal to or less than 4GB, the blocked 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 speed-up is only about 5% when

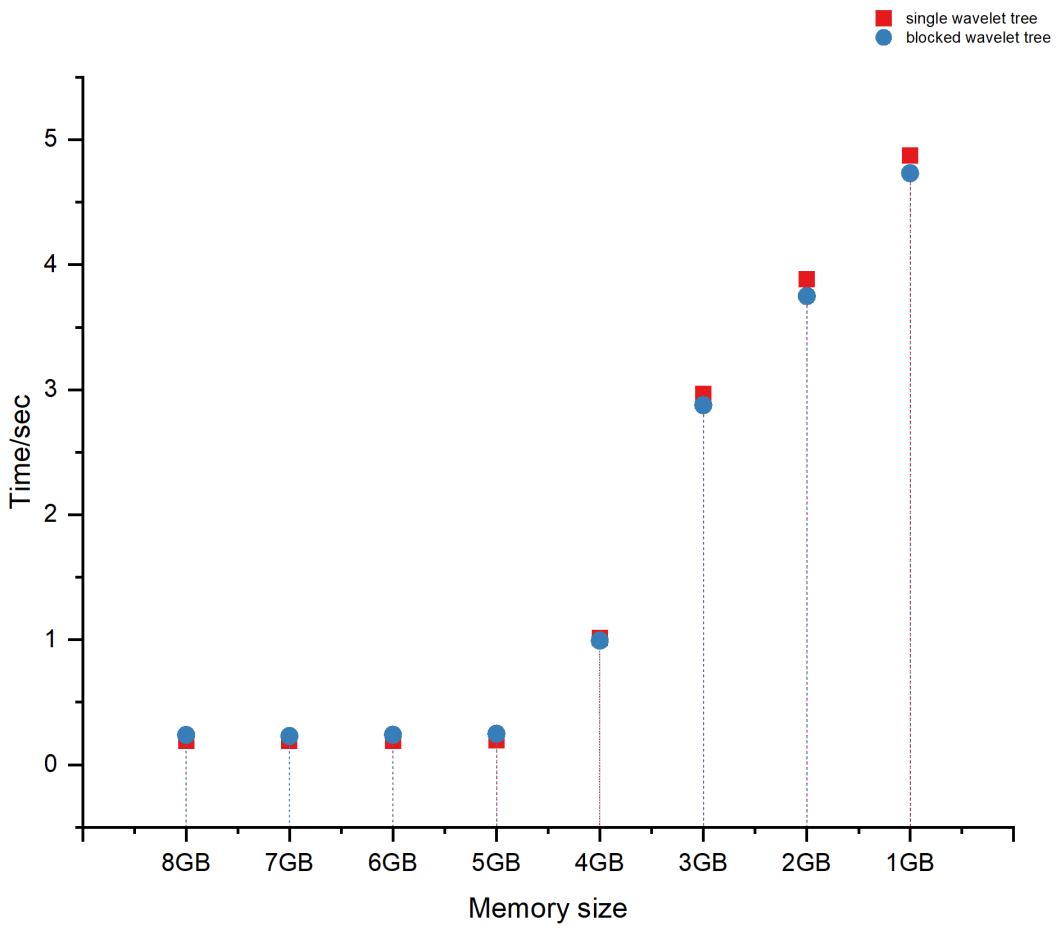

Figure 4.1: Substring extraction using the single wavelet tree and block wavelet trees with block size 1,500,000 bases.

the memory shrinks to 1GB, which reaches the minimum memory requirement of the OS. The preliminary substring extraction experiment shows that the blocked wavelet trees provide only a limited speed-up over the single wavelet tree. As the current block size and cluster size are random guesses, more different block and cluster sizes need to be tried to find the optimal combination.

4.2 BWT reversion results

Based on the preliminary substring extraction experiment, it has been confirmed that the blocked BWT with METIS clustering layout offers better memory locality compared to the traditional uncompressed BWT when the memory is limited, and the improvement grows when the memory size decreases. However, the improvement observed in the initial substring extractions was rather small, which calls for further improvement in the performance of the blocked wavelet tree to achieve a meaningful speed-up.

To obtain significant improvement, the virtual machine's memory size has been set to 1GB for all subsequent substring extractions. More block and cluster sizes will be explored to determine if they provide better performance. Additionally, a switch from solid-state drive (SSD) to a hard disk drive (HDD) is being tested to see if it leads to a more significant difference between the extraction time using the single wavelet tree and the blocked wavelet tree since the blocked wavelet tree produces fewer disk accesses, and each access using HDD would take longer than using SSD. Furthermore, since a disadvantage of generating blocked wavelet trees using SDSL was noticed, an alternative approach for generating blocked wavelet trees has been proposed by Travis, which is a program implemented by the author of SDSL Gog et al., and it is compatible with SDSL.

4.2.1 BWT substring extraction with SSD

In pursuit of optimizing block and cluster sizes for improved performance in substring extraction, the range of block sizes was extended from 1500 bases to 150,000,000 bases, while cluster sizes ranged from 1.46GB to 14.98MB 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 due to the fact that wavelet trees require a large amount of storage space when they are relatively small compared to the entire BWT, as each tree includes its own universal table, and the size of the universal table does not decrease as the wavelet tree size decreases. For instance, the 1500 bases wavelet tree divided the BWT of Douglas Fir into 10,468,950 blocks and occupied around 31GB of space, which is more than ten times the blocked wavelet tree with a block size of 1.5 million or more, and almost twice as large as the original Douglas Fir reference genome. 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 compiled in Figure 4.2, it can be observed that blocked 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 until the cluster size decreases to 299.52MB, after which it demonstrates a significant increase, contrary to the trend observed with 1,500,000 bases. For blocked wavelet trees with a block size of 150,000,000 bases, only a few clusters can be formed, each containing at least 299.52M bases due to the large block size. The only three scatter points obtained show a declining trend, consistent with the 1,500,000 bases wavelet tree.

Therefore, further substring extractions with different block sizes are necessary to determine the optimal one. Nonetheless, the 15 million bases blocked wavelet tree demonstrates an average improvement of 9.32% over the other two 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 blocked wavelet trees outperform the 150 million bases when the cluster is 1.46GB. Since larger block sizes like 150 million do not provide many options for cluster sizes, the substring extraction will be conducted with block sizes ranging from 1.5 million to 15 million bases.

Since the focus is now on block sizes between 1.5 million and 15 million, new

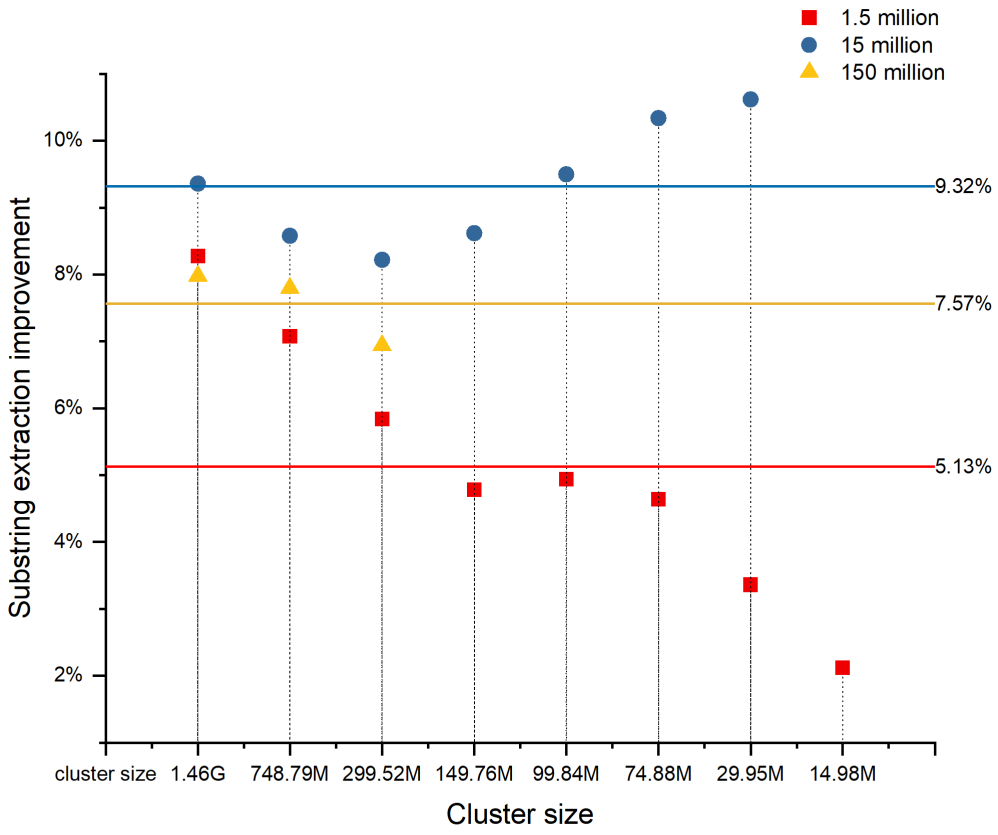

Figure 4.2: Substring extraction using blocked wavelet trees of size 1.5 million to 150 million with different cluster sizes.

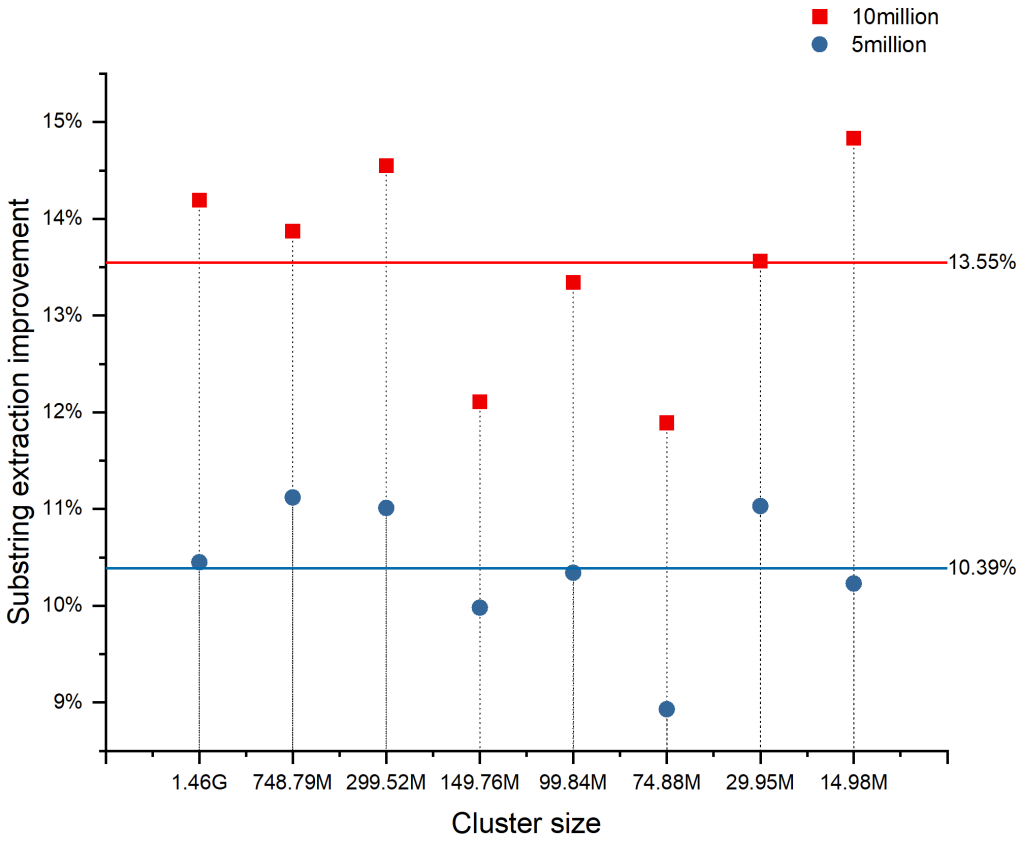

Figure 4.3: Substring extraction using blocked wavelet trees of size 5 million and 10 million with different cluster sizes.

test points are selected with block sizes of 5 million and 10 million, while keeping the cluster sizes the same. As seen from the Figure 4.3, Both the 5 million and 10 million block sizes offer better performance improvements compared to the previously investigated block sizes, with an average improvement of 13.55% and 10.39%, respectively. However, the improvements do not demonstrate a clear pattern of changes in terms of cluster sizes. As the 10 million bases block size wavelet trees give the best performance improvement that was found, further substring extractions will be conducted on block sizes between 10 million and 5 million. Despite the improvement in substring extractions using blocked wavelet trees, 13% is still not a notably high percentage.

Furthermore, it was realized that the Lenovo Legion 5 does not include an HDD, only an SSD for disk equipment. It is believed that the blocked wavelet tree will provide an even better improvement over the single wavelet tree if substring extraction

is conducted on an HDD because each disk access takes a longer time on an HDD. Therefore, an old laptop with an HDD was borrowed from my partner for further substring extraction experiments.

4.2.2 BWT substring extraction with HDD

The new virtual machine host is a Dell Inspiron 14-7472 with an Intel i7-8550U processor, consisting of 8 cores running at 1.80GHz and 16MiB L3 cache, with 8 GiB of DDR3 main memory. The disk capacity is 1TB ST1000LM035-1RK172. The virtual machine still uses the same operating system and memory size of 1GB. However, the size of the swap file has been reduced to 8GB, as the table of the blocked wavelet tree requires less than 10GB to build. Additionally, the substring size has been reduced to 5000 bases due to the Dell laptop's less powerful performance. Since it is unclear whether the block size affects performance on the HDD laptop to the same extent as it does on the SSD laptop, substring extraction needs to be re-conducted on the tested block size. However, block sizes greater than 15 million are 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 are still not considered for HDD substring extractions as they contain too many universal tables, which leads to space explosion. The first two block sizes tested with the HDD are 1.5 million and 15 million, as they are the smallest and largest block sizes that are currently being investigated. It is necessary to ensure that the HDD offers better improvements over SSD in terms of substring extraction. Moreover, as none of the extractions on SSD produced a clear pattern on the cluster sizes, the number of cluster sizes for the HDD extractions is being reduced.

As shown in Figure 4.4, the results show that the HDD offers much better performance improvement compared to SSD. The use of the blocked wavelet tree now yields an improvement of around 30% to 40%, with a maximum of over 45% for the 15 million base block size. In terms of performance difference between the 1.5 million block size and the 15 million block size, the 15 million block size wavelet tree still offers a better improvement with average improvements of 43.04% and 30.48%, respectively. The graph now includes the standard deviation, which shows that the performance improvements are relatively stable at 40% and 30% for the 15 million block size and

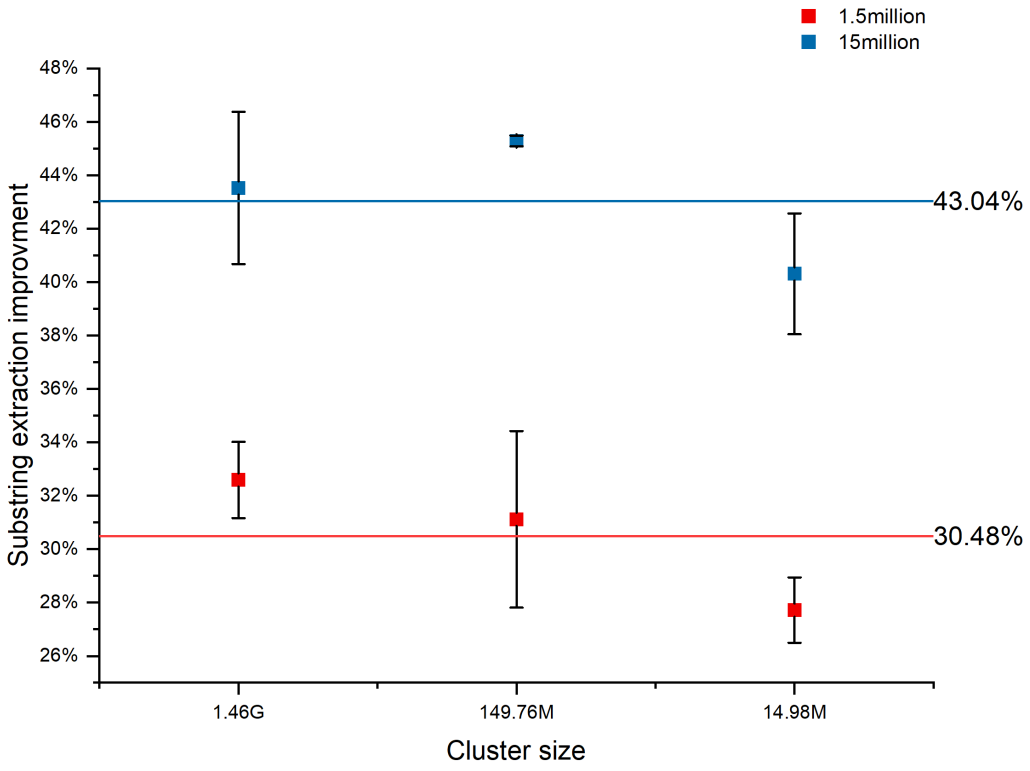

Figure 4.4: Substring extraction using blocked wavelet trees of size 1.5 million and 15 million with different cluster sizes on HDD.

1.5 million block size, respectively. However, there is still no clear pattern on 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 will be tested to find the optimal substring extraction enhancement from applying the blocked wavelet tree.

Since no clear pattern has emerged in finding a better cluster size for substring extraction, whether the METIS clustering was actually improving performance or if the improvement was solely due to blocking the BWT becomes questionable. To address this question, it has been decided to compare the trends of performance improvement with different block sizes, but the same cluster size, to determine if the performance improvements demonstrated a pattern with respect to cluster sizes. Table 4.1 lists the new block sizes that will be tested, covering the majority of block sizes between 1.5 million and 15 million by increments of one million. If no similar pattern of improvement changes is 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 for substring extraction.

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

The Figure 4.5 depicts all the substring extraction experiments that were conducted on block sizes between 1.5 million and 15 million. As observed from the plots, 6 out of the 11 plots show a single peak at either cluster size 149.70MB or 29.95MB, while the remaining 5 plots show the lowest data point at either cluster size 149.70MB 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 is observed among the different cluster sizes in terms of the speed-up for substring extraction, raising doubts about whether the METIS clustering contributes to the efficiency of substring extraction. As the cluster size does 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

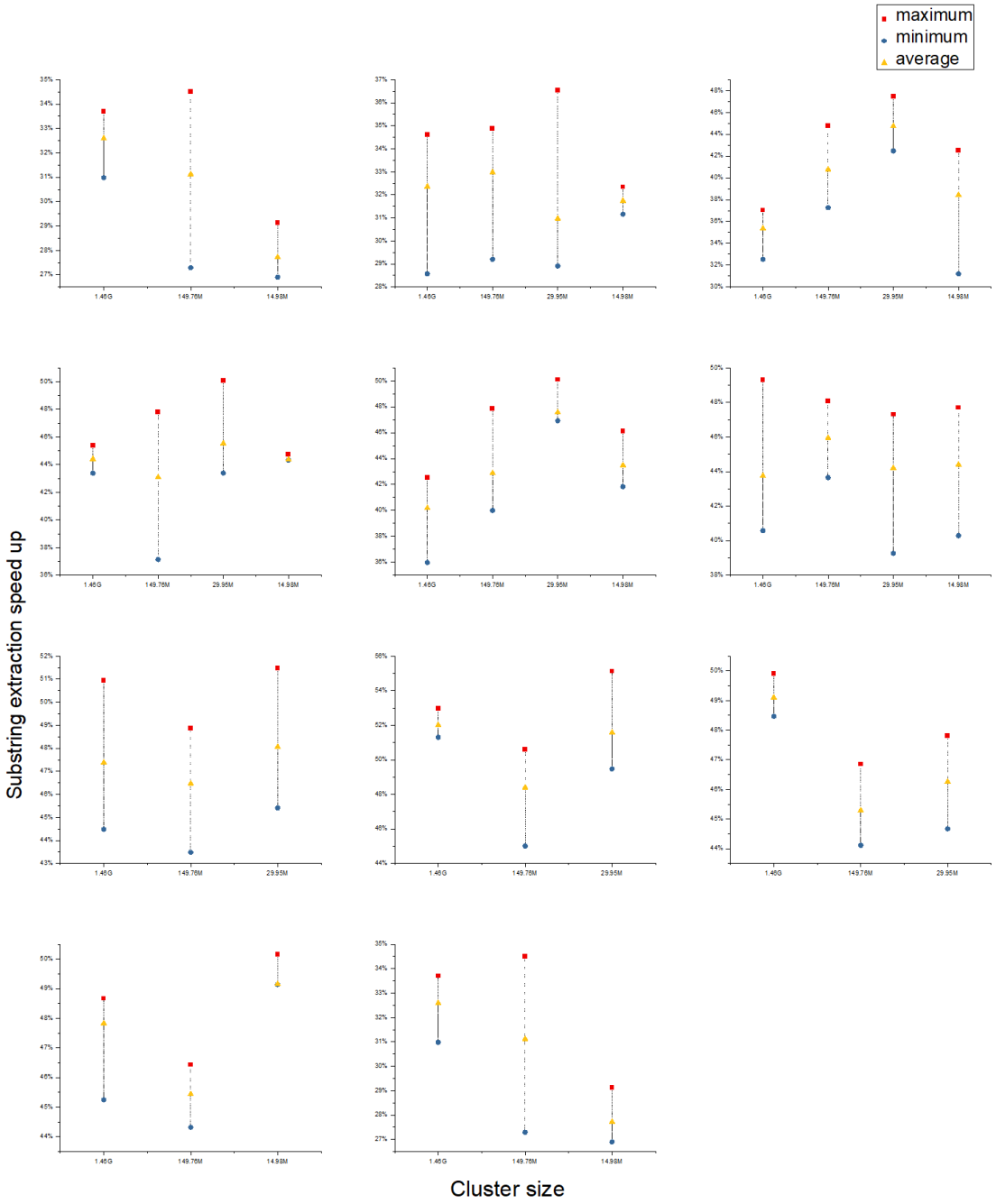

Figure 4.5: Substring extraction using more different block sizes wavelet tree between 1.5 million and 15 million.

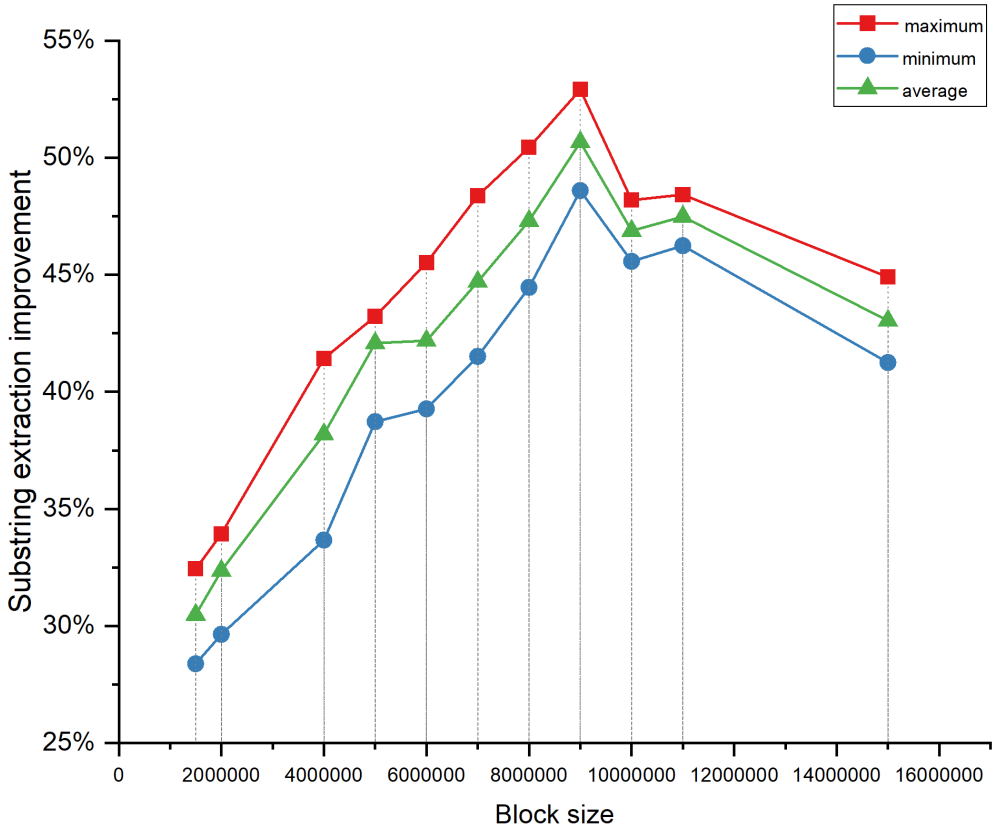

Figure 4.6: The overall performances of substring extraction between 1.5 million to 15 million block sizes

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 4.6 the results indicate that the performance of substring extraction using different block sizes demonstrates a clear trend along with the increase of block size. Specifically, there is a significant improvement of 20% in performance from the block size of 1.5 million bases to 9 million bases, and the peak improvement of 50% is achieved at the block size of 9 million. However, after the peak, the improvement gradually decreases with a similar ratio to the increase. 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.

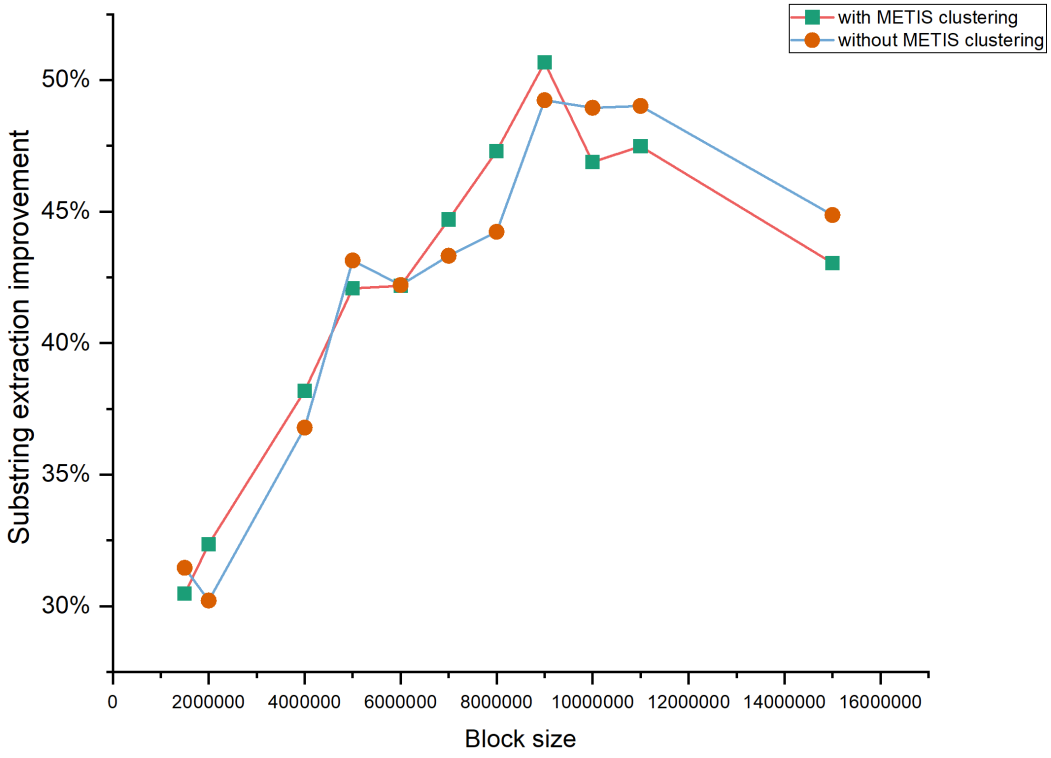

Figure 4.7: Comparing substring extraction with and without METIS clustering.

To investigate whether the METIS cluster offers any benefits when it comes to substring extraction speed, more substring extractions were performed with only blocking the uncompressed BWT and no block permutation from the METIS clustering. 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 4.7, indicates that the METIS clustering does not offer any improvement for the substring extractions on top of the improvement gained from blocking the uncompressed BWT. This is in line with the previous conjecture drawn from Figure 4.5. Therefore, the substring extractions gain a 40% to 50% speed-up just by blocking the uncompressed BWT, which is not consistent with the previous conjecture of how the layout of the BWT would affect the speed of backward stepping of the BWT using LF, but it is still a noticeable increment. Moreover, the optimal block size is actually much bigger than anticipated, and the non-sharing universal table issue of

the blocked wavelet tree prevents the exploration of smaller block sizes. Therefore, an alternative approach is 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) has been proposed by Travis.

4.2.3 BWT substring extraction with Fixed block compression boosting

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 reversion experiments [10]. 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. Furthermore, since the FBB wavelet tree is already compressed, RRR compression is not applied to it again. Its performance is 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. All substring extractions are performed on the HDD laptop Dell Inspiron 14-7472 with a substring length of 5000 bases, 1GB of memory, and a swap file of 8GB. Moreover, since it was shown that the METIS clustering does not improve substring extraction speed, only blocking is applied to the block wavelet tree 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 of the block size base 2, and the logarithm of the superblock size base 2. For the initial tests, all four parameters are set to default values, where the block size is 2^{12} and the superblock size is 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} [10], and since the Douglas Fir dataset is larger than 2^{32}, this technique is used for the experiments. As shown in Figure 4.8, the slowest substring extractions are observed

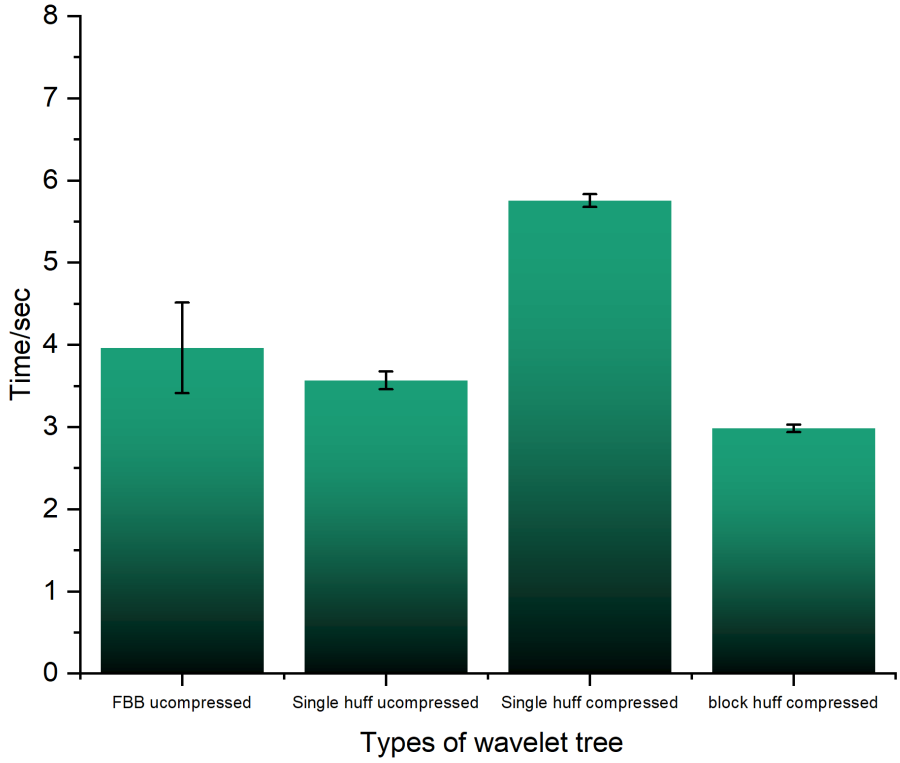

Figure 4.8: Substring extractions using FBB wavelet trees and previous wavelet trees

using the Huffman-shaped compressed single wavelet tree, which was expected, as it yields an average extraction time of 5.754 seconds for 5000 bases. Surprisingly, the second slowest performances are from the FBB wavelet trees, which demonstrate an average extraction time of 3.963 seconds. However, since the block and superblock sizes are default numbers and the standard deviation of the substring extraction time is noticeably larger than other types of wavelet trees, it is expected that the FBB wavelet trees will 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.

Since the FBB wavelet tree did not demonstrate better performance than the block wavelet trees with default settings, the next round of the experiment involves testing more block and superblock sizes. According to Gog et al. [10], 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 expectations of errors or exceptions when building the FBB wavelet trees for 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. On the other hand, extractions were completed without any error or exceptions from the system when the block size exceeded 2^{16}. The remaining block and superblock combinations that were tested are listed in Table 4.2, and their performance results are recorded in Figure 4.9.

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

Table 4.2: New block sizes for the 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 extraction time of 3.544 seconds, which is faster than the uncompressed Huffman-shaped single wavelet tree that previously outperformed the FBB wavelet trees. The

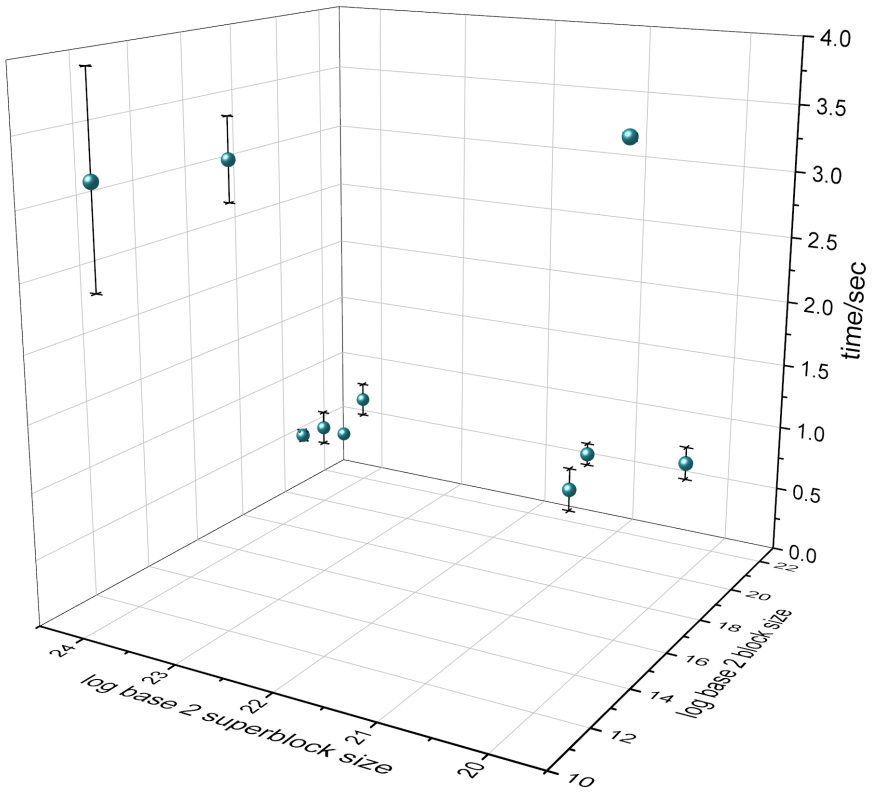

Figure 4.9: Substring extractions using FBB wavelet trees with different block and superblock size combinations

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, the correctness of the extractions using FBB wavelet trees is not checked, and such rapid extraction speed is questionable. 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 is applicable for the Douglas Fir dataset. The most rapid extraction using FBB wavelet trees is still 3.090 seconds, which is slower than the block wavelet trees of 2.984 seconds. Moreover, the most rapid FBB substring extraction uses the block size 2^{16} and the superblock size 2^{24}, which are the maximum size 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 is the best result that can be produced with the FBB wavelet trees for 5000 bases, considering the small differences between different runs of the extractions.

Until this point, all the experiments for optimizing the layout of BWT are finished, and the results will be furtherly discussed 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 memory locality over the sequential layout, similar to the RLBWT experiment. The approach taken to achieve this objective was to compare the reversion time 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 reversion 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 reversion experiment was the hamlet dataset, and the same techniques used in the RLBWT experiment, such as blocking the BWT

and clustering the blocks using the METIS cluster, were applied. The primary reason for starting with this dataset is to check the correctness of the data structure during the construction because hamlet is relatively small. Although no reversions were conducted on the hamlet dataset, blocking and clustering the BWT still resulted in better results in terms of the sum of edge weights and the sum of LF mapping jump distances. Specifically, 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 reversion would also decrease by a factor of 2, and an even better result could be achieved if the block orders within each cluster were optimized.

After a solid data structure was built for the BWT reversion, the focus of the experiment shifted to optimizing the block orders within each cluster. Two techniques were used: reducing the product of all the edge weights and distances by solving an ILP problem using Gurobi, and reducing the sum of edge weights using the SA algorithm. Both techniques showed better results compared to sequential block orders, with the ILP giving a more significant result, a decrease of a factor of 3 in the product of edge weights and distances. Therefore, the reversion tests on the human reference genome were conducted with Gurobi-optimized block orders in each cluster, using the blocking and clustering techniques applied to hamlet. Due to difficulties in solving an ILP with over 10,000 variables, the reversion was conducted on a snippet of the human reference genome with 1 million bases.

However, the reversion time using Gurobi-optimized block orders remained the same as with sequential block orders, which led to the conclusion that the Gurobi optimization did not offer a better reversion speed of the BWT, despite effectively decreasing the product of the edge weights and distances. This was contrary to the expectations from optimizing the block orders within each cluster. It could be due to the need for further constraints on the Gurobi optimizer to perform better, or the algorithmic power and capacity of the current Gurobi were not strong enough to provide satisfactory results. Either way, the Gurobi optimizer with the current setting was not helpful for achieving a better layout of the BWT and was suspended for future reversion tests. However, it is still believed that fully optimized block orders within each cluster would provide a better reversion time of the BWT.

Due to the suspension of Gurobi, the length of the reversion dataset was no longer limited. Therefore, the whole human reference genome BWT was used as the dataset for the reversions. It was also discovered that larger datasets and smaller memory setups are required to demonstrate the advantages of block wavelet trees, which represent the optimized BWT layout over the single wavelet tree of the entire BWT that represents the sequential BWT layout. After trying a few different options, the Douglas Fir reference genome was chosen as the dataset, and a virtual machine with different memory settings was used for future tests. As the correctness of the reversion data structure has been proven, and it takes a huge amount of time to perform reversion on an entire BWT, substring extractions were used to replace the reversions. 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 performance improvement 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 with greater specificity using the same range of block and cluster sizes. As expected, the performance improvement of the block wavelet trees was found to be greater on the virtual machine with HDD. The most significant improvement of 50% occurred with a block size of 9 million bases across all cluster sizes.

However, different cluster sizes did not exhibit a uniform trend in terms of performance improvement 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 expected decrease in the number of cache misses or the incorrect ratio of decrease, such as assuming that the number of cache misses is decreased by the decline of the sum of edge weight with a ratio of 1. It could also be that the graph built from the blocked 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 is clear that applying block wavelet trees improve performance by simply blocking the BWT. However, the block sizes that were tested did not cover smaller block sizes of less than 1 million bases. This was due to the issue that the total size of wavelet trees would explode when the block size is relatively too small compared to the length of BWT because they are not able to share the universal tables. To address this issue, the author of SDSL Gog et al. suggested using the FBB technique to build the block wavelet trees. The FBB wavelet trees have two parameters: block size and superblock size, which have maximum limitations of 2^{16} and 2^{20} respectively. The dataset will also be divided into hyperblocks if its length exceeds 2^{32}. After trying different combinations of block and superblock sizes, the performance improvement of the FBB wavelet tree also grew with the increase of block and superblock size. The best result 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.

Based on the extraction times collected from the FBB wavelet trees, it is 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 size, FBB wavelet trees cannot 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 outperforms or equals 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 performance improvement than the block wavelet trees constructed by the data structure.

Chapter 5

Conclusion

The project was divided into two major experiments. The results of the experiments revealed that the optimized BWT layout was 50% better than the traditional BWT layout in terms of the time of BWT reversions.

In the RLBWT experiment, it was observed that the run-length compression feature did not contribute to providing a better memory locality. However, after applying METIS clustering, the BWT layout could offer a notably smaller sum of edge weights compared to the sequential order of the BWT. Hence, the optimization of the layout was performed on the uncompressed BWT using METIS clustering. Furthermore, additional techniques were applied to optimize the block order within each cluster, including solving an ILP inspired by the MLA problem using Gurobi and using the SA algorithm to find the global optimum. Both techniques provided a better number in terms of the performance estimation parameters compared to the sequential orders of blocks in clusters, but Gurobi provided more significant improvement than the SA algorithm. A program was also implemented to apply the optimized BWT layout using wavelet trees built from each block of the BWT and a look-up table to track the position of each block in the optimized layout. Additionally, the program included a single wavelet tree built from the entire BWT of the dataset for comparison with the block wavelet trees.

The reversion tests were conducted on virtual machines with limited amounts of memory, and the virtual machines were hosted on two laptops, one with an SSD disk and the other with an HDD disk. The block wavelet trees showed a shorter reversion time compared to the single wavelet tree on both virtual machines. This indicates that the optimized BWT layout is causing fewer cache misses compared to the sequential layout, making it a more memory-friendly layout. A performance improvement of 14% was observed on the SSD laptop, while the HDD laptop showed a 50% improvement, with similar block sizes and setups. These results are consistent

with expectations since accessing an SSD takes much longer than accessing an HDD. However, the analysis showed that neither the Gurobi optimization nor the METIS clustering provided any help in reducing the reversion time of the BWT. Hence, the performance improvements were solely achieved by blocking 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, the FBB wavelet tree was introduced. It almost provided the same amount of improvement as the blocked 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 outperform the block wavelet trees eventually.

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 reversion time. Furthermore, the Gurobi optimization was preset to abort and show the current result if the process could not finish within a reasonable time and was never finished. Therefore, it is believed that there is still much further optimization that can be done on the block order within each cluster or even the order of the clusters. Besides, due to the limited time for conducting experiments, there is still some range of block sizes that has not been tested yet. 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. These two problems of non-shareable universal tables and limited block sizes may also limit the performance improvement of the block wavelet trees and FBB wavelet trees. Specifically, if the block wavelet tree can be built with a shared universal table, then its overhead would be reduced dramatically, which would lead to an expectation of an even more rapid reversion 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 100 thousand bases or bigger, it would provide even more rapid backward searches.

This thesis has demonstrated that blocking a BWT and constructing a lookup

table based on the blocks can lead to a better memory locality of the BWT. However, there is still room for improvement in the current performance of BWT backward searches. 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 performance improvement of the reversion speed gained from blocking the BWT. In addition to the improvement gained from blocking 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 memory 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 a good alternative to replace the reversion time for evaluating whether a layout offers better memory locality or not.

The data structure built in this thesis, which employs BWT blocking and the look-up table, can perform BWT reversion 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, Oct 2021.

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

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

[4] R.R. Coifman and M.V. Wickerhauser. Entropy-based algorithms for best basis selection. *IEEE Transactions on Information Theory*, 38(2):713–718, 1992.

[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] 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.

[10] 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.

[11] 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.

[12] Michael Held and Richard M. Karp. A dynamic programming approach to sequencing problems. *Proceedings of the 1961 16th ACM national meeting on -*, 1961.

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

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

[15] 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.

[16] 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.

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

[18] J. B. Kruskal. Multidimensional scaling by optimizing goodness of fit to a nonmetric hypothesis. *Psychometrika*, 29(1):1–27, 1964.

[19] Ben Langmead, Cole Trapnell, Mihai Pop, and Steven L Salzberg. Ultrafast and memory-efficient alignment of short dna sequences to the human genome. *Genome Biology*, 10(3), 2009.

[20] Linux. time(1) — linux manual page.

[21] Linux. Ulimit.

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

[23] 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.

[24] Gonzalo Navarro. A guided tour to approximate string matching. *ACM Computing Surveys*, 33(1):31–88, 2001.

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

[26] NCBI. Grch38.p14 - hg38 - genome - assembly - ncbi.

[27] 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.

[28] Takaaki Nishimoto and Yasuo Tabei. Optimal-time queries on bwt-runs compressed indexes, 2020.

[29] Gurobi Optimization. Gurobi optimizer, Nov 2022.

[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 and Harold Jenkins. *Hamlet*. Thomson Learning, 2005.

[34] 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.

[35] Jouni Siren. Burrows-wheeler transform for terabases. *2016 Data Compression Conference (DCC)*, 2016.

[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] Yu Su, Yi Wang, and Gagan Agrawal. In-situ bitmaps generation and efficient data analysis based on bitmaps. *Proceedings of the 24th International Symposium on High-Performance Parallel and Distributed Computing*, 2015.

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

[40] Huanchen Zhang, Hyeontaek Lim, Viktor Leis, David G. Andersen, Michael Kaminsky, Kimberly Keeton, and Andrew Pavlo. Surf. *Proceedings of the 2018 International Conference on Management of Data*, 2018.