CSCI 6057 Advanced Data Structures – Project Report

Yansong Li

Faculty of Computer Science – Dalhousie University

Section 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), just the human chromosome 19 has almost 59 million bases [1], but its alphabet is just four characters, which are A, T, C, and G. Burrows-Wheeler Transform 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 Burrows-Wheeler Matrix (BWM); then the last column of the matrix is the BWT. And run-length compressed BWT is a BWT that is compressed using run-length encoding, where a run means a sequence of characters that has the same data value occurring in consecutive order [3].

The BWT and RLBWT can be reversed to the original input string using a key property called Last-to-First (L-F) Mapping, which is the i^{th} occurrence of a character c in last column of BWM and the i^{th} occurrence of the same character c in First column correspond to the same occurrence in input string [4]. However, reversing BWT and RLBWT to their original string is practically slow even though the algorithm itself is fast due to the randomness of the jumps of L-F mapping. As a result, the randomness of L-F mapping led to an extremely high number of cache misses, which almost every jump will cause a cache miss during the reversion.

The current best result of computing L-F mapping of the RLBWT is developed by Nishimoto and Tabei, who achieved to compute the L-F mapping in constant time using a look-up table with $O(r)$ (r is the number of runs in RLBWT) space [3]. However, due to the large size of dataset in bioinformatics, the $O(r)$ space look-up table still need to be stored in the memory, which means Nishimoto's look-up table does not give a better memory locality than the traditional data structure of L-F mapping of RLBWT; hence, it still causes an enormous number of cache misses. Jouni Sirén proposed an approach in 2020 by rearranging BWT to achieve a better memory locality for the graph extension of the positional BWT [5]. Therefore, trying to see whether applying Sirén's algorithm about BWT rearrangement on Nishimoto's look-up table can give a better memory locality of RLBWT reversion while maintaining the constant running time is the major task in this project. For acknowledgment, the idea of combining Nishimoto's data structure and Sirén's algorithm is from my supervisor Dr. Travis Gagie, but all the work of proving this idea is feasible is completed by myself.

For the rest of this report, Nishimoto's and Sirén's papers will be further analyzed and summarized in section 2. Next, I will specifically describe how I combined Sirén's algorithm with Nishimoto's data structure in section 3. Then, I will present the results of the project using different datasets in section 4. Finally, discussions and conclusions will be stated based on the results of the project and the direction of future research will be indicated in section 5.

Section 2: Literature Review

In this section, I illustrate how Nishimoto's look-up table works and its improvement compared to previous implementations for RLBWT reversion. Also, I give a more detailed description about Sirén's algorithm.

Section 2.1 : Nishimoto's data structure

In order to store an arbitrary permutation P n string $S = \{0, \dots, n-1\}$ with a property that the sequence $P0), P(1), P(2), \dots, P(n-1)$ is composed of a limited number b of unbroken incrementing subsequences, $O(b)$ space compressed spare 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 L-F mapping from $O(\log\log n)$ to $O(1)$ and keep the same space cost of $O(r)$, [3]. Specifically, if we use an array T to represent a 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 corresponds to it, which is a quadruple that consisted of p; the length of the subsequence $|p|$; the interval $P(p)$; and the offset of $P(p)$ [2].

The reversion of RLBWT is achieved by jumping forth and back in each row of the look-up table starting from the row of the end mark \$ [3]. For each row T_i, its index is the starting position of the L-F 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 [3]. There exist a special case of L-F jumping, which is crossing the boundary. When the offset is larger than the length of the subsequence $|p|$, then we cross the boundaries of the rows to find the ending position of the current jump rather than just jumping to the row specified by the interval $P(p)$ [2]. However, Brown et al. have proved that this boundary-crossing situation rarely happens.

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

Section 2.2 : Sirén's algorithm

Sirén improves the memory locality of graph extension of the positional BWT in his Graph BWT implementation by reducing the number of cache misses. Based on pBWT and its graph extension, The GBWT is implemented as an 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 $\sum v = \{w \in V | (v, w) \in E)$. \$ 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 [5]. A BWT is partitioned to substrings BWT_v corresponding to the prefixes end with v, and each BWT_v is encoded using $\sum v$. Let $\sum v\,(w) = k$ when $w \in \sum v$ is the k^{th} character in $\sum v$ in sorted order [5]. As shown in Figure 1, there is a record corresponding to each $v \in V$ and end-mark \$, which is consisted 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. Meanwhile, a cache-friendly graph layout is assumed, such as $v \in V$ have a low out-degree, and the graph is almost linear and topologically sorted, which are all reasonable assumptions for a large set of biological haplotype sequences over a variation graph [5].

The GBWT construction algorithm is based on a dynamic FM index and uses the BCR algorithm to insert multiple texts into the index in a single batch. When the construction using a single batch is too slow, the dataset will be partitioned into super-batches and separate GBWT indexes will be built for each super batch [5]. Next, the indexes will be merged using the BWT-merge algorithm, and the merge algorithm can also be reversed for removing texts from the index [7]. After that, the haplotype information in the GBWT can be used to simplify the VG graph for k-mer indexing with a series of pruning heuristics, which means the k-mers from the recombinations will be pruned and k-mers from the haplotypes will be kept in the index [5].

Section 3: Methodology

Our primary objective of this project is trying to see if we apply Sirén's algorithm to Nishimoto's data structure will reduce the number of cache misses when we reverse the RLBWT to the original string using L-F mapping. To state the methodology in one sentence, we first cut the RLBWT table into blocks; then we build a graph using the blocks; finally, we cluster the graph and calculate the sum of the weight of the inter-cluster edges, where the sum of weight represents the total number of cache misses caused by L-F mapping. We assume the graph that we build is cache-friendly, which means each node will have a low out-degree and the graph has a low expansion just like Sirén's assumptions.

Section 3.1 : Preliminary Graph Building and Clustering

Before we experiment on real genome datasets, we use a small string example to virtualize the feasibility of implementing the cache-friendly feature onto RLWBT. The following lookup table derived from string "bananaband$" in Table 3.1.1 is collected from Nate Brown in WCTA 2021 [7], who implemented the Nishimoto's look-up 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 3.1.1 RLBWT Look-up table of "bananaband$" [7]

First, the table is cut into four blocks with a size of 2 rows per block. Next, we use the blocks as nodes and the sums of lengths of the inter-block jump as wedges to build a graph as shown in Figure 3.1.1.

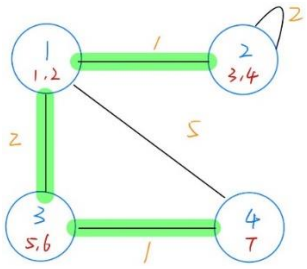

Figure 3.1.1 Graph of "bananaband$"

Once we have built the graph, we first ignore the self-connected edges in each block since they are meaningless in the implementation; then we cluster the graph using an algorithm that is inspired by Sirén's method of rearranging BWT, which is that we choose the most weighted edges that between two un-clustered nodes and cluster them into the same group and repeat such a process until all the blocks are clustered, where all the clusters share similar sizes. The result of clustering the graph of "bananaband$" is shown in Figure 3.1.2.

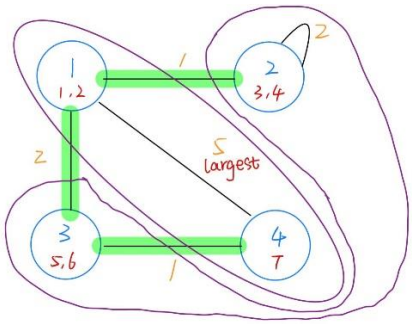

Figure 3.1.2 Clustered graph of "bananaband$"

As we can see from Figure 3.1.2, the most weighted edge between block 1 and block 4 is deleted, and all the other three edges are kept. Also, block 1 and 4 are clustered in the same group and block 2 and 3 are clustered in the same group. As a result, the total weight of the inter-cluster edges in this graph is 4.

Next, we cluster the blocks sequentially (a simulation of the current processing approach of L-F mapping between memory and CPU) and compute the total weight of inter-cluster edges. As we can see from Figure 3.1.3, the sum of the inter-cluster edges increases to 7.

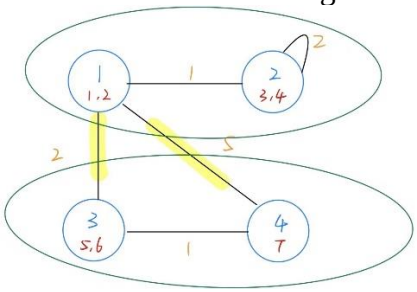

Figure 3.1.3 Sequential clustered graph of "bananaband$"

Based on the result from this small dataset, we can preliminarily determine that our idea of implementing the cache-friendly feature onto RLWBT would work by rearranging the look-up table. Besides, we also use the real genome datasets to check the feasibility of our idea by building graphs using code and applying a graph clustering application called METIS.

Section 3.2 : Real genome graph building and METIS Clustering

The RLBWT tables built from real genomes are also provided by my colleague Nate Brown. After we read the look-up table from a binary file, we follow the procedure that I mentioned in preliminary clustering to cut it into blocks but based on the L1 cache size, for this project I used 1024 rows, which is small enough to be stored in common L1 cache 64K. Then we build the graph using the same approach as we mentioned in the preliminary experiment. If we state it more formally, the blocks are used as vertices $v \in V$ and jumps of L-F mapping that across different blocks are used as edges $e \in E$ to build a graph $G = (V, E)$. Each e in G is weighted and the weight $w \in W$ of e is determined by the total length of runs between two vertices $(v, u) \in V$. The graph constructions for genome datasets are implemented using C++.

After we have generated the graph using the RLBWT table, we will cluster it using METIS with various cluster numbers to see the growth of the number of cache misses. The clustering algorithm applied on METIS is also the same algorithm that we described in preliminary experiment, which is finding $e = (v, u)$ with the largest w then we put v and u into the same cluster and keep clustering e with next the largest w until reaching the size of a cluster. When we have partitioned all the blocks into clusters, we calculate the total weight W_r of inter-clusters edge $e_t \in E_t$.

As a comparison against our METIS clustering, we also cluster the graph G sequentially with the same cluster numbers that we used for METIS. Since we are clustering graph sequentially, we just cluster it by its index in the table. Again, all the clusters share similar sizes. Then, we calculate the total weight of the inter-cluster edges of sequential clustering W_s. If $W_r < W_s$ significantly, then it shows that Sirén et al.'s method does decrease the number of cache misses of the RLBWT table in Nishimoto et al.'s construction.

Section 3.3 : Cache-friendly layout Proofing

The feasibility of implementing cache-friendly feature onto RLWBT is based on the assumption that the graph we built from Nishimoto's look-up table has a cache-friendly layout, and one of the essential aspects is each node has a low out-degree. Nate Brown proves that the occurrence of boundary crossing is low in practice for pangenomes [8], which indicates each node in the graph has a low degree because our 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: 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 in the graph built from Nishimoto's look-up table is at most about $d + \sigma$, where σ is the size of the alphabet.

Proofing: 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 $\frac{b}{\sigma}$, where σ is the size of the alphabet. Suppose we also pick a block B at random. If p_c is the run to which the start of the first run of c's in B maps (according to the 3rd column of Nate'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 $q_c - p_c <= \frac{db}{\sigma}$, where d is the average number of runs in the L column spanned by each run in the F column. Nate's experiments show d is small in practice.

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

We also need to prove that the graph has a low expansion, but due to the limited time of this project and the change of research direction because of some latter results, the proofing has been postponed, but I will work it out shortly. On the other hand, Sirén et al. state that the cache-friendly assumption is reasonable for large biological haplotype sequences [5].

Section 4: Results

The results of clustering real genome datasets are present in this section, which consists of two types of genome datasets, a look-up table derived from salmonella and a look-up table derived from human chromosome 19. Besides, an artificial dataset with different among of flipped character is also included to determine the relation between $\frac{n}{r}$ ratio and decrease of cache misses.

Section 4.1 : Salmonella dataset

The first experiment uses an RLBWT table that was built from the salmonella genome sequence, and the statistics of this dataset are listed as follows. The length of the sequence is 145,595,456, and the number of runs is 12,823,516, which gives us an $\frac{n}{r}$ ratio of 11. The number of nodes in the graph is 12,523, and the number of edges is 62,556. The total weight of the edges is 145,589,783. The METIS clustering and sequential clustering results are listed in Table 4.1.1. We can see that the weight of inter-cluster edges of METIS is reduced by a factor of 3 when the cluster number is 10 compares to sequential clustering. The difference is rather small when the cluster number is 1000

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

Table 4.1.1 Salmonella dataset clustering results

In addition, the line graph is used to virtualize the difference between MEITS clustering and sequential clustering. As shown in Figure 4.1.1, until 100 clusters there is a clear difference in weights between METIS clustering and sequential cluster. The METIS is still 2 times smaller than sequential clustering when the cluster number is 100. Once the cluster number is over 100, the weight difference shrinks significantly.

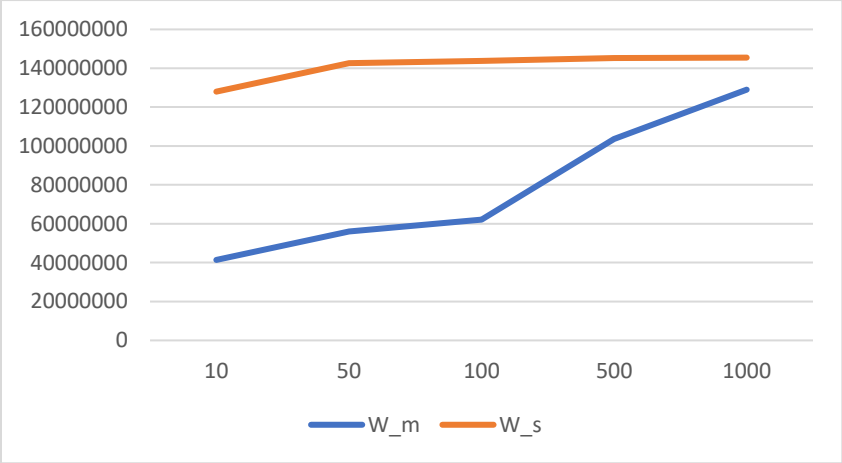

Figure 4.1.1 Growth trends of salmonella dataset

As a result, the salmonella dataset shows a decrease in cache misses by a factor of 3 when the cluster number is 10. Since $\frac{n}{r}$ ratio is not significantly large in this salmonella data, which means the input string is not super consecutively competitive. Therefore, we also experimented on human chromosome 19 to see if it provides better results; also, the human genome is the primary research target in bioinformatics.

Section 4.2 : 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 $\frac{n}{r}$ 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 4.2.1, 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 exist 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
$\frac{W_s}{W_m}$	3.84	3.09	2.82	2.26	2.02

Table 4.2.1 Human chromosome 19 dataset clustering results

Again, let's also virtualize the result using a line graph. This time, we can see there is a bigger difference between METIS and sequential clustering, and the weight of METIS clustering does

not grow as fast as the salmonella data set after the cluster number is over 100. However, the growing rate of METIS clustering still increases after the cluster number is over 100, and it is almost 3 times smaller than sequential clustering when the cluster number is exact 100.

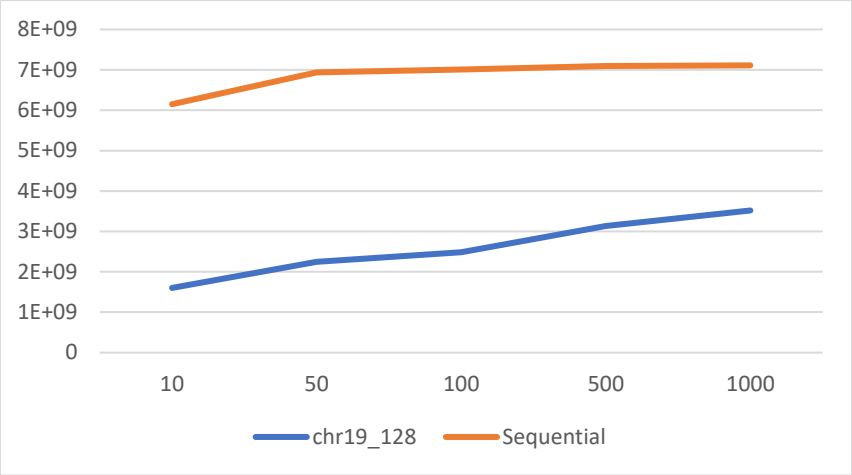

Figure 4.2.1 Growth trends of human chromosome 19 dataset

I also used 256 copies of chromosome 19 to conduct the experiment because the n/r ratio will be even higher and the other statistics will be similar. As we expected, the n/r ratio increases by 2 times, but the weight of the inter-cluster edge increases by 2 times as well, and if we divide both weights by their number of copies, then the two sets of weights of the inter-cluster edges almost draw the same line. Therefore, increasing the number of copies of chromosomes won't give us better results, it will only increase the weight of each edge in the graph. Hence, the best result that is derived from the chromosome 19 is decreasing the number of cache misses by a factor of 4 when the cluster number is 10.

Section 4.3 : Noisy string experiment

To furtherly investigate the relationship between the number of cache misses and the $\frac{n}{r}$ ratio of the input sequence, I also conduce an experiment of clustering artificial sequences with a different number of flipped characters (i.e., A↔G and C↔T). Specifically, the alphabet of the sequence is still [A, T, C, G], and it has a total length of 100,000,000 characters consisting of 100 copies of a subsequence of the length of 1,000,000 characters. Sequence noisy_0 means all the characters are not flipped, sequence noisy_1 means every character is flipped, sequence noisy_10 means every 10 characters is flipped and so on until noisy_10000; also, fewer flipped characters mean longer runs, which gives higher $\frac{n}{r}$ ratio. The METIS clustering result of 6 input sequences is showed in Figure 4.3.1.

Surprisingly, the result shows exactly the opposite of our expectations. After a discussion with my supervisors, we conclude that flipped characters are in favour of the cache misses because it helps to break up the runs, so what was in a single run (i.e., a single row in the look-up table) turns into several shorter runs, which makes more blocks. Therefore, the run-length compression of BWT is

the problem. Such an unexpected result of the noisy strings experiment changes the future direction of the project to some extent.

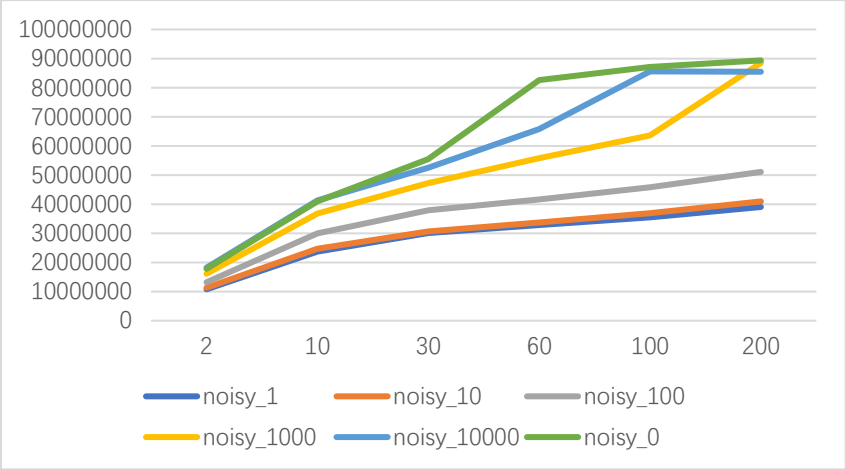

Figure 4.3.1 Noisy strings METIS clustering

Section 5: Discussions and Conclusion

Based on the result that we got from the 2 types of datasets, we can say that METIS clustering gives meaningfully better results when the number of clusters is less than 100. Under the best scenario, the salmonella shows 2 times better, and chromosome 19 shows 3 times better with 100 clusters. 1024 block size and 100 clusters, which is 332 blocks per cluster are feasible based on the L1 and L3 cache size of Waverley and Timberlea (i.e., two Dal severs), which are 64K and 16M respectively. On the other hand, the noisy string experiment reveals that the run-length compression on BWT could cause issues when we implement the cache-friendly feature.

In conclusion this experiment, we can say that the clustering results of genome sequences are significantly better, we manage to reduce the number of caches misses by a factor of 4 when there are 10 clusters in the dataset of chr19. Therefore, the idea of rearranging BWT as blocks to reduce the number of cache misses is feasible. However, the run-length compression may not be the best compression algorithm that can be applied on BWT with the cache-friendly feature.

For future works, we still need to prove the graph has a low expansion. Meanwhile, we also need to optimize the block size and cache size to parameters, so the sizes can adapt based on the CPU. Then we will first work on applying the idea of rearranging BWT as blocks on normal BWT without any compression algorithm, which means the input sequence will just be single genomes. However, the ultimate objective of this project is still to develop a cache-friendly data structure for pangenomes; hence a compression algorithm is still necessary for the future when we developed a cache-friendly feature on normal BWT for single genomes.

References

[1] NCBI. (2022). Genome Reference Consortium Human Build 38 patch release 14 (GRCh38.p14). https://www.ncbi.nlm.nih.gov/assembly/GCA_000001405.29

[2] Brown, N.K., Gagie, T., & Rossi, M. (2021). RLBWT Tricks. ArXiv, abs/2112.04271.

[3] Nishimoto, T., & Tabei, Y. (2021). Optimal-Time Queries on BWT-Runs Compressed Indexes. *ICALP*.

[4] Langmead, B. (2021). Burrows-Wheeler Transform. *Teaching Materials.* https://langmead-lab.org/teaching-materials/

[5] Sirén, J., Garrison, E., Novak, A., Paten, E., & Durbin, R. (2020). Haplotype-aware graph indexes. *Bioinformatics,* Volume 36, Issue 2, pp. 400–407.

[6] Karypis, G., Kumar,V. (1999). A Fast and Highly Quality Multilevel Scheme for Partitioning Irregular Graphs. *SIAM Journal on Scientific Computing*, Vol. 20, No. 1, pp. 359—392.

[7] Sirén, J. (2016) Burrows-Wheeler transform for terabases. *In: Proceedings of DCC 2016,* IEEE, Snowbird, UT, pp. 211–220.

[8] Brown, N.K. (2021). Interval Mapping of BWT-runs to Efficiently Compute LF-mapping in $O(r)$ Space. *WCTA. https://www.cristal.univ-lille.fr/spire2021/wcta/slides/brown_IntervalMappingBWT-runsToComputeLF_WCTA.pdf*