Bioinformatics
doi.10.1093/bioinformatics/xxxxxx
Advance Access Publication Date: Day Month Year
Manuscript Category

Structural bioinformatics

Benchmarking Learned Index Data Structure for the FM-index over the Compacted dBG.

Yiheng Xu [1] and Le Chang [1]

[1] Department of Computer Science, University of Maryland, College Park, 20742, USA

Abstract

Recently, many data structures are designed for sequence analysis and indexing, such as the suffix array, the FM-index, and the de Bruijn graph. While most of them focus on indexing the full sequence, de Bruijn graph represents a sequence in terms of its k-mers. It is efficient and can be constructed on different lengths of sequences. Based on the de Bruijn graph, some data structures like Pufferfish (Almodaresi *et al.*, 2018) and dbgfm (Rayan and Limasset *et al.*, 2015) are developed. These de-Bruijn-graph-based data structures use contigs to store the graph information. In order to find the index of the input query from the contigs, these data structures apply different algorithms. In this report, we proposed a method that tries to replace those construction algorithms with a learned index data structure, LISA (Ho *et al.*, 2019), so that the cost of runtime and memory can be more balanced.
Availability: https://github.com/HenryYihengXu/858D-project
Contact: yhxu@umd.edu, lchang21@umd.edu

1 Introduction

Genome is the typical identification information for a species. A typical genome is a long sequence composed of four characters. The length of the complete genome of a unicellular organism can be greater than 4 million. The difficulty of finding a specific subsequence in a genome string is unignorable, especially when the target subsequence and the reference genome string are very large. For the past decades, methods like Suffix Array, the FM-index, and de Bruijn Graph, have been proposed to efficiently store the relative information about the genome sequence.

Compared to other approaches that analyze a reference sequence based on the full text, de Bruijn Graph can represent an arbitrary sequence through the k-length unique subsequences (called kmer) of the reference. This data structure is efficient and it is the basis of many sequence assembly. Every node of de Bruijn graph is a kmer. A data structure, unitigs array, is a sequence that contains the union of all kmers. Tools like Pufferfish (Almodaresi *et al.*, 2018) and dbgfm (Rayan and Limasset *et al.*, 2015) use the unitigs array for storing the nodes' information of a generated de Bruijn graph. With the unitigs array, it is easy to check if a kmer is in a graph and to lookup for its neighbors. To find if a kmer exists in a graph, these tools adopt different algorithms such as FM-index and minimal perfect hash function (MPHF). These algorithms show a good performance but they also has some limitation. Querying in a FM-index could be slow and difficult and MPHF needs to save the hash table and position vector for querying. In this project, we tried to replace these algorithms with a novel learned index method in order to discover a better trade-off between the runtime and memory space. Our inital idea is to replace FM-index in dbgfm with a learned FM-index model to see if it can speed up querying. After exploring the dbgfm code, we found that it uses FM-index to check if a kmer exists but it doesn't have any solution on finding the exact position of this kmer in the reference sequence. Therefore, we chose Pufferfish to see if a learning index model requires less space than a MPHF.

After reading several papers, we found that most learning index models are developed for suffix array. Because suffix array costs more spaces than FM-index and more searching time than MPHF, it is not suitable for our project. We finally decided to use a learning model named LiSA. This is a learning model that is designed based on a FM-index like data structure. Unlike FM-index which can only lookup one letter at a time, this learning model can find multiple characters at once and return the explicit position of the query in the input string. Before trying to combine LiSA with Pufferfish and compare the runtime and space, we first tested some genomes for benchmarks. From the benchmark, we found that LiSA costs both more time and space than the original method, therefore, we believe that MPHF has better performance than LiSA.

We make the following contributions in this work:

- Propose an idea of applying learned FM-Index to compacted dBG
- Fix some bugs in LISA code and modify it to return positions in the original string. Identify key problems of LISA and make suggestions.
- Conduct a comprehensive benchmarking and analysis of LISA. From that we report how different parameters impact its time and space performance and the optimal parameters, which are missing in original LISA paper.

- Benchmarking performance of LISA and corresponding parts in Pufferfish with human chromosomes. Evaluate the performance loss and thus conclude it's not worthy to apply current LISA implementation to Pufferfish.

The structure of this report is: section 2 introduces some background of de Bruijn graph (dbgfm, pufferfish) and learned index (LiSA); section 3 describes our method on how we tried to benchmark and combine learned index and pufferfish ; Some test results are shown in section 4, this section discusses the results and shows why we think it's not a good idea to replace MPHF with LiSA; section 5 concludes our project and discusses the future work.

2 Background

In this section we provide background and relevant literature to compacted dBG and learned FM-Index. In particular, we introduce Pufferfish, a state-of-art compacted dBG implementation, and LISA (Ho *et al.*, 2019), probably the only existing learned FM-Index. A detailed literature survey document is included in the GitHub repo.

2.1 compaced dBG

2.1.1 de Bruijn graph

de Bruijn graph is a directed graph that can represent arbitrary sequences in terms of the k-mer of the references. It can be used to compare the differences between two references by coloring the nodes and to represent an unknown length of string.

When building a de Bruijn graph, a set of unique subsequences with the length of k will first be found from the reference sequence. These k-length subsequences are the nodes of the graph. Then it adds a directed edge between two nodes if these two k-mers occur next to each other in the reference. The direction of edge is from the node at the front position to the node that occurs after it.

In a built de Bruijn graph, there could be some pairs of nodes that only connected to each other and has no other branching. A compacted de Bruijn graph replaces all pairs of nodes from a normal de Bruijn graph that has this property with a single edge. In a compacted de Bruijn graph, a node can not just represent a single kmer but also possibly to represent a union sequence of those pairs of nodes. This compaction process can significantly increase the speed of alignment for a genome that has many repeated subsequences.

2.1.2 Tools with dBG

There are many tools that are built upon de Bruijn graph. Because this project's initial motivation was to replace FM-index with a learned index over the compacted de Bruijn graph, we looked at several papers. In (Beller and Ohlebusch, 2016), they proposed several algorithms to construct a de Bruijn graph using FM-index. In other words, this algorithm builds de Bruijn graph where each node is not a kmer but is the FM-index information of this kmer relative to the reference genome so that it requires less space. Since this algorithm requires explicit FM-index data to build the graph, it is hard to combine it with a learned index.

Other data structure we explored includes dbgfm (Rayan and Limasset *et al.*, 2015), which also represents de Bruijn graph in low memory. Dbgfm first constructs the contig of a de Bruijn graph, then uses non-lexicographic minimizers to enumerate the maximal simple paths. Finally, dbgfm uses FM-index to save those paths and check if a kmer exists in the reference and who are its neighbors. This idea is efficient but one limitation is that it cannot points the original position of a kmer in the reference string. Because of those limitations, we decided to choose Pufferfish as our project tool.

2.1.3 Pufferfish

Pufferfish (Almodaresi *et al.*, 2018) is a novel data structure for colored compacted de Bruijn graph. It takes input from compacted de Bruijn graph and then build the sparse/dense pufferfish index. This index is composed of six components: a unipath array (unitigs array, useq), a boundary vector that is corresponding to the useq, an MPHF that maps kmers to a unique number, a position vector that stores the position of kmers in useq, a unipath table that stores the reference string of a unipath, a edge table that stores the information of each edge and an equivalence class table. When querying through the pufferfish index, it will look up into the MPHF to find the index in the position vector then find its position in the useq and boundary vector and finally finds it position in the reference string. Because MPHF is a hash function, it can look up a kmer in constant time. It is very fast while it requires more space to save the hash table and position vector. Therefore, we proposed this idea to replace it with a learned index model to see if it can save some space while only increasing a little amount of time.

2.2 Learned FM-Index

2.2.1 FM-Index

FM-Index (Ferragina and Manzini, 2000) is a full-text index essentially based on Burrows-Wheeler Transform (BWT). BWT (Burrows and Wheeler, 1994) takes all circular permutations of the original text and sorts them, producing a matrix called BWM. The last column of BWM is the output of BWT. Because of the nice LF mapping property, one can reconstruct the original text and query the text by backward search. With some well designed structures such as sparse suffix array, tally, etc., it can achieve log(m) query time where m is the query length, and linear space w.r.t the text length.

2.2.2 Learned Index

Learned Index essentially means augmenting classic index structures with machine learning models. We have surveyed several papers about learned index and found most of them use machine learning models to predict a smaller initial search range. For example, Sapling (Kirsche *et al.*, 2020) applies a fully connected neural network and a piece-wise linear architecture to suffix array. Given a query, it predicts a much smaller interval on the suffix array for the binary search. BWA-MEME (Jung and Han *et al.*, 2022) uses a similar idea on suffix array. It further optimizes the last-mile search and generalizes the model to work for query with variable length.

As predicting a smaller search interval sounds promising and FM-Index searches in intervals as well, our initial attempt was to apply this idea directly to FM-Index. However, a key difference between querying FM-Index and suffix array is it's not doing binary search. Although the search interval is getting smaller, it has to go through m backward searches. Then it's meaningless to predict a smaller initial interval. Moreover, the predicted interval cannot be the same as the actual interval in each step, so we still have to do all the work. Therefore, we gave up coming up with our own structure and decided to adopt the following work.

2.2.3 LISA: Learned FM-Index

LISA (Ho *et al.*, 2019) applies learned index specifically for FM-Index, and it is probably the only existing learned FM-Index implementation. Compared with regular FM-Index where it backward search one character by one character, the key idea of LISA is to backward search k characters at a time supported by a special data structure called Index-Paired BWT (IP-BWT). However, a new issue is it cannot efficiently locate those k characters in the first several columns of BWT. The regular FM-Index can do this in constant time because there are only four different characters.

But in LISA when k gets large, it has to do binary search to locate them, and that is where the machine learning model comes in.

To speed up the binary search in each backward search step, LISA uses the similar idea as other learned indices. It uses Recursive Model Index (RMI) to predict a much smaller search interval given those k characters. Hence, LISA is able to reduce the number of backward search steps by a factor of k, but in each step, it adds an overhead of $log(predicted\_interval\_lenght) + prediction\_time$. Nevertheless, LISA reported to achieve 1.3 - 2.2X higher throughput than Trans-Omics Acceleration Library (TAL), which is a baseline FM-Index implementation.

However, a big problem of LISA is that its evaluation is far from comprehensive. First of all, it didn't show the impact of k and the number of RMI leaf nodes, which should be important factors for both time and space. More importantly, it didn't evaluate the space at all. It turns out that LISA is actually taking a lot more space than one would expect. In addition, LISA's code has several bugs that makes it doesn't work as it is described in the paper. We will discuss these problems in detail and provide fixes in later sections.

3 Methods

In this section, we introduce our methodology of benchmarking learned FM-Index and compacted dBG. We show how we evaluate the feasibility and performance benefits of potentially applying LISA to Pufferfish.

3.1 Overview

The key insight for introducing LISA into Pufferfish is that the MPHF lookup and the position vector lookup in Pufferfish can be substituted by an FM-Index. Figure 1 shows the structure of Pufferfish. It takes in a kmer and go through a bunch of lookups and finally returns the positions of this kmer in original references. A principal component is to find the kmer position in the unitig sequence (useq) by looking up the MPHF and the position vector. The highlighted part is essentially a query to useq and it is exactly something we can replace with an FM-Index.

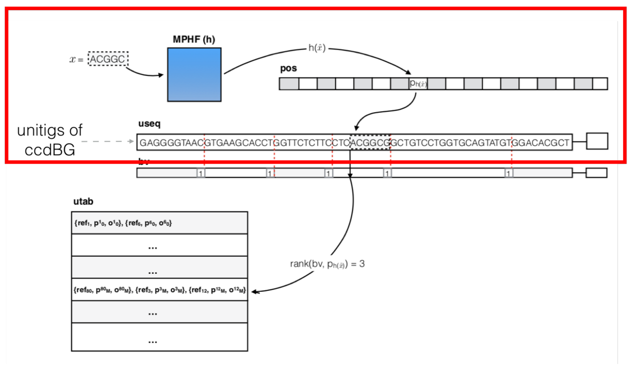
MPHF (h)
x = ACGGC
$h(\hat{x})$
pos
$p_{h(\hat{x})}$
unitigs of ccdBG
useq
GAGGGGTAACGTGAAGCACCTGGTTCTCTTCCTCACGGCGGCTGTCCTGGTGCAGTATGTGGACACGCT
bv
1
utab
{ref1, p10, o10}, {ref6, p60, o60}
...
rank(bv, $p_{h(\hat{x})}$) = 3
{ref80, p80M, o80M}, {ref3, p3M, o3M}, {ref12, p12M, o12M}

Fig. 1: Puffish data structure and the compenents that can be replaced by an FM-Index

Based on this insight, our method is to combine ideas of both work. But due to time limit, we didn't actually combine them into a single tool. Instead, we decided to first benchmark the time and space of the these components. Figure 2 shows the space distribution of Pufferfish. We can see MPHF and position vector (highlighted) take most of the space. If we replace this with a learned FM-Index, we can possibly achieve a much better space usage. However, this is based on a big prerequisite that LISA must have small space usage. As mentioned previously, LISA paper didn't report space at all. So it's important to first benchmark and evaluate both tool and see if it's worthwhile to do the substitution. Therefore, our work primarily focuses on studying both code, fixing bugs, making modifications so they would work together, and benchmarking them.

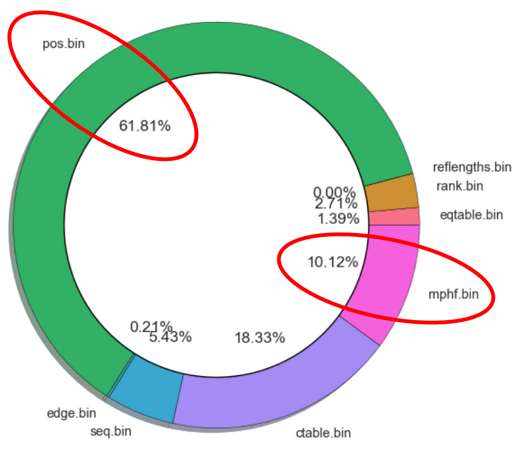

Fig. 2: Space distribution of Pufferfish components

3.2 Studying Pufferfish and LISA code

We studied the code of Pufferfish and LISA to locate those components mentioned above in the code. We studied their structures and formats, input output interfaces, saved files, etc. For Pufferfish, we built it without any trouble. We found where the unitig sequence (useq) is built, saved, loaded. In particular, we identified the code region where Pufferfish query useq with supports of MPHF and position vector. Although we don't actually combine them into a single tool, this is useful for our benchmarking because we now know the timing region. Moreover, we checked the format of useq and made sure the comparison between it and LISA would be fair. In addition, if future we want to actually combine the code, this will be a good starting work.

For LISA, we located where the FM-Index and the ML model are built, saved, loaded, and used in querying. We located where it goes through all queries, does backward search, and returns results. That again tells us the timing region and modifications we need to make. In contrast to Pufferfish, LISA does requires a lot more work in compiling, fixing bugs, and make modifications. We will discuss this in the next subsection. A detailed code reading note is included in the GitHub repo.

3.3 Debugging LISA

The first issue we encountered is LISA only compiles on machines with Intel AVX512 instruction supports, which took us some time to get access to such a machine and get familiar with the environments. Besides, its dependencies are built separately. For example, the machine learning module is downloaded and compiled at runtime by evoking some scripts. That makes it slightly harder to locate compilation glitches. We have fixed a missing file issue in its Makefile and a missing flag issue in one of the scripts.

Furthermore, we have found some logical bugs when testing LISA code. We started by running the code with a very short test sequence. The code breaks down at processing queries with OpenMP or ends up with infinite loop, since the parallel batch size processing queries becomes 0 when the reference sequence is very short. This is a minor issue and the fix is really easy. A more serious bug is that LISA only work for queries where the length is multiple of k. This is not what they claimed to do in the paper. The paper implies there is no restrictions on query length w.r.t to k, and Algorithm 2 in their paper clearly indicates this. If the query length is not multiple of k, then the code ends up with infinite loop. We investigated the code and found they reduce the loop variable by k at each backward search step, but the loop only stops when the loop variable is 0. So when it is not a multiple of k, the loop never ends. This should be an easy fix so we let the loop stop when the loop variable is less or equal to 0. However, it then gives wrong output for such queries. In addition, the code seems to assume all queries having the same length. If the length varies, then only the longest queries can have correct results. As a result, we may say LISA's work is sort of problematic and has discrepancy between what they claimed to achieve and what they actually achieved. Furthermore, this leads to a serious problem that LISA actually cannot be applied to Pufferfish for many cases. Because many realistic kmer lenghts such as 31 and 63 are prime. LISA cannot accelerate them unless this bug is fixed. Due to time limits we are not able to fix it, but the benchmarking is still meaningful because once it is fixed, the benchmarking result won't change so will still be a useful outcome.

In addition, LISA doesn't return the actual query position in the original reference sequence. It doesn't store the suffix array and only returns the final interval in the FM-Index. To get the position in the original reference sequence, we saved the suffix array and added a single step to get the actual position. Since it does backward search by k characters at a time, it's more complicated to implement a sparse suffix array. Due to time limit we didn't do that.

3.4 Benchmarking Pufferfish and LISA

To get an idea how applying learned FM-Index to compacted dBG will perform without actually writing them as single tool, we benchmark the performance of corresponding parts in both tools. We add timers to the code regions we located. Then we run both tools with the same set of reference sequences and queries. We directly report the size of saved binary files of those parts. For Pufferfish, we only include files for MPHF and position vector. For LISA, we include the IPBWT and the saved ML model as well.

Since LISA paper didn't report the impacts of k, the number of RMI leaf nodes, and the number of threads, we decided to first benchmark the performance impact of these parameters. Supposedly, larger k would give shorter query time but larger IPBWT. Similarly, more RMI leaf nodes would give better predictions (so shorter query time) but larger space for the model. Different configurations of them can make the experiments complicated. Therefore, we first do experiments with small reference sequences to obtain a relatively optimal parameter configuration for LISA that gives good query time and acceptable space. We then use this optimal configurations to run LISA and Pufferfish with larger sequences (human genomes), and make the final comparison.

4 Results

In this section, we introduce our experimental setup and present results of our experiments. We show our benchmarking for time and space performance of Pufferfish and LISA w.r.t different parameters. We analyze and compare their performance and then draw conclusion on how applying learned FM-index over the compacted dBG will perform.

4.1 Experimental Setup

Firstly, as mentioned in Method section, we first benchmark LISA to study the performance impact of its parameters and obtain a relatively optimal

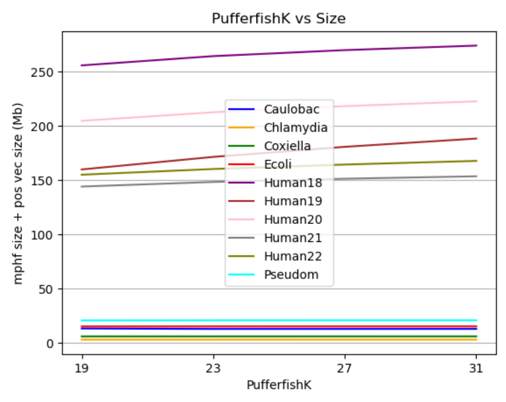

Fig. 3: Total size of MPHF and Pos vector versus Pufferfish k for different reference lengths

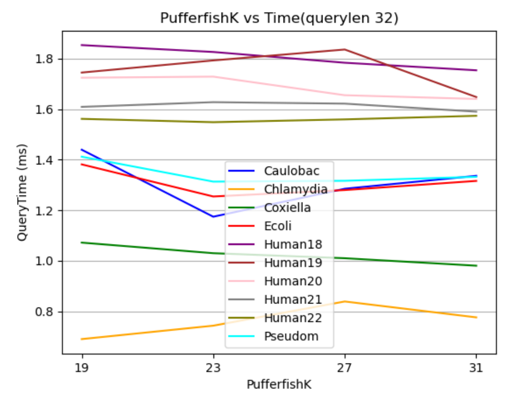

Fig. 4: Time of 10000 Pufferfish queries (each query length is 32) versus Pufferfish k for different references

parameter configuration. To do so we use 5 short reference sequences with different size downloaded from NCMI GenBank: Chlamydia (1.1MB), Coxiella (2MB), Caulobacter (4.1MB), Ecoli (4.7MB), Pseudomonas (6.4MB). For each sequence, we create query sets of 4 different lengths: 32, 64, 96, 128. We generate each query sets by randomly picking 10000 positions and then take m characters starting there as a query pattern. For LISA parameter k (the number of characters processed in one backward search step), since LISA only works for queries with length that is multiple of k, and the maximum value of k is set to 21, we pick 3 different k: 4, 8, 16. For the number of RMI leaf nodes, we checked LISA paper and found they used 2^{23}, 2^{26}, and 2^{27}. So we test 10 different values: $2^8, 2^{10}, ..., 2^{26}$. For number of threads, we tested 5 different values: 1, 2, 4, 8, 16. Therefore, in total we have run 5 x 4 x 3 x 10 x 5 = 3000 experiments.

After we obtained LISA result and decided the optimal parameters for LISA, we then benchmark both tools with longer sequences. We downloaded human chromosome 18 (81.5MB), chromosome 19 (59.5MB), chromosome 20 (65.4MB), chromosome 21 (47.4MB), and chromosome 22 (51.5MB). This time we are only testing with query sets containing 10000 queries of length 32. Since we are timing and calculating size for looking up kmers in the unitig sequence, to make fair comparison, we will treat those queries as kmers for Pufferfish. But Pufferfish only allows odd kmer sizes and kmer size cannot be greater than 31, therefore, for Pufferfish we build it with kmer size = 31. Moreover, we have confirmed that Pufferfish query time and size don't change a lot w.r.t kmer size. Figure 3 shows that when reference sequence is short, the total size of saved MPHF and position vector remains almost the same. When reference sequence is large, then the size of MPHF and position vector increases insignificantly. In Figure 4, total query time of 10000 queries (each query's length is 32) slightly changed in the range of ± 0.2 milliseconds, which can be ignored. Therefore, this should be fair and general enough. For LISA we build it with those optimal parameters. We report query time and structure size for query length = 32 for those built indices. We only time the unitig lookup for Pufferfish and only report its MPHF, position vector, and unitig sequence sizes. We calculate size by checking saved binary file sizes.

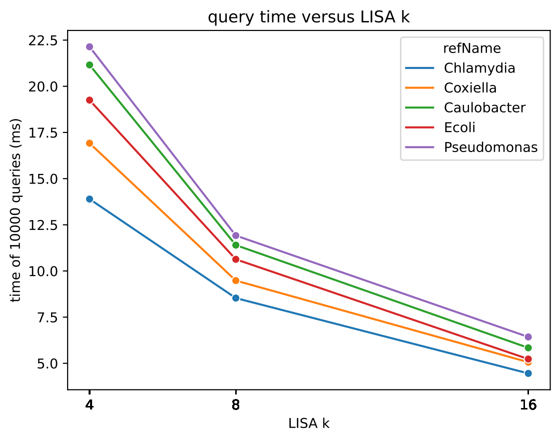

Fig. 5: Time of 10000 LISA queries versus LISA k for different query lengths

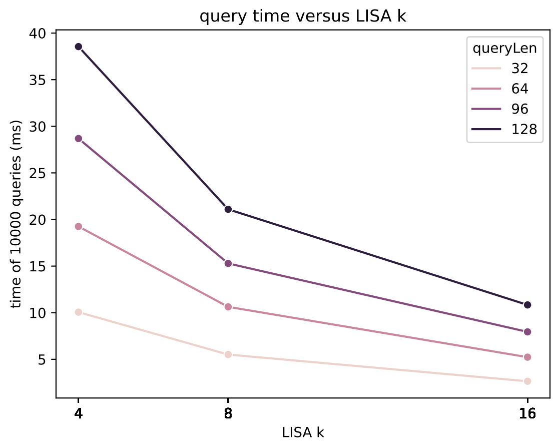

Fig. 6: Time of 10000 LISA queries versus LISA k for different references

4.2 LISA Benchmarking Results

We first analyze how k impact LISA query time and IPBWT size. Figure 5 and Figure 6 show the time of 10000 queries with different k's. Figure 5 uses query length = 64, Figure 6 uses Ecoli sequence. We can see as k grows, the query time decreases, and this is true for all sequences and all query lengths. This is expected as k is the speedup factor.

We then check the size of IPBWT. We don't include the model size since it is irrelevant to k. And query length is also irrelevant to the size. Surprisingly, Figure 7 indicates that the size of IPBWT is not related to

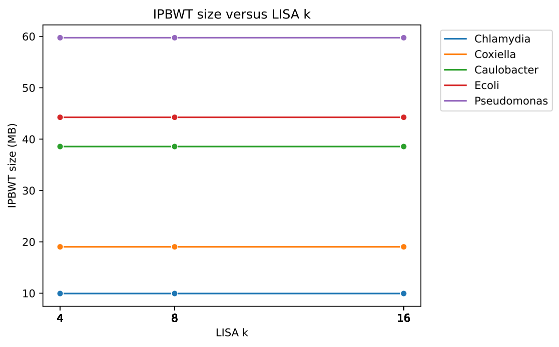

Fig. 7: IPBWT size versus LISA k for different references

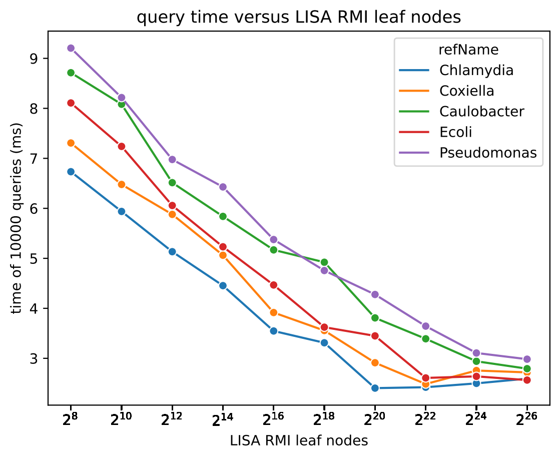

Fig. 8: Time of 10000 LISA queries versus RMI leaf nodes for different references

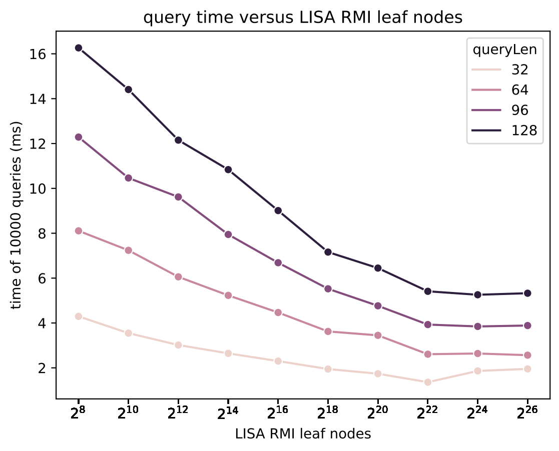

Fig. 9: Time of 10000 LISA queries versus RMI leaf nodes for different query lengths

k for all reference sequences. Ideally smaller k should give smaller size because each IPBWT entry stores fewer characters. We checked this is also true for much larger sequences (human genomes). We then checked the code and found LISA fixes the size of a single IPBWT entry to be **10 bytes**. Although kmers are encoded to shorter values, this might not be the best design. They could compress the size based on k and reference size but they chose to pick a large value that works for any k and reference sizes. But anyway, based on this design, we should use the largest k (16) since the time decreases and the size is constant.

Next, we analyze the impact of the number of RMI leaf nodes. Figure 8 and Figure 9 show the time of 10000 queries with different numbers of RMI leaf nodes. Again Figure 8 uses query length = 64, Figure 9 uses Ecoli sequence. We can see as the number of RMI leaf nodes grows, the query time decreases, and this is true for all sequences and all query lengths. This is expected since more number of RMI leaf nodes means the model is more accurate. So it predicts a smaller interval thus makes query faster.

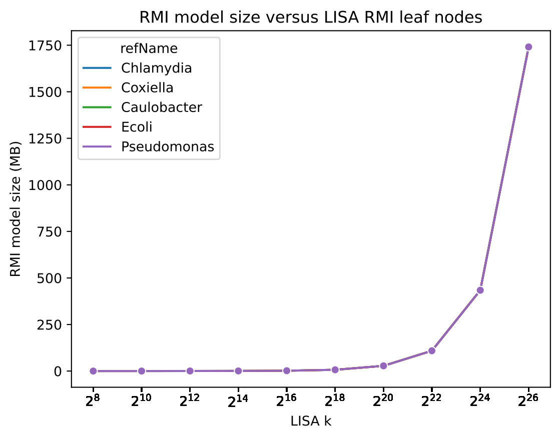

Fig. 10: Model size versus RMI leaf nodes for different references

We then check its impact of size. This time we only calculate model size since IPBWT is irrelevant to the number of RMI leaf nodes. Query length is irrelevant to the size as well. Figure 10 shows the model size is relatively small and constant before 2^{20} leaf nodes. After that it grows exponentially. Combined with the time plot, we found 2^{20} happens to be the fastest point or close to the fastest point for all sequences and query lengths. Therefore, we are happy to say 2^{20} is a optimal value for RMI leaf nodes.

Besides, we checked how multi-threading impact the performance. It's weird that the more threads we use, the longer it takes to query. We didn't figure out why is this happening due to time limits. So in all experiments and analysis we are using only one thread for both tools.

To summarize, by conducting a comprehensive experiment set for LISA, we visualized the relationship between LISA parameters and its time and space performance. A complete performance data file can be found in the Github repo. We found an optimal configuration that is k = 16, RMI leaf nodes = 2^{20}, and 1 thread. We will use these for the experiments on human genomes.

4.3 Comparison between Pufferfish and LISA

In this section, we present our experiment results on human genomes and make comparison between Pufferfish and LISA with the optimal configuration. Figure 9 shows the time of 10000 queries of both tools on human chromosomes with different lengths. As mentioned in the setup, Pufferfish is using kmer size = 31 and LISA query length is 32. We can see LISA is taking about twice longer than Pufferfish. This is expected because Pufferfish is constant time query and LISA takes about logarithmic time.

LISA is not doing too bad in terms of time compared with Pufferfish, however, we didn't expect it to perform much worse than Pufferfish on size. Figure 10 shows the structure size of both tools on human chromosomes with different lengths. We can see LISA takes a much larger space than Pufferfish. We further printed out the size just for the model in LISA. Even with 2^20 nodes, the size is small, so the IPBWT takes a huge space. Table 1 lists detailed data.

Therefore, LISA performs worse than Pufferfish on both time and space. As a result, we have to conclude that unless significant changes are made to LISA, it doesn't make sense to replace the unitig sequence lookup in Pufferfish with LISA. Perhaps LISA runs faster than regular FM-Index as reported in their paper, but it expands the size too much, which makes it not a worthy trade-off between time and space. And we do suspect that's why the original LISA paper didn't include any analysis for their size.

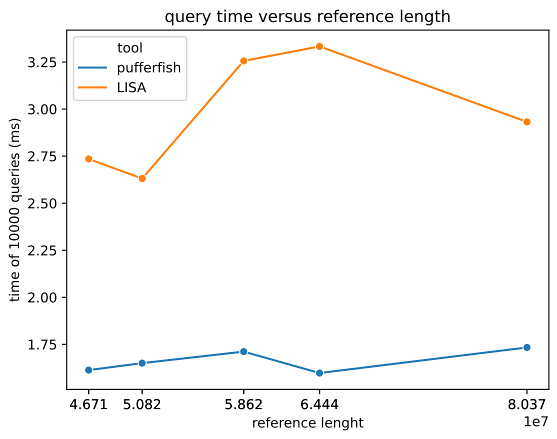

Fig. 11: Time of 10000 queries versus reference length for both tools

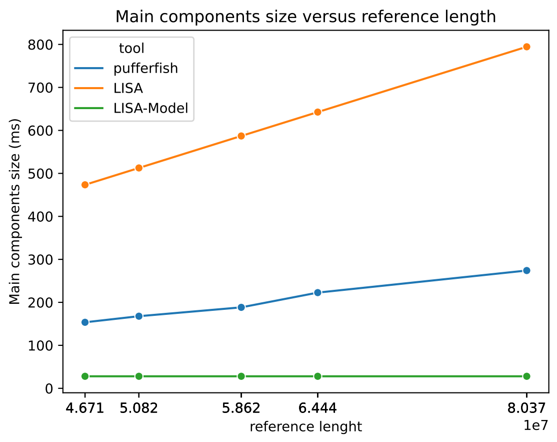

Fig. 12: Main component size versus reference length for both tools

5 Conclusion

To summarize, in this project, we proposed an idea to replace FM-index in some de Bruijn graph tools with learning index model and test if it can perform a better trade-off between runtime and space.

There were some challenges we faced when doing this project. The first challenge is that we had was we didn't have enough knowledge of popular de Bruijn graph tools. So we think it is important to learn about modern tools of de Bruijn graph and learn index models in order to do this porject. After discussing with Professor Rob Patro, we decided to try dbgfm and pufferfish. Dbgfm is a tool that relies on many dependencies. Some of those dependencies have version issues and bugs that need to

Table 1. Time and space data for Pufferfish and LISA on human chromosomes

tool	refName	refSize	time(ms)	size(MB)
Pufferfish	Human 18	81.5	1.733435	274.03
Pufferfish	Human 19	59.5	1.711304	188.37
Pufferfish	Human 20	65.4	1.597484	222.64
Pufferfish	Human 21	47.4	1.613134	153.61
Pufferfish	Human 22	51.5	1.650157	167.78
LISA	Human 18	81.5	2.932	794.49
LISA	Human 19	59.5	3.256	587.02
LISA	Human 20	65.4	3.333	642.58
LISA	Human 21	47.4	2.735	473.46
LISA	Human 22	51.5	2.631	512.64

be fixed. When we got it run, we noticed that this tool is lack of some functionality so we finally decided to use pufferfish and LiSA.

Then the biggest challenge of this project shows up. Since LiSA is the only FM-index based learning model, it is our only choice. But this tool is built upon Intel compiler and this compiler is hard for personal machines to run. In the meantime, there were several issues in this tool which makes it difficult to run. Nevertheless, we spent a lot of time and managed to locate those problems and fixed them or provided fixing suggestions. Our original idea was to combine LiSA and pufferfish to make a new tool and test its performance. But because of the time limitation, we switched to first do benchmarks with different references and parameters. We have demonstrated a comprehensive analysis of LiSA time and space performance regarding its parameters, which could make up the insufficiency in the original LiSA paper. We further demonstrated an comparison between performance of Pufferfish and LiSA. Comparing the benchmark result of them, we conclude that it is not worth to replace MPHF and position vector with LiSA in Pufferfish.

6 Future Work

Although the result shows that LiSA doesn't have an expected performance, it is still interesting to learn about learned index and benchmark this tool. In the future, we could discover more about this learning model and try to improve its performance by fixing those issues. If people are interested, they can use our work as a baseline or a guide for future attempts. On the other side, we can also add the functionality in dbgfm so that it can find the position of a kmer in the reference sequence, then try to combine it with LiSA to see it this could improve the performance of dbgfm.

Acknowledgements

We appreciate Professor Rob Patro for providing related papers and helping us come up with ideas. We appreciate Professor Rob Patro and UMIACS for helping us get access to CML machines.

Advice

We started this project with limited knowledge of existing de Bruijn graph tools and learned index model. So we spent a lot of time on exploring different papers and algorithms. If we were re-starting doing this project, we wouldn't spend too much time on find resources but would discuss our idea with professor sooner so that we can get more helpful advice.

Implementing learning index over dBG is an interesting topic that combines machine learning and bioinformation. It is also fun to try different tools and data structures with actual genomes. Our initial goal was too ambitious that is too difficult to finish it in half semester. So it would great if we could aim on a small thing first and then gradually expand the plan.

Although the result shows that LiSA doesn't reach our expectation, it is still great to know that there are many learning index model for suffix array and other data structures. We think it could be an interesting project that tries to combine learning index with other tools that uses suffix array to see how the performance will change.

References

Ho, Darryl and Ding, Jialin and Misra, Sanchit and Tatbul, Nesime and Nathan, Vikram and Md, Vasimuddin and Kraska, Tim (2019) LISA: Towards Learned DNA Sequence Search. *bioRxiv*

Ferragina, P. and Manzini, G. (2000) Opportunistic data structures with applications. *Proceedings 41st Annual Symposium on Foundations of Computer Science*, 390-398.

M. Burrows and D. J. Wheeler (1994) Block-Sorting Lossless Data Compression Algorithm. *Technical Report, 124, Digital Equipment Corporation*

Melanie Kirsche and Arun Das and Michael C. Schatz (2020) Sapling: Accelerating Suffix Array Queries with Learned Data Models. *bioRxiv*

Jung, Youngmok and Han, Dongsu (2022) BWA-MEME: BWA-MEM emulated with a machine learning approach. *Bioinformatics* **38(9)**, 2404-2413.

Fatemeh Almodaresi, Hirak Sarkar, Avi Srivastava, Rob Patro (2018), A space and time-efficient index for the compacted colored de Bruijn graph. *Bioinformatics*

Chikhi, Rayan and Limasset, Antoine and Jackman, Shaun and Simpson, Jared T. and Medvedev, Paul (2015), On the Representation of De Bruijn Graphs. *Journal of Computational Biology*

Beller, T. and Ohlebusch, E. (2016). A representation of a compressed de Bruijn graph for pan-genome analysis that enables search. *Algorithms for molecular biology*