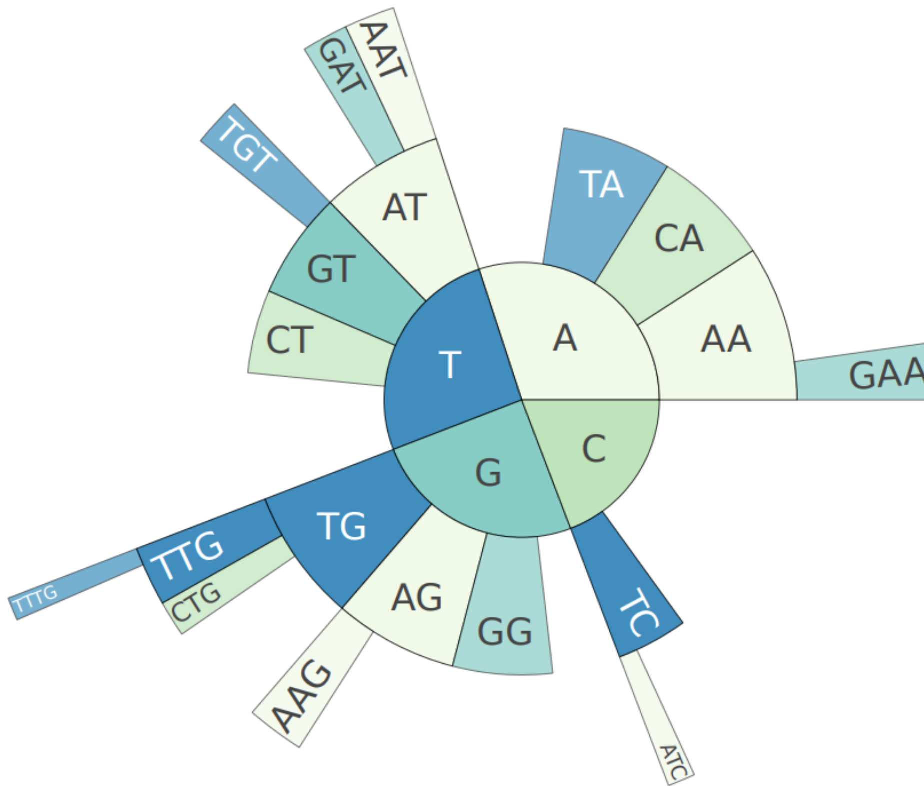
CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG



Efficient distance computations of VLMCs in DNA sequencing

Master's thesis in Computer science and engineering

Sebastian Holm
Johan Atterfors

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2023

Master's thesis 2023

Efficient distance computations of VLMCs in DNA sequencing

Sebastian Holm
Johan Atterfors

UNIVERSITY OF
GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2023

Efficient distance computations of VLMCs

Sebastian Holm, Johan Atterfors

© Sebastian Holm, Johan Atterfors, 2023.

Supervisor: Joel Gustafsson, Department of Infectious Diseases, Institute of Biomedicine, Sahlgrenska Academy, University of Gothenburg
Examiner: Dr. Alexander Schliep, Department of Computer Science and Engineering, Chalmers University of Technology

Master's Thesis 2023
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Cover: Visualization of a Variable Length Markov Chain. Displayed in the image are the k-mers present in the VLMC.

Typeset in LaTeX
Gothenburg, Sweden 2023

Efficient distance computations of VLMCs

Sebasitan Holm, Johan Atterfors
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Abstract

Classifying species is a crucial part of efforts to stop emerging pathogens e.g. viruses and dangerous diseases. As DNA sequences can be many gigabytes large and the size of current DNA databases is enormous and growing fast, the classification of species is a computationally expensive task. One type of model for DNA sequences is the Variable-Length Markov Chain (VLMC) which has been shown to capture the essential aspects of sequences while reducing memory footprint. A promising efficient method of comparing VLMCs is the d_v^* measure developed by Gustafsson et al. where the main bottleneck is finding the intersection between two VLMCs. This has been shown to currently be cache inefficient.

In this thesis, we reduce the execution time and memory access of the d_v^* dissimilarity function, a problem that involves quickly accessing data. We investigated cache-oblivious and other data structures that were adapted to store VLMCs for efficient dissimilarity computation. These structures include the van Emde Boas layout, Eytzinger layout, B-tree layout, Sorted vector, Hash map, and a sorted vector with an index structure we call Sorted Block Search (SBS). We further move from a nested for-loop to utilizing a cache-oblivious matrix transpose algorithm, which we call matrix recursion when constructing a matrix of distances between VLMCs.

The data structure, Hash map, attains a speedup of up to 19 times compared to the state-of-the-art when computing all distances between 15 446 VLMCs created from the NCBI Virus Ref Seq database. For larger VLMCs of 10-50 MB, one of the fastest data structures, Sorted vector, attains a speedup of up to 15 times. The new algorithm is benchmarked on a laptop, desktop and one compute cluster. The presented improvements can be utilized in pathogen classification and evolutionary genomics as the new algorithm show good scaling for compute clusters, making these optimizations promising.

Keywords: computer science, Markov Chain, Variable Length Markov Chain, VLMC, K-mer, algorithm engineering, oblivious, bioinformatics

Acknowledgements

We extend our gratitude to our supervisor Joel Gustafsson for his great support and stimulating suggestions which always pushed us to new ideas. We also appreciate our examiner Alexander Schliep for his great enthusiasm for the project and its end purpose in all its domains.

Johan Atterfors & Sebastian Holm, Gothenburg, June 2023

Contents

List of Figures **xi**

List of Tables **xiii**

1 Introduction **1**
1.1 Purpose . . . 3
1.2 Limitations . . . 3

2 Theory **5**
2.1 The alignment-free model and Markov Chains . . . 5
2.1.1 k-mers . . . 5
2.1.2 Higher-order Markov Chains . . . 5
2.1.3 Variable Length Markov Chains . . . 6
2.1.4 Computing the distance between VLMCs . . . 7
2.2 The cache hierarchy . . . 9
2.2.1 The formal model . . . 9
2.2.2 Types of cache misses . . . 11
2.2.3 Flatness . . . 11
2.3 Cache awareness and obliviousness . . . 12
2.3.1 Matrix transpose problem . . . 12
2.4 Cache efficient data structures . . . 12
2.4.1 Scanning an array . . . 13
2.4.2 Eytzinger Layout . . . 13
2.4.3 van Emde Boas layout . . . 13
2.4.4 B-tree layout . . . 14
2.5 Prefetching . . . 15
2.6 Vectorization . . . 16
2.7 Ahmdal's law . . . 17

3 Methods **19**
3.1 Input data . . . 19
3.1.1 Data sets . . . 19
3.1.2 VLMC inspection . . . 21
3.2 VLMC data structures . . . 23
3.2.1 k-mer representation . . . 24
3.2.2 Probabilistic Suffix Tree Hash Map . . . 24

3.2.3 Hash Map . . . 24
3.2.4 Sorted vector . . . 25
3.2.5 Eytzinger layout . . . 25
3.2.6 van Emde Boas layout . . . 27
3.2.7 B-tree . . . 28
3.2.8 Sorted Block Search . . . 29
3.2.9 K-mer buckets . . . 29
3.3 Algorithmic Implementations . . . 30
3.3.1 Building VLMCs . . . 30
3.3.2 Loading VLMCs to memory . . . 32
3.3.3 Computing distances between all VLMCs . . . 32
3.3.4 Matrix recursion . . . 32
3.3.5 Computing the distance between two VLMCs . . . 33
3.4 Parallelization . . . 34
3.5 Downsampling VLMCs for speed . . . 34
3.6 Benchmarking . . . 35
3.6.1 Perf diagnostics tool . . . 36

4 Results and Discussion **37**
4.1 Speedup . . . 37
4.1.1 Speedup to VLMC set size . . . 41
4.2 Memory & Cache . . . 42
4.2.1 Cache misses . . . 44
4.2.2 Space usage . . . 50
4.3 Parallelization . . . 51
4.4 Matrix recursion . . . 52
4.5 Effects of Downsampling . . . 52

5 Conclusion **55**

Bibliography **57**

A Code listings **I**

B Additional table **IX**

List of Figures

1.1 The DNA helix strand with the nucleotides A, C, G, T, and the phosphate backbone. . . . 2

2.1 Illustration of a first-order Markov Chain with the states; A, C, T, G and the transition probabilities. . . . 6
2.2 Illustration of a second-order Markov Chain with the states; AA, AC, CA and the alphabet $\Sigma = \{A, C\}$. . . . 6
2.3 VLMC with the genetic alphabet, showing all 1-mers ("A", "C", "T", "G") and one 2-mer ("CA"). Next-character probabilities are stored at each node. The root is the empty string. . . . 7
2.4 The memory hierarchy with the CPU registers at the top, being the fastest and most expensive memory. L1-L3 are the three most common cache levels of common respective sizes; 128 KB, 1MB, and 8MB on a laptop. Main memory and disk are the last and slowest memory type, these are also the cheapest, which is why they are usually much larger in size [11]. . . . 10
2.5 The two-level memory model with Demaine's notation for cache and block size. Note that the main memory is considered infinite and that the cache contains M blocks of B bytes. . . . 10
2.6 The binary search tree of a set of integers 1,. . . ,7. As binary search starts by examining the middle element, it is stored in the beginning with the next elements to be examined after it. . . . 14
2.7 van Emde Boas layout for a set of n elements. The top subtree A has $\sqrt{n}$ elements and there are $\sqrt{n}$ subtrees each of size $\sqrt{n}$. . . . 14
2.8 A B-tree layout for $B = 2$. . . . 15
2.9 Visualization of Ahmdals' law for different sizes of parallelizable section. 17

3.1 Average percentage of k-mers included for each k and data set. Each k-mer length has, for the length n, 4^n possible k-mers. . . . 21
3.2 Percentage of hits, misses, and ignored k-mers on average when comparing VLMCs from different datasets. . . . 22
3.3 The $\log 10$ of average misses in a row over slices of k-mers $\frac{n}{100}$ large from the VLMCs compared. . . . 23
3.4 Ordering of the first 100 k-mers when loading a Virus VLMC from disk. Index 0 indicates the first item of the unsorted vector. . . . 26

3.5 Ordering of the first 100 k-mers when loading a human VLMC from disk. Index 0 indicates the first item of the vector. 5 major subsets can be seen between index 0-19, 20-39, etc. with further sorted subsets within. . . . 26
3.6 Layout of the Sorted Block Skip data structure. . . . 30
3.7 Steps of the procedure to compute the distance of DNA sequences. . 31
3.8 Vizualizing the recursive splitting of a 4x4 matrix. Note the new base case of one element (pair of VLMCs). . . . 33
3.9 The execution order of the matrix recursion when evaluating a 4x4 matrix. . . . 33
3.10 Parallelization of the distance computation on 4 cores, with the two sets of VLMCs of size 7 and 12 respectively. . . . 35

4.1 Laptop speedup versus PST for different containers and data sets. . 38
4.2 Desktop speedup versus PST for different containers and data sets. . 39
4.3 HPC speedup versus PST for different containers and data sets. . . . 40
4.4 Desktop speedup versus PST for the different containers when comparing the increasing number of VLMCs from the virus dataset. . . . 42
4.5 Desktop execution times for the different containers when comparing the increasing number of VLMCs from the virus dataset. . . . 43
4.6 Laptop cache misses for different containers and data sets. . . . 46
4.7 Desktop cache misses for different containers and data sets. . . . 47
4.8 HPC cache misses for different containers and data sets. . . . 48
4.9 Laptop cache reference counts for different containers and data sets. . 49
4.10 Parallel speedup of the program in relation to the theoretical maximum speedup. Theoretical maximum parallel speedup is almost infinite since distance computation takes about 99% of execution time on large enough data sets. . . . 51
4.11 Relative error and execution time when comparing distance for the turkey dataset with the vEB container. . . . 53
4.12 Classification test on the Virus dataset. The 0th closest is the VLMC compared to itself. After 10% downsampling the three closest samples are misclassified and after 30% only the identity is correctly identified. 54

List of Tables

3.1 The average GC-content and amount of characters that are unidentifiable (The unidentifiable characters are ignored during the creation of the VLMC). 20
3.2 Characteristics of the virus genome data sets. 20
3.3 Characteristics of the human genome data sets. 20
3.4 Characteristics of the corn genome data sets. 20
3.5 Characteristics of the turkey genome data sets. 20
3.6 Table of the benchmarking machine's hardware. 35

4.1 Elapsed time on Laptop for distance computation on the Human, Turkey, and Corn data sets combined. DNF is a crash, killed by the OS. 43
4.2 Elapsed time on Laptop for distance computation on the Virus-to-Virus data set. 44
4.3 Heap memory usage for the different implementations on Small Human (24 VLMCS to 24 VLMCs) and Small Virus (2000 VLMCs to 2000 VLMCs). The Heap column is the total heap allocated, and the Unused heap is the extra memory allocated to the program which is not used. The heap memory evaluation was generated on the Laptop. 50
4.4 Performance difference when using matrix recursion and a nested for-loop when computing the distances between all VLMCs in the Virus data set. Time and cache misses are averaged over 5 runs, with the standard deviation in the rightmost column. 52

B.1 Execution time on Laptop for the Large Human and Virus data sets when using `Double`, `Eigen`, and `Float`. IX

1 Introduction

Classifying species is crucial to stopping emerging pathogens (e.g. viruses) before they become pandemic. Consider a new contagious disease spreading across the world, or a pathogen among farmers in a rural area of a country that lacks clean drinking water. In the former, rapid and accurate classification of the new virus could help negate the worldwide effects. The recent pandemic caused by the emergence of a novel coronavirus species, and variants of this species, is an obvious example. In the latter, fast classification of the infectious substance is highly prioritized to start the correct countermeasures quickly.

Another important field of research is how species evolve. This is commonly studied through the lens of evolutionary trees (phylogeny) where related species are ordered according to their evolution from a common ancestor. Studying evolution can reveal how species came to be and what genetic mutations they have undergone to explain different characteristics.

Today the task of classifying species is generally done through genome analysis. An organism's genome contains all the information to create proteins, cells, organs, etc., and is a unique identifier among all living entities. All organisms can be ordered by their taxonomy, a system of classifying organisms given specific biologic characteristics. Species in the same taxonomic family have generally similar genomes. For example, dogs and wolves share large parts of their genomic sequences [1].

The genomic sequence is stored in all organisms' cells in the DNA which is composed of long sequences of nucleotides. Nucleotides are active in many parts of living organisms, in the genomic sequence they typically form long chains of DNA, or for some viruses, RNA. DNA is two long chains of nucleotides curled together in the shape of a helix, while RNA is one chain wrapped around itself. The chain of molecules is made up of the four nucleotides Adenine (A), Cytosine (C), Guanine (G), and Thymine (T), or in the case of RNA instead of Thymine, Uracil (U). In DNA, these nucleotides form pairs, base pairs, with each other; A-T, T-A, G-C, and C-G. These pairs are held in place in the helix by the phosphate backbone, see Figure 1.1. Due to the symmetry of the base pairs, it is sufficient to view the double-stranded DNA as a single sequence of As, Cs, Gs, and Ts.

The genomic sequences of organisms vary in length with viruses from the NCBI Virus[1] dataset ranging from 136 to 2 500 000 bp (base-pairs) and human chromo-

[1] `https://www.ncbi.nlm.nih.gov/labs/virus/vssi/#/virus?SeqType_s=Nucleotide&`

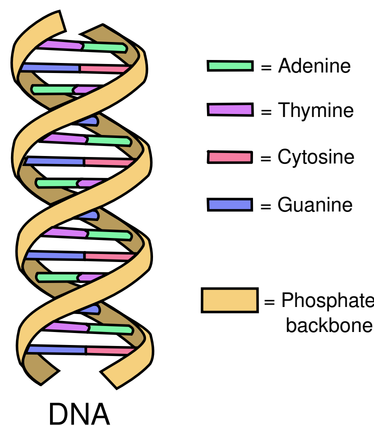


Figure 1.1: The DNA helix strand with the nucleotides A, C, G, T, and the phosphate backbone.

somes from NCBI human genome assembly[2] in the ranges of 45 000 000 to 200 000 000 bp. An entire human genome uses slightly more than 3GB of memory in a computer. The progress in data storage and management as well as the improvements to sequencing technology has led to large databases with accessible genomic data with species information. New genomes can then be compared to subsets of the species in these databases to classify them and determine their evolutionary relationship.

The sheer size of genomic sequences in combination with the number of available samples in the databases makes comparisons over these a time-consuming task. Computational models have expanded from just direct comparison of the sequences, also known as sequence alignment, to also include alignment-free methods which generally rely on counting the frequencies of short sequences [2]. A common example is GC content, comparing the percentage of G and C nucleotides. Alignment-based methods are usually slower with computational complexities in $O(n^2)$ [3], while GC-content is of the order of $O(n)$, where n is the length of the longest genomic sequence.

One type of alignment-free method is based on utilizing a Variable Length Markov Chain (VLMC) to represent a genome [4], [5]. As an example, counting all pairs of nucleotides and computing their relative frequencies can be used to represent a second-order Markov chain. Counting all As, ACs, Cs, CAs and GACs with their respective frequencies would create a Variable Length Markov Chain as sequences may vary in length. The chosen sequences are extracted through an algorithmic procedure that computes the information in each sequence, creating compact representations of the genomes. See [6] for algorithms in the efficient computation of VLMCs.

The compact representation through VLMCs speeds up the comparisons between genomes since the time complexity is no longer dependent on the length of the

SourceDB_s=RefSeq

[2]https://www.ncbi.nlm.nih.gov/data-hub/genome/GCF_009914755.1/

genome representation. The comparison of VLMCs is valid and provides accurate similarity measures between genomes [6], [7]. This distance computation is improved upon in this thesis to allow faster comparisons to other genomes.

1.1 Purpose

This thesis is an extension of an ongoing project by Gustafsson to utilize VLMCs in DNA sequence comparison [6]. Gustafsson identified the problem of memory access patterns to be a limiting factor in execution time [7]. The thesis presents improvements to reduce memory bottlenecks by considering memory-efficient algorithms and data structures. One use case is when clustering many VLMCs, which is the procedure of computing the distance between all pairs of VLMCs in a dataset. Improvements are compared to the current state-of-the-art Probabilistic Suffix Tree (PST)[3] approach.

The purpose of improving the algorithm is to extend the current capabilities in classification by improving the computational efficiency of comparing genomic sequences. An increase in efficiency could prove useful in disease detection by sequence classification and evolutionary research.

1.2 Limitations

We limit the optimization effort to one distance function, d_v^* described in section 2.1.4, which has been found to perform well by Gustafsson et al. [8]. In general, many of the optimizations made can easily be translated to some other distance metric used for comparing VLMCs.

[3] `https://github.com/Schlieplab/PstClassifierSeqan`

2

Theory

This chapter provides a description of the alignment-free model used to represent DNA sequences, followed by an introduction to caches and cache misses and formal descriptions of the cache. Further, various optimizations that can be used to construct faster algorithms are presented. Finally, a review of Ahmdal's law, the law of maximal speedup of a program is given.

2.1 The alignment-free model and Markov Chains

As described in section 1, alignment-free models rely on the frequencies of subsequences in DNA. This section will explain one alignment-free model, the Variable Length Markov chain (VLMC) that provides an efficient representation of DNA sequences. This is done by introducing k-mers, higher order Markov Chains, and finally the VLMC.

2.1.1 k-mers

A k-mer is identified by a sequence of k characters and a count of the number of occurrences of the sequence. In the domain of DNA, the alphabet of possible characters is $\Sigma = \{A, C, G, T\}$, and a k-mer is the k-length word w_k. An example of a 4-mer could then be the 4-character sequence "ATTA" and its occurrence count. Obtaining a set of k-mers from a sequence can be achieved by a sliding window approach, where a window covering k characters is moved, one character at a time, along the sequence, capturing all subsequences of length k.

2.1.2 Higher-order Markov Chains

A Markov Chain (MC) is a model which has states and probabilities to reach each state that corresponds to the probability of reading each nucleotide after w_k. The probabilities are conditional on the history of a fixed number of states prior to the current state, counting from the current state and backward, Figure 2.1 illustrates a first-order MC. The sum of the transitions from a state always sums to 1. In the higher-order MC, more states are included in the history, e.g., in a fourth-order MC, the current state and three states previously visited determine the probabilities to reach the next state. A second-order MC is depicted in 2.2 where each transition is conditional on each state which now contains the two-letter words w_2.

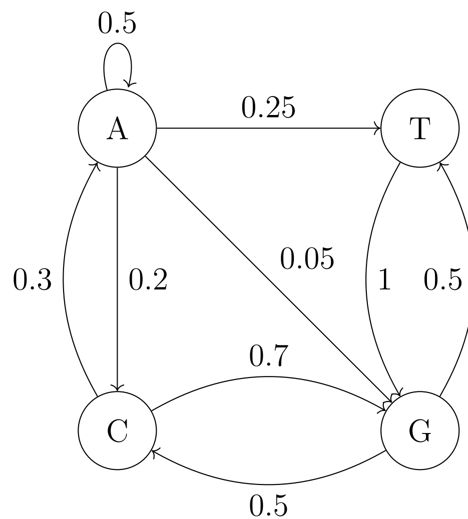


Figure 2.1: Illustration of a first-order Markov Chain with the states; A, C, T, G and the transition probabilities.

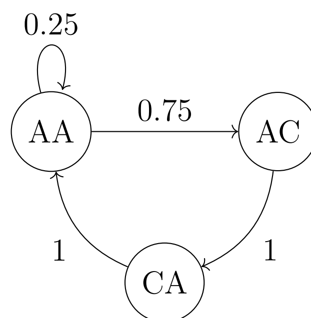


Figure 2.2: Illustration of a second-order Markov Chain with the states; AA, AC, CA and the alphabet $\Sigma = \{A, C\}$.

Note that the number of possible transitions is $O(k^n)$, where k is the number of states in an nth order MC. For example, the DNA alphabet is assumed to be the states, a 5th order MC has the order of $O(4^5)$ transitions, each with different probabilities. The 5th order MC in the example can be constructed by counting all words w_0, w_1 up to w_5 in a DNA sequence and computing all conditional probabilities based on their frequency.

2.1.3 Variable Length Markov Chains

One issue with the fixed order of the MC is its size. Namely, many words may appear infrequently or not at all in the DNA sequence leading to waste in space when storing it, and many transition probabilities that can not be accurately estimated. One solution is the VLMC. The VLMC is only briefly described here, for more details see Ron et al. [5] and Bühlmann et al. [4].

One method to construct a VLMC is to count the words w_k of length k, $k-1$, $\dots 1$ i.e. the k-mers with k, $k-1$, $\dots 1$ are counted as described in Section 2.1.1.

In contrast to the static order MC, however, only the words which appear in the sequence need to be stored. The chain is then constructed by starting from the empty string ϵ as the root of a tree, see Figure 2.3, where sequences are then added with the probability of each next character computed by using their counts. With each k-mer, there are four probabilities, one for each character of the alphabet, these probabilities will be referred to as next-character probabilities.

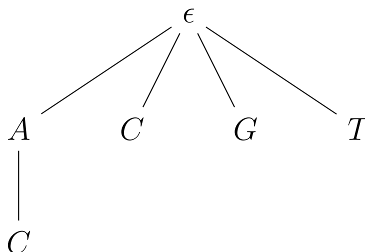


Figure 2.3: VLMC with the genetic alphabet, showing all 1-mers ("A", "C", "T", "G") and one 2-mer ("CA"). Next-character probabilities are stored at each node. The root is the empty string.

Once all occurrences of k-mers are included in the VLMC, some branches of the tree may be removed if they do not provide any useful information. Information in this context is the Kullback-Leibler divergence [4]. For example, in Figure 2.3, if the next-character probabilities for the k-mer "A" are the same as the next-character probabilities for "CA" the "C" can be dropped without loss of information. The act of dropping k-mers, or branches, is called pruning. The degree of pruning can be adjusted via a parameter K which determines how much loss of information is allowed between two versions of the tree, one with the k-mer included and one tree without the k-mer.

2.1.4 Computing the distance between VLMCs

The distance between two VLMCs is measured using d_v^*, an extension to the alignment-free dissimilarity measure d_2 [6], [8]–[10]. The distance only compares k-mers present in both VLMCs since it has been shown to produce the best results [8]. In the following, we give the definition of d_v^* where we let the alphabet be the set of all nucleotides, $\Sigma = \{A, C, T, G\}$ and a character in the alphabet is denoted by σ. A VLMC, λ_i, contains a set of k-mers, with which it shares a subset of its k-mers with another VLMC, λ_j. This shared set of k-mers is defined by the intersection of the sets $C \in \lambda_i \cap \lambda_j$. With these, we define the next-character probability for a character $\sigma \in \Sigma$ in the word $w \in C$ as $\hat{p}(\sigma|w, \lambda_i)$. The equations are

$$\hat{p}^M(\sigma|w,\lambda_i) = \frac{\hat{p}(\sigma|w\lambda_i)}{\sqrt{\hat{p}(\sigma|w[|w|-M:|w|],\lambda_i)}}, \tag{2.1}$$

$$F_v^*(\lambda_i) = \sum_{w\in C}\sum_{\sigma\in\Sigma}\hat{p}^M(\sigma|w,\lambda_i)^2, \tag{2.2}$$

$$D_v^*(\lambda_i,\lambda_j) = \frac{\sum_{w\in C}\sum_{\sigma\in\Sigma}\hat{p}^M(\sigma|w,\lambda_i)\hat{p}^M(\sigma|w,\lambda_j)}{\sqrt{F_v^*(\lambda_i)}\sqrt{F_v^*(\lambda_j)}}, \text{ and} \tag{2.3}$$

$$d_v^*(\lambda_i,\lambda_j) = \frac{2\ \texttt{arccos}(D_v^*(\lambda_i,\lambda_j))}{\pi}. \tag{2.4}$$

Equation 2.2 and 2.3 forms the so-called cosine similarity of two vectors. The cosine similarity of two n-dimensional vectors is the normalized dot product of the vectors. If we let the two vectors be called A and B we can derive $\cos(\theta)$ of the vectors from the Euclidian dot product formula as

$$A \cdot B = ||A||\ ||B||\cos(\theta), \text{ and}$$

$$\cos(\theta) = \frac{A\cdot B}{||A||\ ||B||} = \frac{\sum_{i=1}^{n} A_i\ B_i}{\sqrt{\sum_{i=1}^{n} A_i^2}\sqrt{\sum_{i=1}^{n} B_i^2}}.$$

If we substitute $\sum_{n=1}^{n} A_i^2 = F_v^*$ and $A_i = \hat{p}^M(\sigma|w,\lambda_i)$, and likewise for B_i we reach Equations 2.1-2.3. Finally, Equation 2.4 forms the so-called angular distance of two VLMCs by normalizing the cosine similarity as

$$\text{angular distance} = \frac{2\arccos(\cos(\theta))}{\pi}.$$

d_v^* is thereby a type of angular distance with the elements of the vectors as 4-dimensional vectors probabilities.

In the field of DNA classification, it is useful to normalize the probabilities of the next-character probabilities, $\hat{p}$, since we may assume certain expected frequencies of w_i for words of length i. The dissimilarity measure can include this type of assumption by using a *background-order*. The background-order of the probability is done through parameter M and the conditional probability changes to $\hat{p}^M$, Eq. 2.1. For example, the GC-content of a sequence can be normalized by applying a background order of 0, which normalizes $\hat{p}$ by the probability of each character σ. $w[j]$ is the word w without the first j characters, $|w|$ is as usual the length or number of characters in w.

Note that the distance metric does not obey the triangle inequality since different sets of k-mers can be shared between different VLMCs. However, it is reflexive, symmetric, and non-negative. See Gustafsson et al. [6], [8] for motivations.

In practice, a $d_v^*(\lambda_i, \lambda_j)$ dissimilarity close to 0 indicates that λ_i and λ_j are similar, and a dissimilarity close to 1 indicates that the sequences are highly dissimilar.

2.2 The cache hierarchy

As mentioned in section 1.1, the current bottleneck when comparing VLMCs is the memory access pattern producing a high amount of cache misses [7]. This section presents the notion of a cache and its analytical models.

The modern memory hierarchy is made up of several levels, where the speed of transferring data is correlated with its distance from the CPU [11]. The memory located closest to the CPU is the fastest to access but also the most cost intensive, which leads to it usually being the smallest in size. The hierarchy then consists of larger and slower memory levels, depicted in Figure 2.4. The cache is usually made up of three levels L1, L2, and L3.

When a program requests data that is not present in the cache, a cache miss occurs, which requires the program (and hardware) to fetch the data from RAM, to which the latency is much higher compared to the cache. While fetching data from slower memory, the program stalls until its data are available to continue executing, increasing total execution time.

An additional important aspect of the cache is associativity. The associativity of the cache is a number, describing the number of places one block from main memory may be placed in the cache. As the main memory is always assumed to be at least as large as the cache, multiple blocks share the same possible block placement in the cache. For example, in a 4-way associative cache, the block may be placed in any of 4 different locations in the cache. Associativity is important as a program may access the same memory addresses many times in succession, in which case associativity is used to reduce contention for the same cache blocks, dispersing the locations in which the data is placed.

2.2.1 The formal model

The cache hierarchy can be modeled directly but involves tracking each level's cache parameters making the process of designing algorithms that take the size of the cache into account (cache-aware algorithms) cumbersome. An alternative to explicitly modeling the memory hierarchy is the cache-oblivious model by Frigo et al. [12]. The model is a two-level memory hierarchy, usually referred to as memory and cache but can represent any two memory levels. Frigo et al. show that this model can represent arbitrarily many levels of the memory hierarchy.

The two-level memory hierarchy is best explained using Demaine's nomenclature [13] for a system with a CPU with some constant number of registers, a cache, and a main memory (RAM) (see Figure 2.5). The cache is divided into blocks of B bytes in a contiguous layout, where transfer between layers of memory is done in these fixed-sized blocks. These blocks are referred to as cache lines. The main memory is assumed to have unlimited size while the cache is limited to M bytes of $\frac{M}{B}$ blocks.

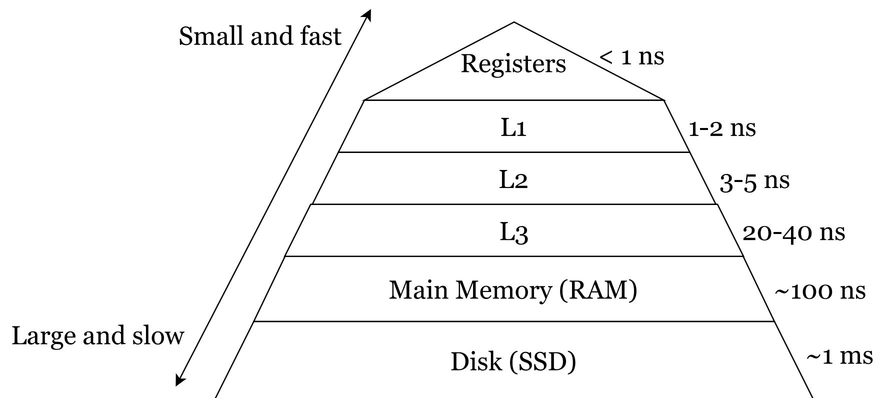


Figure 2.4: The memory hierarchy with the CPU registers at the top, being the fastest and most expensive memory. L1-L3 are the three most common cache levels of common respective sizes; 128 KB, 1MB, and 8MB on a laptop. Main memory and disk are the last and slowest memory type, these are also the cheapest, which is why they are usually much larger in size [11].

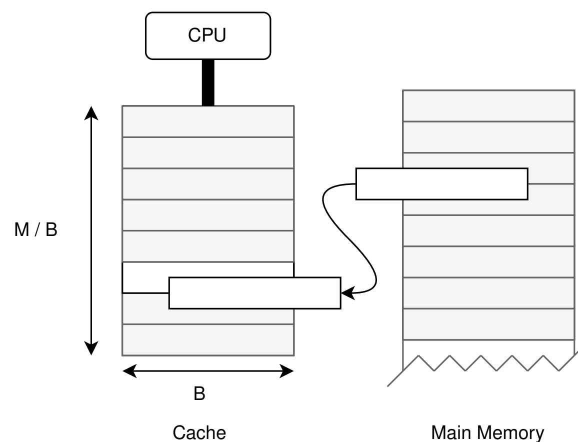


Figure 2.5: The two-level memory model with Demaine's notation for cache and block size. Note that the main memory is considered infinite and that the cache contains M blocks of B bytes.

2.2.2 Types of cache misses

Generally, there are a few types of cache misses, also known as the 5Cs (five types, each named with a letter beginning with C). The first three Cs are relevant for both single-threaded and parallel systems or programs, while the last two occur only for parallel systems.

The first type is the Compulsory miss, which occurs when data requested by the CPU has not been referenced before, and it must be fetched from the main memory. This type can also be referred to as a "cold" miss and can usually not be avoided. Next is the Conflict miss, which occurs when the same data compete for the same cache line due to associativity. This type of miss can be a problem in low associativity caches (4-way or less) when accessing addresses that map to the same cache line. The third C is the Capacity miss, caused by the size of the cache being too small. This occurs when the so-called working set, or the memory which a program frequently uses, is larger than the cache.

Cache misses that arise from multiple cores accessing the same data are related to the coherence of the cache, as all caches must be kept synchronized. The fourth type is a Coherence miss. It occurs when a core writes to an address that some other core has already fetched, but due to the write invalidating the data, the core must fetch the new value, resulting in a cache miss. The final type is the Coverage miss. When the cache has a coherence directory (memory reserved to track cache lines among cores) the miss occurs when data has been evicted from the coherence directory as the directory does not have the capacity to track all cache lines at once, and the memory cannot be assumed to be non-modified.

2.2.3 Flatness

Generally, cache-aware or cache-oblivious algorithms frequently assume a so-called flatness property of the data structure. Flatness is defined here as "a single, contiguous block of memory assigned to one data structure". A common data structure fulfilling this definition is the standard array for which data is physically stored in adjacent addresses.

Flatness is relevant as cache lines are loaded through contiguous memory blocks, and access to a specific memory address also loads the neighboring addresses into the cache. If a data structure fulfills the flatness property, accessing elements that are in the same cache line results in cache hits.

As an example, a naive approach when implementing treelike data structures is to not use a flat memory allocation. Instead, nodes in the tree are connected through pointers with a separate memory allocation for each node. Not only can segmented memory allocation strain the operating system but it also segments the data structure. Each node that is visited must be explicitly loaded from main memory. In comparison, in sections 2.4.2 - 2.4.4 below, layouts, where the next nodes of a tree are implicitly loaded into the cache due to flatness, are presented.

2.3 Cache awareness and obliviousness

As previously mentioned, in the cache-aware solution, the data structure is tuned after the system's memory parameters, such as cache levels, sizes, and transfer speeds. This allows the data structure to minimize data transfers to and from cache. In cache-oblivious solutions, the data structure has no knowledge of the systems memory parameters but instead, memory is placed in a near-optimal structure that does not depend on block size. Often, this leads to cache-aware algorithms performing slightly better, whereas cache-oblivious algorithms perform worse with the benefit of not requiring configuration for each machine [14].

2.3.1 Matrix transpose problem

As an example, Matrix transposition can be achieved using a cache-oblivious algorithm. Matrix transposition is a common operation in linear algebra with particularly poor cache behavior using the naivë approach [15] (seen in listing 1 for a matrix A). The main problem with the naive approach is that elements swapped are stored in different parts of the memory, so there is barely any benefit to caching nearby elements since they will not be used until later.

```
1  for (i = 1; i < n; i++){
2      for (j = 0; j < i; j++){
3          tmp = A[j][i]; A[i][j]=A[j][i];
4          A[j][i]=tmp;
5      }
6  }
```

Listing 1: Naive matrix transposition with poor cache performance [15].

A better approach to transposing a matrix was presented by Frigo et al. [12]. A matrix A is recursively split on its largest dimension, creating two separate matrices which are to be transposed. Eq. 2.5 shows the transposition of some sub-matrix A into matrix B.

$$B = A^T = \begin{bmatrix} A_1 \\ A_2 \end{bmatrix} \tag{2.5}$$

The process of splitting on the largest axis continues until A consists of two elements, these are then swapped. The recursive matrix transposition algorithm has been shown to obtain asymptotically optimal bound on memory transfers, $(1 + \frac{mn}{M})$, with a matrix of size n, m.

2.4 Cache efficient data structures

This section exemplifies cache-oblivious and aware data structures by one familiar example and three array layouts for fast search in static structures.

2.4.1 Scanning an array

A simple example to begin exploring cache-oblivious data structures is the array scanning problem. It is a common procedure in programming to iterate over a flat array and apply some function to all elements in the array. Assume a standard algorithm of starting at the first element and continuously accessing the next item in the array. This is a cache-oblivious algorithm as there is no parameter tuning. To find the bound of memory accesses, we have N items and a cache-line size of B. If the elements are placed optimally, the first element at the beginning of a cache line and the last element at the end of a cache line, there will be N/B memory transfers. In the worst case, the array will span two extra blocks, one at the start and one at the end and there will be one element in each of these blocks. The upper bound of memory transfers for a flat array is then $\lceil N/B \rceil + 1$.

2.4.2 Eytzinger Layout

One cache-efficient data structure which utilizes the flatness property to arrange elements in a contiguous block of memory for fast binary search access is the Eytzinger layout. In this layout, elements are ordered as a left-to-right breadth-first ordering of a binary tree, see Figure 2.6. The Eytzinger layout provides easy traversal from root to nodes through the offsets of the children. For a node at index i, the respective children are at index $2i + 1$ and $2i + 2$. Khuong et al. argue that the memory transfers are roughly the same as regular binary search, with the added benefit of the first C elements of the first $log(C)$ levels of the binary tree [16]. The number of memory transfers should then be roughly $log(N) - log(C)$. The number of elements C is dependent on the block size B and how large each element is.

Relying on arithmetic to find children is also known as the implicit pointer approach, using explicit pointers to the children would require extra space (and more memory transfers) in each node but would remove the overhead of computing the address. Here, the address computation is so simple it is not considered to be a bottleneck. The cache efficiency arises from the access pattern since each element to be compared is placed contiguously in memory.

2.4.3 van Emde Boas layout

A slightly more sophisticated cache-efficient layout is the van Emde Boas layout (vEB). It is asymptotically optimal in the number of memory transfers. Specifically it incurrs at most $2 + 4log_B(n)$ memory transfers [13], [17] during element search. The construction is as in the Eytzinger layout with a binary search tree representing the n elements. The tree is then stored by recursively splitting the tree at a middle level, resulting in one top tree and several bottom trees. After the split, there are $\sqrt{n}$ bottom trees of roughly size $\sqrt{n}$ each as seen in Figure 2.7. The tree is then laid out in a flat array where the top tree is followed by the recursive bottom trees.

Splitting of the tree is the key to the cache efficiency and is handled by splitting the tree such that the bottom recursive subtrees each have heights that are a power of 2. Commonly, the hyper-ceiling function is defined for this purpose as $\lceil\lceil x \rceil\rceil = 2^{\lceil log_2\ x \rceil}$,

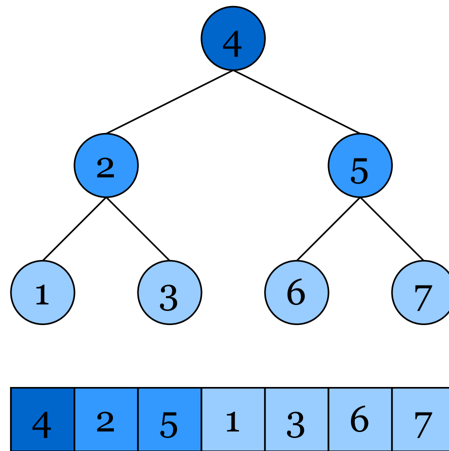


Figure 2.6: The binary search tree of a set of integers 1,. . . ,7. As binary search starts by examining the middle element, it is stored in the beginning with the next elements to be examined after it.

where x is an integer. The tree, of height h, is split such that the bottom subtrees obtain a height of $\lceil\lceil h/2\rceil\rceil$. The top recursive subtree is then left with a height of $h - \lceil\lceil h/2\rceil\rceil$ which may not be a power of two. If it is not a power of two, the top subtree is recursively split again.

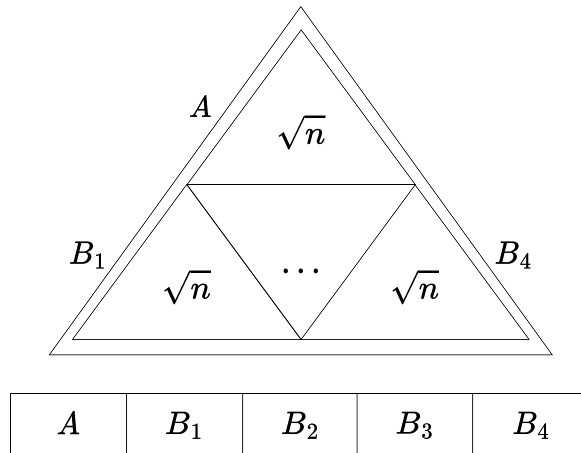


Figure 2.7: van Emde Boas layout for a set of n elements. The top subtree A has $\sqrt{n}$ elements and there are $\sqrt{n}$ subtrees each of size $\sqrt{n}$.

2.4.4 B-tree layout

By utilizing the van Emde Boas layout it is possible to create a cache-aware B-tree layout. In this layout, nodes are stored such that searching and updating the tree is done in an optimal number of memory transfers. This type of B-tree has been described by Arge et al. and supports updates (insert and delete) in amortized $\Theta(1 + log_{B+1}(N))$ memory transfers [18].

The B-tree can be seen as a search tree with nodes containing more than one element. Each node stores B elements in order with $B + 1$ children. As in both the Eytzinger

and vEB layouts, the tree is stored in the same order as a breadth-first search through the tree would encounter the nodes, see Figure 2.8. Contrary to both the Eytzinger and vEB layouts, the B-tree has a parameter, B, which is tuned according to the cache line size. This tuning allows memory transfers to be minimized as only one transfer is done for each node visited.

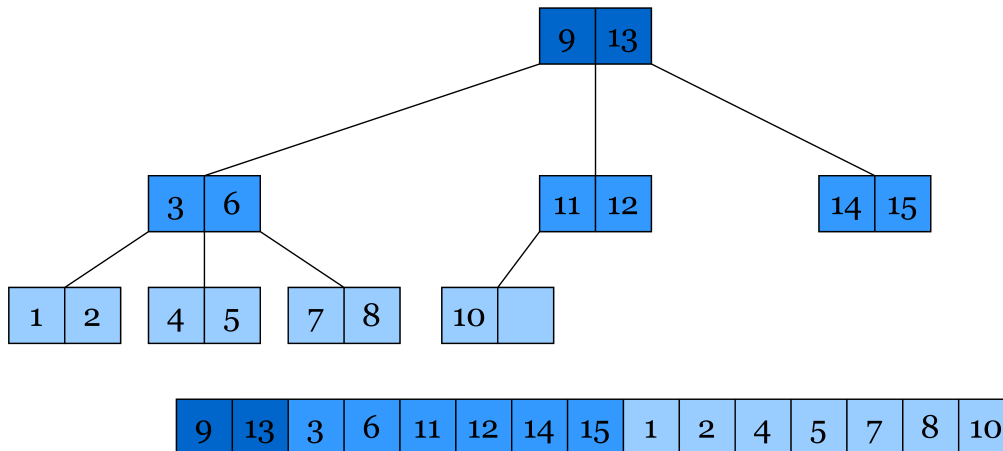


Figure 2.8: A B-tree layout for $B = 2$.

2.5 Prefetching

When constructing efficient algorithms, it is not only theoretical bounds that limit execution time and data transfers. When data is accessed in certain predictable patterns, hardware in the CPU may sometimes be able to predict the next addresses to be accessed by a program. For example, traversing a contiguous block of memory, such as an array, is a very typical procedure in programming and it is also an example of an easily predictable data access pattern. The hardware prefetcher sees the accesses, predicts the next addresses, and loads these into the cache before data is referenced, which can directly serve the memory requests without a cache miss.

Sometimes the hardware prefetcher can not predict the pattern since accesses seem random or follow too complicated patterns. Typical examples are hash map access and binary searching. The former is caused by good hash functions that place data uniformly and thus are impossible to predict. The latter may not be random but appear random to the hardware prefetcher, continuously accessing halves of an array. Prefetching is however not limited to hardware, it is possible to use software prefetching which can be done by so-called compiler intrinsics. These intrinsics are inserted in source code and are compiled into instructions that perform certain hardware-oriented functions, for example, fetch specific memory addresses. The typical example is to speed up binary search by inserting these instructions. Listing 2 gives an example of binary search with included prefetch instructions for the Gnu Compiler Collection (GCC) compiler.

The GCC compiler intrinsic `__builtin_prefetch` takes three arguments, an address, read or write hint, and temporal hint. The address specifies from where a

```
int find(const int elem, const int *array, const int size) {
    int low = 0;
    int high = size - 1;
    while (low <= high) {
        int mid = (low + high) / 2;
        __builtin_prefetch (&array[(mid + 1 + high)/2], 0, 1);
        __builtin_prefetch (&array[(low + mid - 1)/2], 0, 1);
        if (array[mid] < elem) {
            low = mid + 1;
        } else if (array[mid] > elem) {
            high = mid - 1;
        } else {
            return array[mid];
        }
    }
    return -1;
}
```

Listing 2: Classic binary search with added prefetch intrinsics for the two possible data accesses. The CPU will fetch the two elements that can be compared during the next iteration.

cache line will be loaded. The read and write hint tells the CPU that either a read or a write will be done to this address, most architectures ignore this hint. The temporal hint is between 0-3 and hints at how often the adress will be accessed. If the temporal hint is high, 3, memory will be used often, and 0 implies almost no near-time usage. This could be used to load the data into different levels of cache but is most often ignored on modern architectures and the behavior is highly dependent on hardware.

2.6 Vectorization

It is common in programming that the same operation should be applied to a collection of data, for example normalizing vectors in graphics. Modern CPUs support such an operation using Single Instruction, Multiple Data (SIMD) instructions. It is implemented in hardware by large registers capable of storing more bits than the regular types occupy. Today, there are CPUs that support instructions operating on 128 bits up to 512 bits. Usually, they can be configured to hold different-sized data, for example, the 256-bit SIMD instructions can typically operate on 8 integers (of 4 bytes each) or 4 double-precision floating point values (of 4 bytes each).

Vectorization can often be handled by the compiler when using optimization flags. Vectorization can potentially come with some tradeoffs. Data must be aligned in memory such that its address is divisible by a power of 2. SIMD instructions can possibly be slightly slower to execute in comparison to scalar instructions and may

slow down the CPU pipeline. On the other hand, if the execution time per data access is significantly reduced it leaves less time for the hardware prefetcher to bring in data from main memory, possibly increasing the load on the memory system and causing more cache misses.

2.7 Ahmdal's law

To accurately determine how much a program may be sped up by parallelization, Ahmdal's law provides an upper bound given timing measurements of the program [19]. The law is given as

$$S(s) = \frac{1}{(1-p) + \frac{p}{s}} \tag{2.6}$$

where s is the speedup of a parallelized task and p is the original proportional time of the program which has been given the speedup s. $S(s)$ then provides the maximal theoretical speedup possible. Letting s tend to infinity (supply infinite access to resources) the speedup is only affected by the serial part of the program, thus reaching an upper limit to the gain of more execution resources. Figure 2.9 show the possible speedup given different sizes of the parallelizable section. Note how the parallel section must be above 90% to keep scaling when using more than 16 cores. As modern high-performance clusters use well up to 64 cores and beyond, the parallelizable section should be more than 95% if a program need to scale.

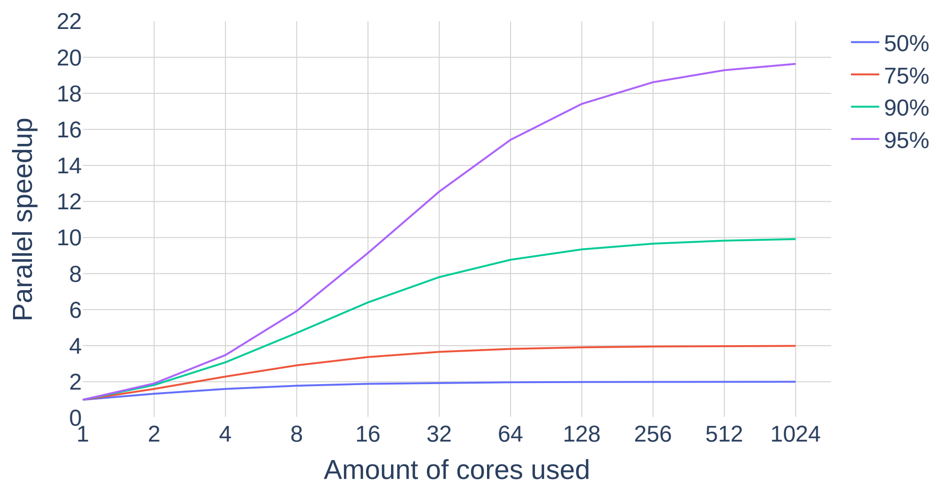


Figure 2.9: Visualization of Ahmdals' law for different sizes of parallelizable section.

3

Methods

This chapter describes implementation details, with special attention to actual VLMC data aspects and the data structures to represent VLMCs.

3.1 Input data

The input data, or k-mer distributions, are presented to motivate data structure and algorithm decisions.

3.1.1 Data sets

The execution time of distance computation is dependent on some key factors, namely the DNA sequences length, the number of sequences, and the parameters used when creating the VLMCs, see section 2.1. The data sets chosen are the NCBI Virus (Virus) data set[1], the NIH human genome assembly (Human) dataset[2], the EnsemblPlants Zea mays (Corn) dataset[3] and the Ensembl Meleagris gallopavo (Turkey) dataset[4]. The virus dataset is made up of 15 446 sequences, while Human, Corn, and Turkey have 24, 10, and 33 chromosomes respectively. In the Virus data set each VLMC is constructed from each genome, while in the remaining data sets, a VLMC is constructed from each chromosome.

The individual virus genomes are generally small, while the turkey, human and corn genomes are increasingly larger in size. The tables 3.2, 3.3, 3.4 and 3.5 show the average size of the VLMCs created from the different data sets. Table 3.1 shows some general statistics on the data sets. The data sets were chosen to investigate the performance for large numbers of sequences in distance computation (Virus) and how the algorithm performs with much larger-sized VLMCs with greater k-mer distribution differences (other data sets).

[1] https://www.ncbi.nlm.nih.gov/labs/virus/vssi/#/virus?SeqType_s=Nucleotide&SourceDB_s=RefSeq

[2] https://www.ncbi.nlm.nih.gov/data-hub/genome/GCF_009914755.1/

[3] https://plants.ensembl.org/Zea_mays/Info/Index

[4] http://mart.ensembl.org/Meleagris_gallopavo/Info/Index

Data set	GC-content	Percent unidentifiable
Virus	45.8 %	0.01 %
Human	41.2 %	0.00 %
Corn	46.8 %	0.17 %
Turkey	44.3 %	3.14 %

Table 3.1: The average GC-content and amount of characters that are unidentifiable (The unidentifiable characters are ignored during the creation of the VLMC).

	Avg. file size (MB)	Avg. nr. k-mers	Threshold	Min-count	Max-depth
Small	0.0098	110	3.9075	9	6
Medium	0.0182	205	3.0	6	8
Large	0.0425	478	2.0	3	10

Table 3.2: Characteristics of the virus genome data sets.

	Avg. file size (MB)	Avg. nr. k-mers	Threshold	Min-count	Max-depth
Small	0.483	5 430	3.9075	9	6
Medium	36.0	409 083	3.9075	12	10
Large	86.0	970 942	3.9075	15	15

Table 3.3: Characteristics of the human genome data sets.

	Avg. file size (MB)	Avg. nr. k-mers	Threshold	Min-count	Max-depth
Small	0.486	5 456	3.9075	9	6
Medium	103.0	1 167 401	3.9075	12	10
Large	428.0	4 816 541	3.9075	15	15

Table 3.4: Characteristics of the corn genome data sets.

	Avg. file size (MB)	Avg. nr. k-mers	Threshold	Min-count	Max-depth
Small	0.361	4 058	3.9075	9	6
Medium	6.9	77 598	3.9075	12	10
Large	14.6	164 344	3.9075	15	15

Table 3.5: Characteristics of the turkey genome data sets.

3.1.2 VLMC inspection

The most important characteristic of the VLMCs created will be the frequency of matching k-mers. Finding the intersection, C in Equation 2.3, can be achieved by either scanning or searching through a VLMC pair. Depending on where the shared elements are placed and the size of the intersection, both scanning and searching could yield very different execution times.

A simple observation is that shorter k-mers occur at a greater frequency than longer k-mers, see Figure 3.1, as the number of possible words increases exponentially. Hence, when comparing VLMCs with a larger max-depth many long k-mers will be unique to each VLMC.

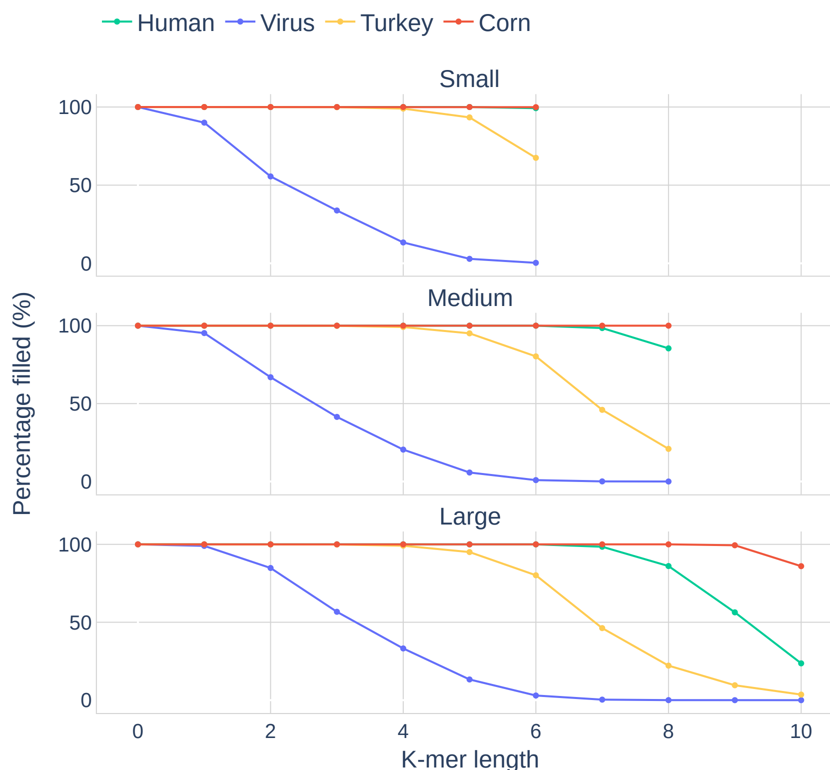


Figure 3.1: Average percentage of k-mers included for each k and data set. Each k-mer length has, for the length n, 4^n possible k-mers.

Comparing the k-mer relationships between data sets, Figure 3.2 shows the average amount of k-mers shared between data sets. The intersections were obtained by placing two pointers at the start of two sorted lists of k-mers from two VLMCs. If the k-mers at the pointers matched the hits were incremented, otherwise the pointer to the smaller of the k-mers was incremented together with the misses counter. Lastly, when one of the pointers reached the end of its list, the skipped counter was set to the size of the remaining list.

Considering 3.1 and 3.2 together, data sets that contained a smaller percentage of all possible k-mers in Figure 3.1 also share a lesser percentage of k-mers in Figure

3.2. Predictably, the data sets also share the most k-mers with themselves. However, the most important information is the number of misses generated when comparing the Human, Turkey, and Corn data sets. It can be conjectured that some data sets will be faster to iterate over rather than search through, such as when computing the distance from Corn to Corn. However, other data sets produce a greater than 80 % miss rate, possibly warranting searching as the better option.

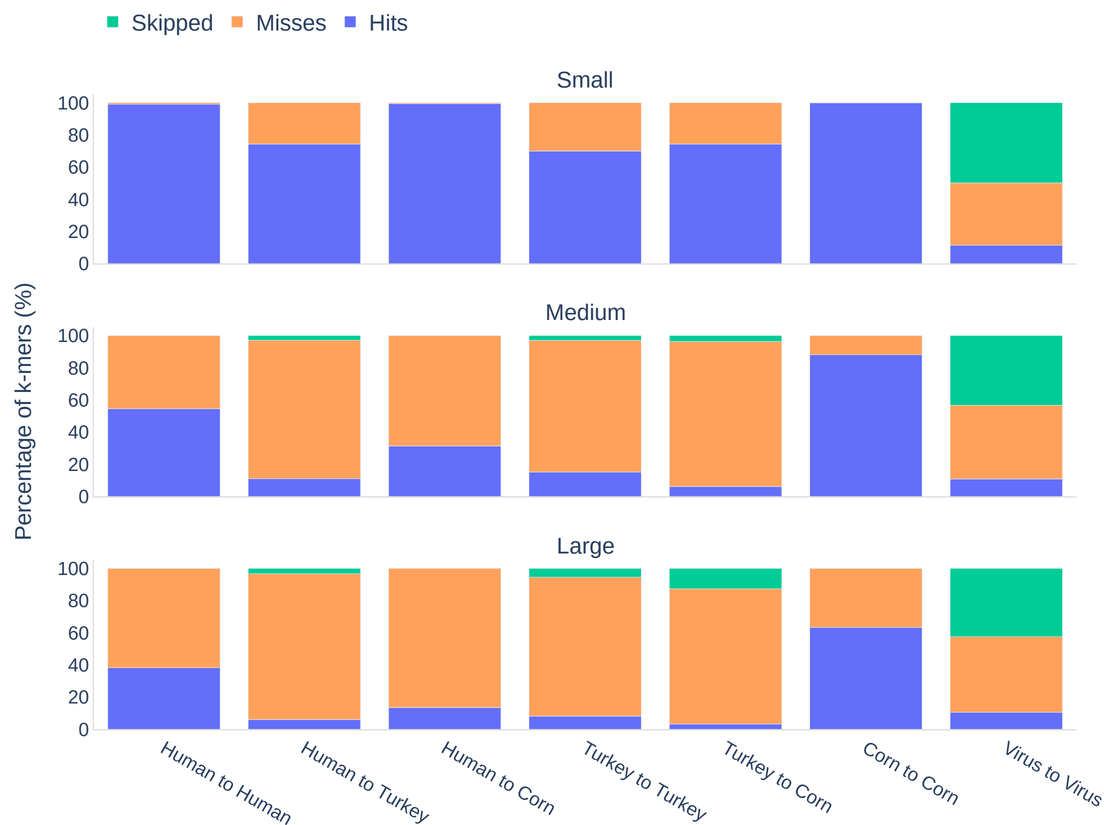


Figure 3.2: Percentage of hits, misses, and ignored k-mers on average when comparing VLMCs from different datasets.

It is not only the pure miss rate that will affect the performance of iterating to find the intersection. The number of misses occurring in succession will also be an important factor. For example, if the misses always occur every fourth k-mer, we will have a hit rate of 25 % but searching for the next hit won't be faster than simply incrementing a pointer and checking the value 3 times in succession.

To visualize these differences between data sets for the successive hits and misses, Figure 3.3 shows the log of the average amount of misses in a row for the different Large datasets. Similar to the previous plot the process of finding matches and mismatches was done through iteration over two sorted lists of k-mers for every VLMC pair. Each time one percentage of the lists had been iterated through, the average amount of misses for that percentage was accounted for.

Figure 3.3, again suggests that the algorithm will perform better by finding the intersection through search rather than incrementing for some data sets. For example,

when comparing the human to turkey data sets there are almost constantly more than 1000 misses in a row. There is also a clear difference between the number of misses in a row at the beginning and end of the VLMCs. This is to be expected since Figure 3.1 shows that the shorter k-mers often appear in all data sets, while the opposite is true for the longer k-mers.

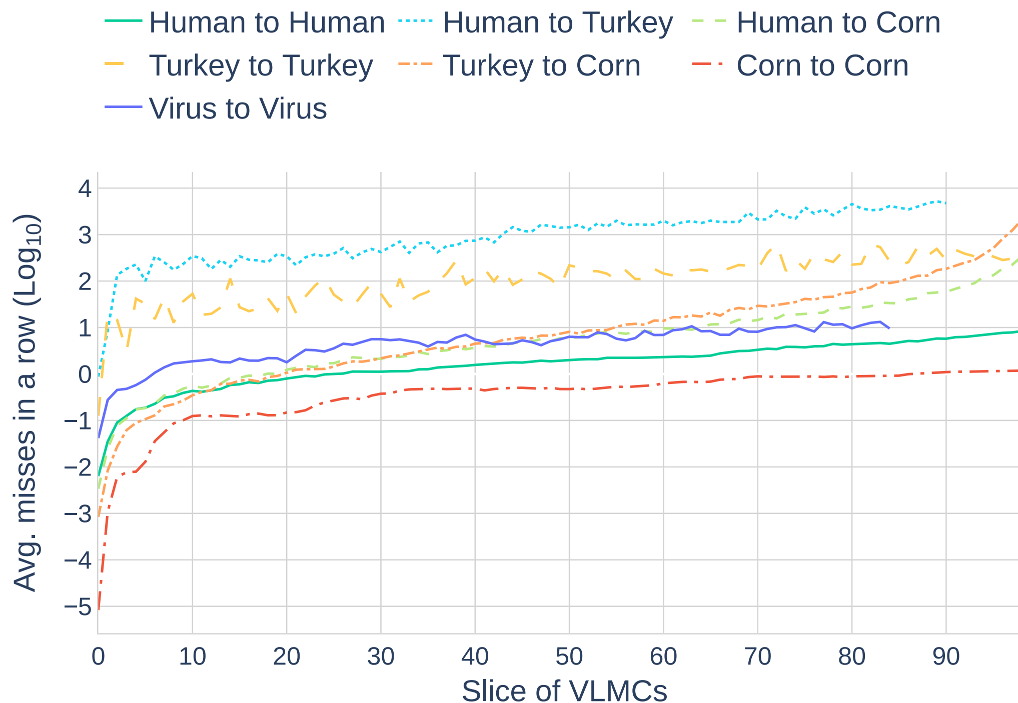


Figure 3.3: The log 10 of average misses in a row over slices of k-mers $\frac{n}{100}$ large from the VLMCs compared.

3.2 VLMC data structures

This section describes the implemented data structures for representing the VLMC as well as k-mer. Only basic familiarity with the C++ language is expected and most terms should be familiar.

To compute the dissimilarity of two VLMCs, k-mers from both VLMCs are compared and the dissimilarity is continuously updated. To compare all k-mers, the data structure needs to retrieve all k-mers in the intersection of the VLMCs, the set C described in section 2.1.4. As such, the data structure used to represent a VLMC only need to support fast insert and find operations. They can be considered to be static, no delete operation is used to compute the dissimilarity. These structures containing VLMCs will be referred to as VLMC containers.

3.2.1 *k*-mer representation

The *k*-mer in the new implementation, from now on referred to as read-in *k*-mer, uses a subset of the fields the *k*-mer used in the construction of the VLMC (that will be loaded from disk). It is represented as a `struct` with one integer field followed by one `std::array` of double point precision numbers, containing the next character probabilities. The integer field is the unique id of the *k*-mer. The integer representation is the corresponding value a sequence of characters generates, specified by placement and character value. Note that this is not related to the string ordering of a *k*-mer word. For example, the *k*-mer "G" has 3 as representation, and *k*-mer "AT" has representation 8.

In total, the `struct` covers 20 bytes when using `float` and 40 bytes when using `double` as the type of next-character probabilities. However, `float` produced an error up to 1E^{-4} between VLMC distances in comparison to PST (which also uses `double`).

As most modern computers use 64 bytes as cache-line size, almost two *k*-kmers can be stored in a single line when using the `double` type. This is beneficial when *k*-mers are stored in a flat structure since accessing the integer representation of the second *k*-mer in the cache line is always a cache hit. Further, integer comparisons are faster than the previous bit-wise comparison of strings where bit representations were shifted and compared.

Another approach was tested to ensure performance. The Eigen library's[5] double and float arrays were used to represent the next-character probabilities as these provide automatic vectorization of products, sums and inverse square root. To support this vectorization the array need to be aligned in memory, but it causes the array to be extended to 32 bytes for `float`s and 64 for `double`s. The selected design thereby trades away SIMD for lower memory footprint.

3.2.2 Probabilistic Suffix Tree Hash Map

The previous implementation (PST) use a hash map approach for VLMCs. The hash map is given by the Robin Hood library [20]. Each *k*-mer is hashed on its string value, e.g. "AGCT" as Robin Hood is optimized to hash on strings. During the distance computation of two VLMCs, one of the two is selected as the main. For each *k*-mer in the main VLMCs' hash map, a find-call is issued on the other VLMCs' hash map. If the *k*-mer is found, the distance metric is updated using the new information else it is skipped, see Listing 3. Comparison is also based on comparing the binary representation of the *k* letter word.

3.2.3 Hash Map

As the current implementation uses a hash map, a Hash map container is also implemented using the read-in *k*-mer representation, see Section 3.2.1. The hash map is given by the Ankerl:unorder_dense library [21], a newer implementation by

[5] `https://eigen.tuxfamily.org/index.php?title=Main_Page`

```
for kmer_string_representation, kmer in mainVLMC:
    res = otherVLMC.find(kmer_string_representation)
    if res == kmer_string_representation:
        Update distance metric using kmer and res...
```

Listing 3: Psedo code for finding the intersection between two VLMCs using a hash-map.

the same author as the Robin Hood library [20] (utilized by PST). Finding the intersection with this container is done as in Listing 3. Compared to the PST hash map the hashing is done on the integer representation instead of the string representation of the k-mers.

This container was created as there are multiple changes to the PST algorithm beyond data structure changes. Implementing a container with similar functionality makes it possible to analyze how specific data structures affect performance.

3.2.4 Sorted vector

The sorted vector approach uses one sorted `std::vector` for each VLMC to store all k-mers. Finding the next present k-mer is fast ($O(1)$) and finding a specific k-mer can be done in $O(log(n))$ time, with n as the number of k-mers. As the vector is stored contiguously in memory, iteration is expected to be cache efficient as the cache will be loaded with upcoming k-mers. This is both due to the access pattern being predictable for the hardware prefetcher and due to cache lines being loaded with relevant k-mers, see Section 2.5 and 2.4.1.

During the construction of the sorted vector, k-mers are continuously added to the end of an unsorted vector. When all elements have been inserted, the vector is sorted with the standard C++ sort for containers. The sort is parallelizable via the `par` or `par_unseq` execution policies. As ordering of data can impact the performance of sorting, k-mer integer representations, and their ordering are presented in Figures 3.4 and 3.5. The k-mers are always outputted partially sorted from the construction of VLMCs. This can be clearly seen in the case of the human sequence (Figure 3.5).

The intersection of k-mers can be found by concurrent iteration from the start to the end of the two VLMCs. To find the next shared k-mer, pointers to the sorted vectors are increased until the integer representations are matched. This means increasing the pointer for the vector with the smallest integer representation. If both integer representations are equal both indexes are increased by one (and the dissimilarity is updated). See listing 9 in Appendix A for source code.

3.2.5 Eytzinger layout

When the number of shared k-mers is low, the following search structures may prove useful as less time is spent referencing k-mers not present in the intersection. The Eytzinger layout, presented in section 2.4.2, is constructed by first sorting the k-

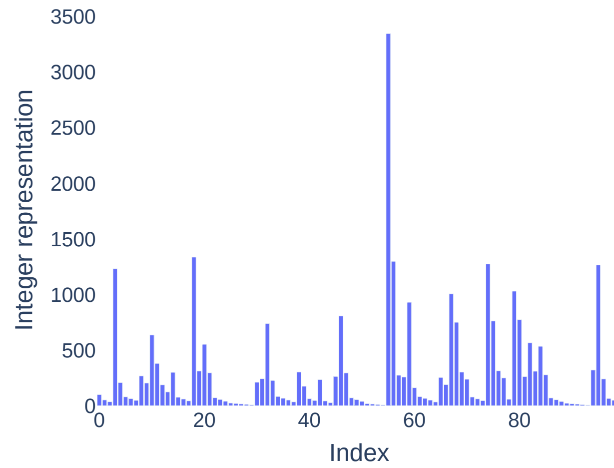


Figure 3.4: Ordering of the first 100 k-mers when loading a Virus VLMC from disk. Index 0 indicates the first item of the unsorted vector.

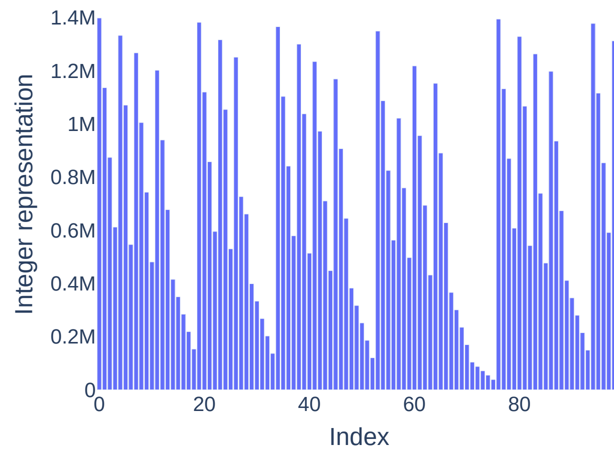


Figure 3.5: Ordering of the first 100 k-mers when loading a human VLMC from disk. Index 0 indicates the first item of the vector. 5 major subsets can be seen between index 0-19, 20-39, etc. with further sorted subsets within.

mers, followed by a recursive copy step which places the k-mers in a breadth-first search order, it can be seen in the listing below, listing 4.

```
1 int construct(int i = 0, int k = 1) {
2     if (k <= size) {
3         i = Ey_array::construct(i, 2 * k);
4         ey_sorted_kmers[k] = kmer_from[i++];
5         i = Ey_array::construct(i, 2 * k + 1);
6     }
7     return i;
8 }
```

Listing 4: Copying k-mers from the sorted vector `kmer_from` to a vector `ey_sorted_kmers`, which will have the Eytzinger layout.

Searching in the Eytzinger layout follows the algorithm introduced by Khuong et al. [16] which is a branch-free implementation with prefetching and an aligned array. Khuong et al. found this implementation to work best out of all tested static search layouts (Eytzinger, vEB, B-tree). The implementation of the search algorithm is shown in Listing 5. Let x be the integer k-mer representation. As described in section 2.4.2, Eytzinger's search progress as a binary search, but with different indices. The index k is updated with $k := 2k$ if x is less than the k-mer at k, else the index is updated with $k := 2k + 1$ if x is larger then the k-mer at k. This can be seen in line 5 in Listing 5. The binary representation of k encodes the path of left and right turns taken. If the search does find a k-mer, all following k-mer integer representations will be greater than x, creating a sequence of right steps. This means searching for the index of a k-mer on a path that ends in a series of right steps, and at least one last right step. As the left and right steps are encoded as 0 and 1 respectively in k, the right steps to be discarded are in the least significant bits of k. GCCs (Gnu Compiler Collection) `__builtin_ffs` returns the $i + 1$ of the least significant 1 bit of its integer argument. Line 7 in Listing 5 shows how the last right steps are canceled by shifting k right by locating the final left step which occurred before the sequence of right steps.

The Eytzinger search is different from the other search structures as it must proceed through the entire list. The search does not return on the found element. Khuong et al. found this implementation to perform best for objects up to 16 bytes, and this is the reason it was chosen in this project (where the k-mers are 32 bytes). A branchy implementation could however be faster, as they were not tested and presented some promise as branchy code leads to data beeing prefetched.

3.2.6 van Emde Boas layout

Following the Eytzinger layout is the slightly more complex van Emde Boas layout. As described in section 2.4.3 it is a cache-oblivious search structure with asymptotically optimal memory transfers without parameters to tune. The implementation follows that of Khuong et al. [16], which uses implicit pointers via a small lookup

```
int search(int x) {
    int k = 1;
    while (k <= size) {
        __builtin_prefetch(ey_sorted_kmers.data() + k * block_size);
        k = 2 * k + (ey_sorted_kmers[k] < x);
    }
    k >>= __builtin_ffs(~k);
    return k;
}
```

Listing 5: Searching in the Eytzinger layout. Line 4 prefetch the next block of k-mers which will be accessed in the next iteration. Note that only one prefetch instruction is needed in comparison to two instructions of classical binary search (see Section 2.5).

table, and a current path from the root to a node, both used to compute the child nodes' placements during the search. The size of the lookup table is the height of the vEB tree and the path is encoded by 0 and 1 bits to represent left and right turns in the path. The formula to compute the pointers follows that of Bender et. al. [18], and is found during the recursive splitting of the tree in the construction phase. Note that it is this recursive splitting that forms the requirement of the lookup table as each split results in uneven heights and as such it is the layout for which known offsets can not be used. The construction implementation is given in Appendix A, Listing 12.

Searching the vEB layout proceeds similarly to binary search. The element searched for is compared to the current element, and a left or right path is recorded accordingly. Given the current path, the next element to compare against is found by utilizing the lookup table which holds the size of the subtrees at different depths, the path, and the equation by Bender et al. [18]. Search implementation is given in Appendix A, Listing 13.

3.2.7 B-tree

In the B-tree layout, k-mers are layout in a breadth-first-search order, similar to the Eytzinger layout, where the nodes of the tree store B k-mers. The B-tree is a cache-aware layout with B usually equal to the cache line size, as a cache line of 64 bytes is most common, and it is the size of the benchmark machines, B is set to 2 as each read-in k-mer has a size of 32 bytes.

Construction follows very similarly to the previous search structures as it is a recursive procedure for which the source code can be found in Appendix A, Listing 10. It follows a breath-first-search order of the k-mers present in a sorted list.

Similar to the Eytzinger layout, the search can utilize the fact that all child nodes are at pre-defined offsets. Consider for example the nodes $i, i+1, \ldots, i+B-1$. The jth child, where $j \in \{0, \ldots, B\}$, is stored at $f(i,j), \ldots, f(i,j)+B-1$ where $f(i,j) =$

$i(B+1)+(j+1)B$. With these known offsets, the search proceeds by an inner binary search for a k-mer in each node, and while the k-mer is not found, we proceed to the node where it should be placed. We found an unrolled (loops transformed into repeating code), branch-free implementation with prefetching to work best, with little variation between naivë, only unrolled and unrolled, and branch-free versions with and without prefetching. The implementation is listed in Appendix A, Listing 11.

The downside of the implementation is that B is set to 2. The main benefit of B-trees is when B is slightly larger, otherwise, there is little difference to regular binary search (or in this case the Eytzinger layout). A proper parameter sweep of values up to 16 or similar would have been preferred but due to time, B was set to match the cache-line size, which Khuong et al. [16] also found optimal. Further, as B is only 2, binary search is likely not optimal but was kept as it is unrolled. The unrolling of binary search with two elements is simply two element comparisons, which is the same as the alternative of scanning the block with B elements. Further, implementation supports an arbitrary B which can be tuned.

3.2.8 Sorted Block Search

To combine the strength of iteration and search, a new data structure was developed, called Sorted Block Search (SBS). SBS is an extension of the Sorted vector, and it utilizes a summary vector in addition to a sorted vector. The summary vector is created by taking every $\log_2 n$'th value and index from the sorted vector. This value will then be the max value of each block of $\log_2 n$ values, see Figure 3.6. Note that iteration of the sorted vector and the summary structure is cache-oblivious, see Section 2.4.1. The memory access pattern in both structures is predictable by the hardware prefetcher (Section 2.5), however, switching between the structures could reduce the prefetcher's ability as switches can not be predicted.

Finding the intersection of two VLMCs starts the same way as Sorted vector by iterating pointers to each vector and updating the dissimilarity. However, the iteration process can be sped up by skipping forward if the k-mer is not in the current block. Searching in the summary structure is done through iteration, where the index of the current block is known, hence iteration starts from that block. The Eytzinger layout and binary search were explored as an alternative but proved slower.

The most notable design choice in SBS is the size of the summary structure. Multiple sizes were tested; every $\sqrt{n}$th value and $\log_2 n$'th value with different constants, but the $\log_2 n$ solution proved to be over all the fastest.

3.2.9 K-mer buckets

Utilizing the fact that only matching pairs of k-mers in VLMCs contribute to the overall distance, this method aims to remove the overhead of finding matching pairs. This is achieved by storing k-mers with the same representation in buckets, with an identifier of which VLMC the k-mer belongs to.

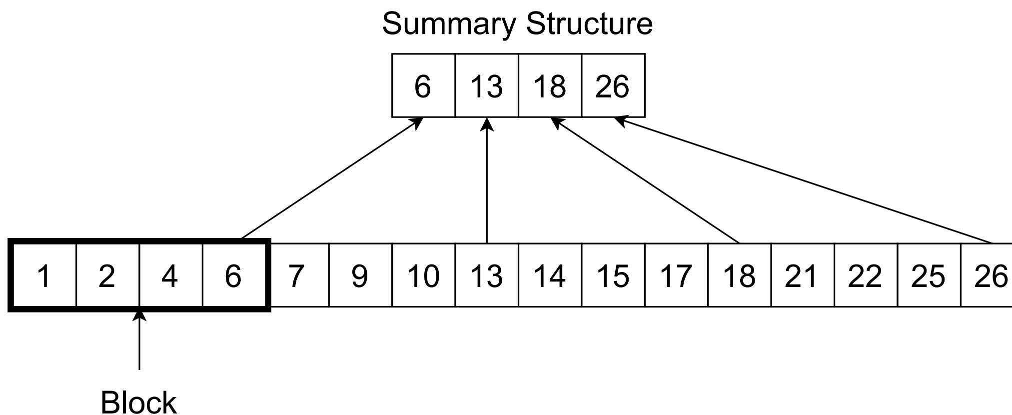


Figure 3.6: Layout of the Sorted Block Skip data structure.

When the VLMCs are loaded from disk, each k-mer is appended to an `std::array` in a `std::unordered_map`, a typical hash map with $O(1)$ average insertion and search.

For distance computation, three matrices are kept track of, one for the dot product and two for the normalization factor of Equation 2.2. For each array in the hash map, the dot product and normalization are calculated for each k-mer pair and added to the matrixes, where the position in the matrixes is given by the VLMC identifiers. Finally, the matrixes are processed following Eq. 2.3 and 2.4. Parallelization of this step is done over the buckets, the buckets are evenly split over all cores. As the buckets of the lower k-mer representations are expected to contain more k-mers, multiple solutions to split the buckets better were tried. Among these were thread pools and splitting exponentially. The exponential split was based on giving the last half of the bucket to one core, and the first half was recursively split again and distributed. Neither solution provided significant improvements in execution time.

3.3 Algorithmic Implementations

An overview of the VLMC construction by Gustafsson et al. [6] is given in addition to implementations that incorporate the data structures presented in Section 3.2. An overview of the steps of the procedure to compute the distance between DNA sequences represented as *.fasta*-files, depending on which data structure is used for the VLMC is given in Figure 3.7.

3.3.1 Building VLMCs

The creation of VLMCs from DNA sequences is described in depth in [6]. Briefly, the relevant parameters when creating a VLMC are *min-count*, *max-depth*, and *threshold*. First, a set of k-mers is created where max-depth specifies the maximum k used, i.e. if max-depth is 4 then the set of k-mers will be created with k equal to 1,2,3 and 4, given that they exist in the genomic sequence. Next, identical k-mers in the set of k-mers are counted, if the count is less than the min-count, the k-mers are discarded.

```
n = len(dir1)
m = len(dir2)

unordered_map_left = createUnorderedMap(dir1)
unordered_map_right = createUnorderedMap(dir2)

dot_product = matrix(n,m)
left_norm = matrix(n,m)
right_norm = matrix(n,m)

for bucket_left in unordered_map_left:
    bucket_right = unordered_map_right.find(bucket_left.integer_representation)
    for kmer_left, id_left in bucket_left:
        for kmer_right, id_right in bucket_right:
            dot_product[id_left, id_right] = f(kmer_left, kmer_right)
            left_norm[id_left, id_right] = f(kmer_left, kmer_right)
            right_norm[id_left, id_right] = f(kmer_left, kmer_right)

# Normalize the matrices ...

def CreateUnorderedMap(directory):
    id = 0
    for VLMC in directory:
        for kmer in VLMC:
            unordered_map.find(kmer.integer_rep).append({kmer, id})
        id++
    return unordered_map

def f(kmer1, kmer2):
    apply Equation 2.3 or 2.2 (multiply vector probabilities)
```

Listing 6: Pseudo code for the k-mer buckets implementation. Given two directories `dir1` and `dir2` evaluate the numerator of Equation 2.3 and Equation 2.2.

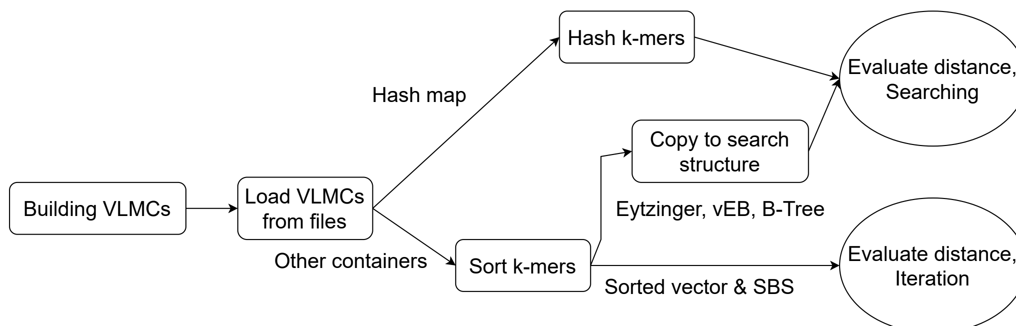


Figure 3.7: Steps of the procedure to compute the distance of DNA sequences.

Lastly, the remaining k-mers are filtered based on their information, this is adjusted with the threshold parameter. For example, if the next-character probabilities for "CAG" is similar enough to "AG", "CAG" may be removed. For the generation of benchmarking data, lowering the min-count, increasing max-depth, or lowering the threshold increases the size of the VLMC.

3.3.2 Loading VLMCs to memory

Given a directory of VLMC files (`.bintree`), each consisting of a set of k-mers, all VLMCs are sequentially read from disk. The VLMCs are placed in an `std::vector` and for each VLMC, the k-mers are loaded into an `std::vector`. As each k-mer arrives, they are translated into read-in k-mers, see section 3.2.1.

In comparison to the Probabilistic Suffix Tree (PST), the next-character probabilities are adjusted according to the background order, see Eq. 2.1, during this loading phase. Previously this was done in each distance computation (when evaluating Eq. 2.4). Moving the adjustment to the loading phase reduces the number of times this operation is applied to $O(n)$ from $O(n^2)$. Further, as the background order is commonly small (0-2), only a small vector of values needs to be stored and applied to all k-mers, allowing the cache to keep all these values while adding k-mers.

Most containers (Sorted vector, SBS, Eytzinger, vEB, and B-tree) rely on the k-mers being sorted. During development, multiple sorting algorithms were tested. These included the STL *introsort*, the STL *stablesort*, and a library version of *timsort*. Introsort and timsort have a worst-case complexity of $O(n\ log(n))$, stablesort has $O(n\ log(n))$ complexity or $O(n\ log(n)^2)$ depending on memory availability. Briefly, all three sorting algorithms can benefit from partially sorted input data, which the k-mers are from inspection. From testing on both Virus and Human VLMCs, introsort performed best.

3.3.3 Computing distances between all VLMCs

The current implementation supports the distance computation either between all VLMCs in one directory or between two different directories. In both cases, a matrix containing the distance between the ith and jth VLMC is returned by the program, from here on referred to as the distance matrix. In PST, distance is computed by a nested for-loop for all VLMCs.

3.3.4 Matrix recursion

Consider the matrix transpose problem, see Section 2.3.1, where we apply a different function to each element instead of swapping the placement of pairs of elements. Let the elements of the matrix be pairs of VLMCs. The function we apply to all these pairs is then the d_v^* function from Equation 2.4. The base case is also changed, in the matrix transpose problem the base case is the two elements to be swapped, but now the base case is the single element (pair of VLMCs). This new version of the matrix transpose problem is what we will refer to as matrix recursion.

The splitting of a 4x4 matrix can be seen in Figure 3.8. Further, the order d_v^* is applied for a similar matrix is shown in Figure 3.9.

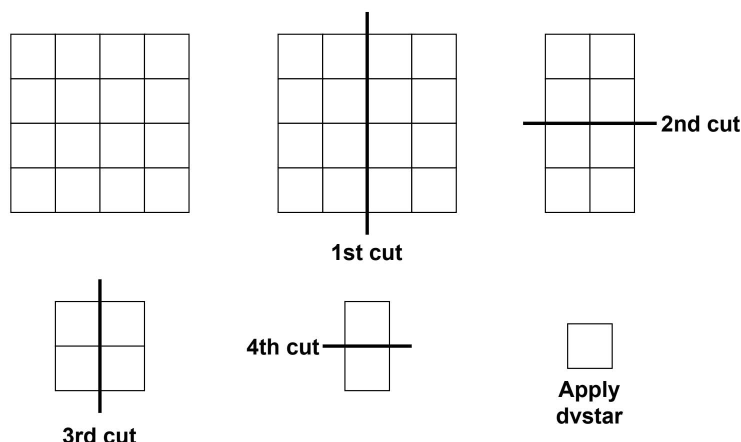


Figure 3.8: Vizualizing the recursive splitting of a 4x4 matrix. Note the new base case of one element (pair of VLMCs).

Order of evaluation

1	3	9	11
2	4	10	12
5	7	13	15
6	8	14	16

Figure 3.9: The execution order of the matrix recursion when evaluating a 4x4 matrix.

In the new implementation, the matrix recursion algorithm organizes the order of VLMC distances. As such the distances are computed by first dividing the distance matrix into blocks by halving the longest axis, see 7 for source code, and applying Equation 2.3 to both VLMCs.

Two important notes in the case with the single directory distance matrix are the 0-valued diagonal (the reflexive distance is 0) as well as the symmetric attribute. This provides a trivial optimization of only computing the distances of the upper triangle of the matrix, without the diagonal.

3.3.5 Computing the distance between two VLMCs

In PST, the dissimilarity metric is updated by iterating the k-mers in a hash map of one VLMC and finding the corresponding k-mer in the other. When a pair present in both are found, the next-character probabilities are updated to reflect the background order, see Eq. 2.1. For each VLMC pair three sums are tracked, the

dot product, left normalization, and right normalization. Once the next-character probabilities have been updated, the three sums are incremented by the sum of products and the sum of squares, respectively. This corresponds to Equation 2.2 and the numerator of Equation 2.3.

In the new implementation, the background-order no longer needs to be applied before updating the dissimilarity as it has already been done during loading. PST utilize some vector operations when updating the equations, see Listing 15 in Appendix A. When using Eigen vectors for the next-character probabilities, the update becomes shorter and completely vectorized, see Listing 14 in Appendix A. Due to the non-aligned read-in k-mers the new implementation can not use SIMD instructions however.

3.4 Parallelization

Parallelization is applied in the loading step, described in Section 3.3.2, and the distance computation between all VLMCs, described in Section 3.3.3.

When loading, the VLMCs are divided into equally sized groups loaded in parallel. During development, using more than 4 cores (on the Laptop, see 3.6) increased execution time as the memory subsystem could not service all memory requests in time. Hence, this part was capped to only parallelize on 4 cores.

For distance computation, there are two `std::vector`s of VLMCs. The largest of the two vectors is divided into as many groups as the number of cores specified, then each core is responsible for calculating the distance between one of the sub-groups and the smaller of the two vectors, see Figure 3.10. It is on this slice matrix recursion is applied. For this part, thread pooling was also tested, but showed no increase in performance or higher CPU usage to the former approach.

Since the distance computation is performed in $O(n^2)$ compared to loading which is done in $O(n)$, the vast majority of time should be spent on distance computation. In theory, the program should be infinitely parallelizable since neither of the parts runs sequentially. When measuring running time the program was deemed to be 99 % parallelizable at $n > 5000$, according to Ahmdal's law, see Section 2.7. In practice, this is always hard to achieve.

3.5 Downsampling VLMCs for speed

A possible approach to compute the dissimilarity is to relax the correctness constraint and allow some error in the dissimilarity. This allows coarser distances to be obtained, by not including all k-mers in each VLMC. By including fewer k-mers, the execution time and memory usage may be reduced.

Downsampling is done by ignoring a percentage of all k-mers. Other downsampling techniques were tried such as removing a percentage of the first or last k-mers or skipping every n'th k-mer, however, these proved to be very inaccurate. It is possible

Figure 3.10: Parallelization of the distance computation on 4 cores, with the two sets of VLMCs of size 7 and 12 respectively.

to create smaller VLMCs from the start by other parameters to build the VLMC but then a new construction would be necessary for greater accuracy. It is also possible to prune the VLMCs directly, but this is a computationally expensive operation. By downsampling, a larger VLMC is constructed once, and depending on the necessary accuracy, different rates of downsampling can be used.

3.6 Benchmarking

Benchmarking is done by computing the distances in a directory containing the described data sets and comparing different metrics against PST results. Statistics are gathered on Laptop (CPU: Intel(R) Core(TM) i5-10300H CPU @ 2.50GH) running Linux Mint 21 and kernel version 5.15.0-69-generic, an HPC (High-performance computing) cluster (CPU: Intel(R) Xeon(R) Gold 6126 CPU @ 2.60GHz) running Ubuntu 22.04.2 LTS and kernel version 5.15.0-67-generic and a regular Desktop (CPU: AMD Ryzen 7 5800X 8-Core Processor @ 3.3GHz) running Linux Mint 21 and kernel version 5.15.0-69-generic. Cache sizes and CPU configuration is provided in Table 3.6. Statistics are averaged over 3 runs.

	L1 (KB)	L2 (MB)	L3 (MB)	Cores	Threads
Laptop	128	1	8	4	8
HPC	32	1	20	24	48
Desktop	265	4	32	8	16

Table 3.6: Table of the benchmarking machine's hardware.

3.6.1 Perf diagnostics tool

Measurements are taken using the `perf` tool in Linux[6]. `perf` can be used mainly in two modes, `stat` or `record`. `stat` counts hardware events throughout the execution of a given program and outputs the results to standard error in readable text. These events are dependent on the system and different architectures. A few events are however defined in a global space, which is supported on all systems. These are the instructions, cycles, cache references, cache-misses, branches, branch-misses, context switches and kernel-, user-space- and total time. Cache references and cache misses refer to the last level cache-misses, since different systems have different levels of cache. The last level is the most important as it incurs the most overhead to miss a reference. On the benchmarking machines, there is also support for tracking the L1 data and instruction cache references and misses.

To track memory usage, `Valgrind` was used. The tool provides information on heap usage during the execution of a program, but only peak heap usage is presented.

Beyond the `stat` mode, `record` is useful to collect information on call-stack measurements. The call stack is the chain of functions called during execution and is combined with measuring events such as total time and cache-misses. This provides information on which functions take the most time or in which functions most cache-misses occur. Both were heavily utilized during development to find bottlenecks.

[6] `https://perf.wiki.kernel.org/index.php/Main_Page`

4

Results and Discussion

This chapter presents the differences between the current state-of-the-art Probabilistic Suffix Tree (PST) and new implementations in speedup and cache misses. Further, space usage and results regarding speedup compared to Ahmdal's law are given.

Throughout the benchmarking process, the container *kmer-bucket* (see Section 3.2.9) performed poorly on all metrics, hence it was excluded from the majority of graphs in the result section. Furthermore, measurements on the laptop generated results with very high deviation compared to the desktop and HPC, which explains the high variance in speedup.

4.1 Speedup

Speedup is defined as the ratio of the PST execution time to the new implementations' execution times. All new containers provide some speedup over PST, see Figures 4.1, 4.2, 4.3. In both the Medium and Large versions of the Human, Turkey, and Corn data sets, Sorted vector (see Section 3.2.4) and SBS (see Section 3.2.8) achieve the greatest speedup except for Turkey to Turkey on Laptop, see Figure 4.1. For the smaller versions of these data sets and with the Virus to Virus data set, the Hash map attain the most speedup.

For the Turkey to Turkey and Turkey to Corn data sets, both the iteration and search implementation perform equally well. In comparison, for the Human to Human and Corn to Corn data sets, the iteration methods, Sorted vector, and SBS, are clearly better than the search methods. The Human to Human and Corn to Corn data sets contain VLMCs with many shared k-mers whilst the Turkey to Turkey and Turkey to Corn has a lesser amount of shared k-mers as well as many more significant gaps between matching k-mers (see Section 3.1.2).

Further comparison to the Hash map also provides more information on how expensive sorting is. The Hash map is the only container that does not sort the k-mers in order but relies on constant (average) time inserts in the construction phase. When the Hash map is not faster than the sorted approaches, it could indicate that sorting in the construction phase pays off in execution time during distance computation due to the structure of the VLMCs. It could also indicate that the lookup time for the Hash map is slower than iterating and finding the next matching k-mer since

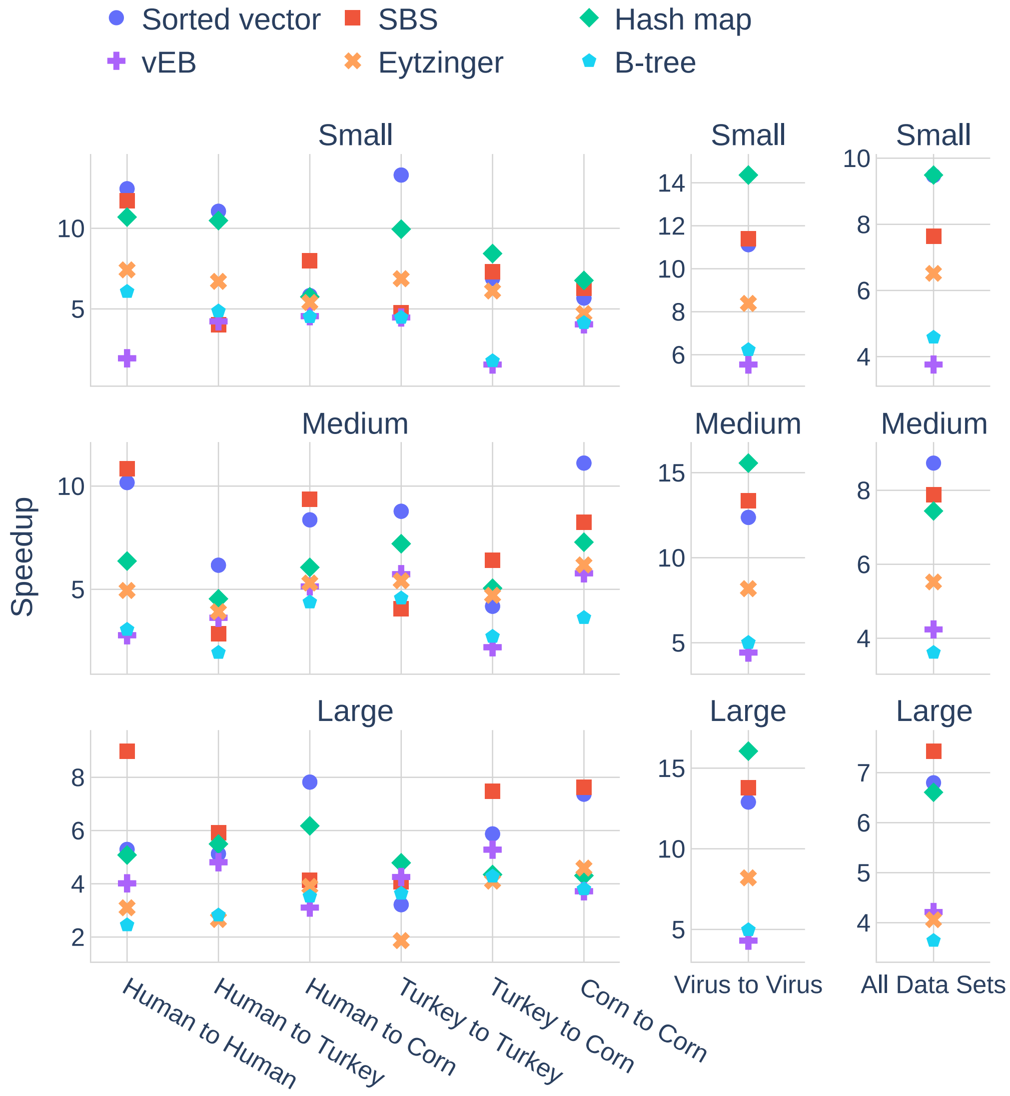


Figure 4.1: Laptop speedup versus PST for different containers and data sets.

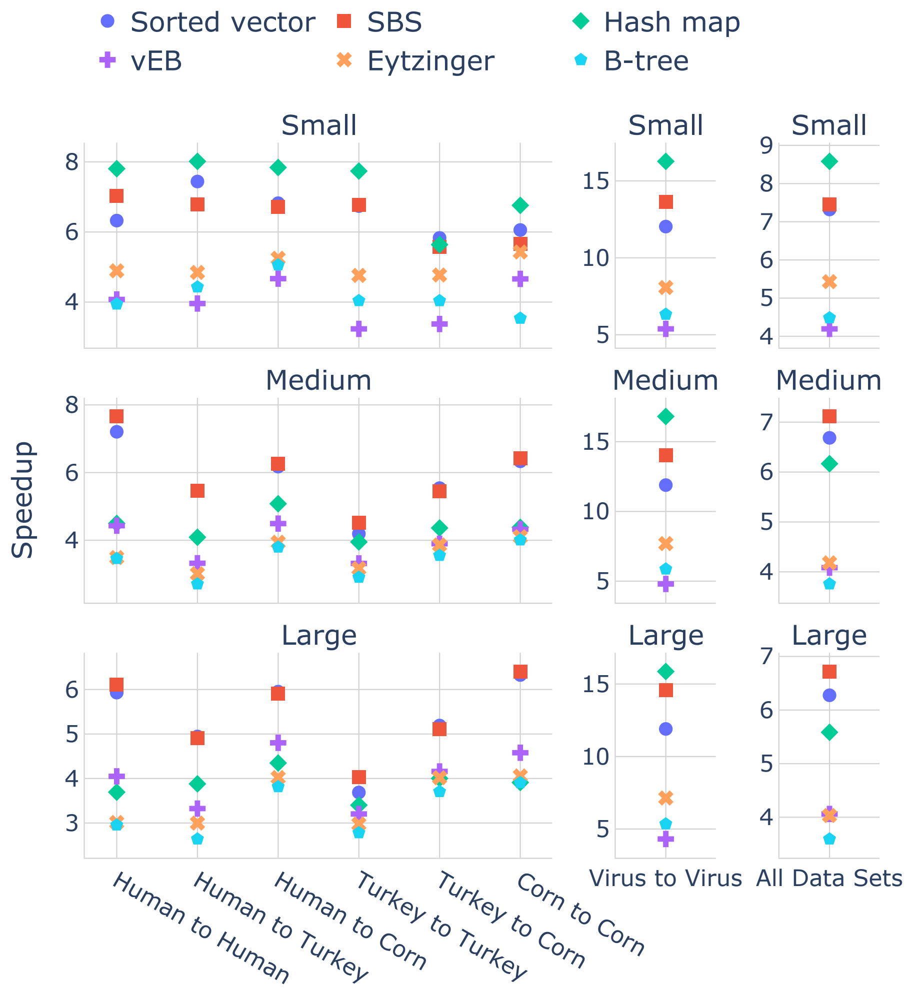


Figure 4.2: Desktop speedup versus PST for different containers and data sets.

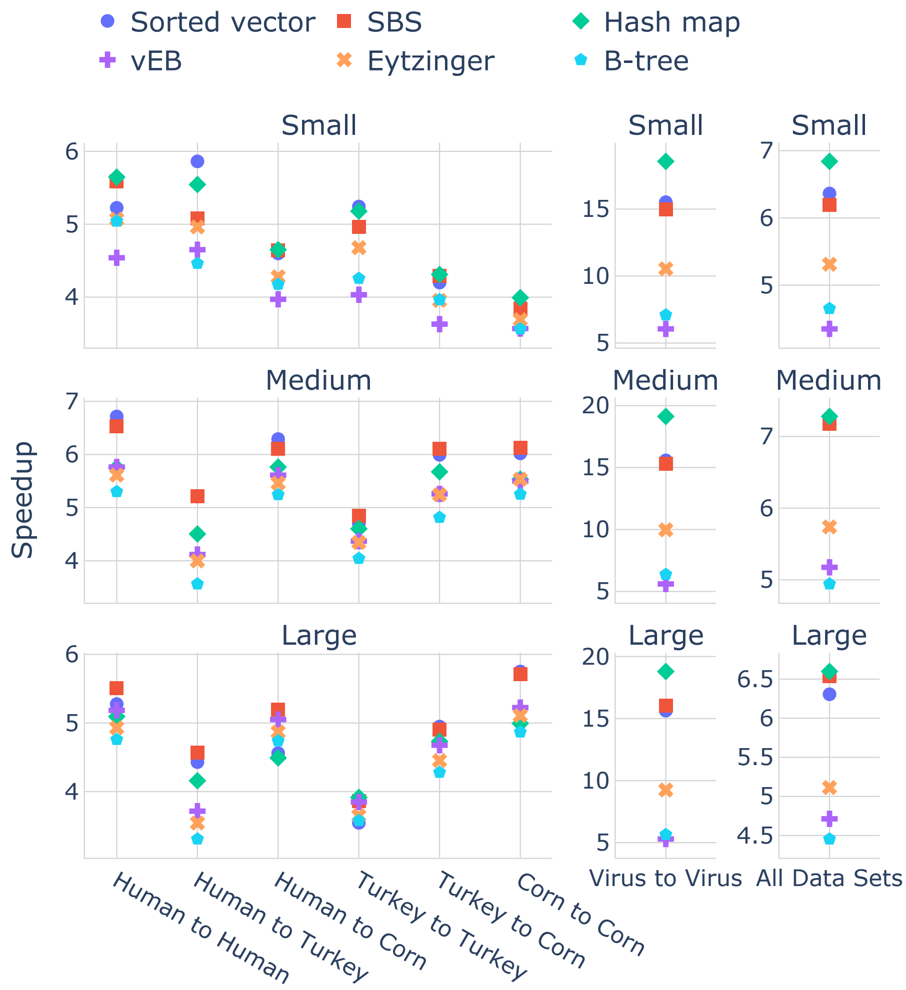


Figure 4.3: HPC speedup versus PST for different containers and data sets.

the map no longer fits into the cache.

Sorting is also a large topic in itself. There exist many fast sorting algorithms for strings or objects with strings as identifiers, for example, Radix sort, Bucket sort, and trie sorts. As the execution time is dominated by the distance computation and not the loading phase, this was not further investigated but is an interesting extension.

Another possible improvement was the use of vectorization. An Eigen-vector (from the Eigen Package [22]) was tested to represent the next-character probabilities in the k-mer, to utilize SIMD implementations, see Section 2.6. This resulted in the k-mer representation occupying 64 bytes instead of 40, since the Eigen representation automatically padded the size, making only one k-mer fit into the cache line. This improved the arithmetic speed, and a greater speedup was gained compared to PST on the smaller data sets (Virus to Virus and Small Human, Turkey, and Corn). On the other data sets the speedup was less than the presented results, creating a dilemma of which implementation to use, whether to improve the speedup on the smaller or the larger data sets, see Table B.1 in Appendix. We settled on using the smaller k-mer representation since this made the most sense for the cache-oblivious data structures.

To summarize, the choice of smaller `struct`s for k-mers is dependent on the size of the VLMCs that are used in distance computation. If the size is small enough, containing at most a few thousand k-mers, memory access is not the bottleneck in the new implementation, but rather the arithmetic speed, and thus Eigens vectors should be utilized. On the other hand, if much larger VLMCs are expected, a more compact representation is preferable since memory access becomes the bottleneck. This is dependent on hardware, the Desktop performed better for almost all data sets with Eigens vectors, and should be a consideration in future improvements.

4.1.1 Speedup to VLMC set size

Beyond speedup between data sets, the algorithm also scales differently to different numbers of VLMCs included in the distance computation. In Figure 4.4 the speedup to PST for different containers is presented at different set sizes of Virus VLMCs. At about 128 VLMCs and fewer in each set, the new implementations are in fact slower than PST. However, the slower execution times when comparing less than 128 VLMCs are relatively insignificant as can be seen in Figure 4.5. When including more VLMCs, the new implementations all provide some speedup. Notable is SBS, Hash map, and Sorted vector scaling up to 18 times faster whilst the three containers vEB, Eytzinger, and B-Tree with much less speedup.

The relatively slower execution time, when comparing less than 128 VLMC in Figure 4.4, is likely due to the loading phase, see Section 3.3.2. The major procedures done here are the conversion to the new, smaller read-in k-mer and the adjustment of the background probability. Sorting is also done in this phase for all containers except the Hash map, but the Hash map container also shows about the same slowdown.

The differences in speedup scaling of the containers could be due to the ineffective-

ness of finding the intersection of Virus VLMCs, as they have been shown to share many k-mers, see Section 3.1.2.

Table 4.1 and 4.2 show the execution time of the different implementations on Laptop. These show the impact the speedup can have on VLMC comparison. Table 4.1 is generated by comparing all the VLMCs from the Human, Turkey, and Corn data sets in one run. PST was killed, likely due to memory usage by the implementation. The SBS finished with the best execution time on the Human, Turkey, and Corn data sets while Sorted vector performed best on the Virus data set. This further sets the new implementations apart, likely due to a smaller k-mer size (read-in k-mer) the new implementations can handle strictly larger data sets.

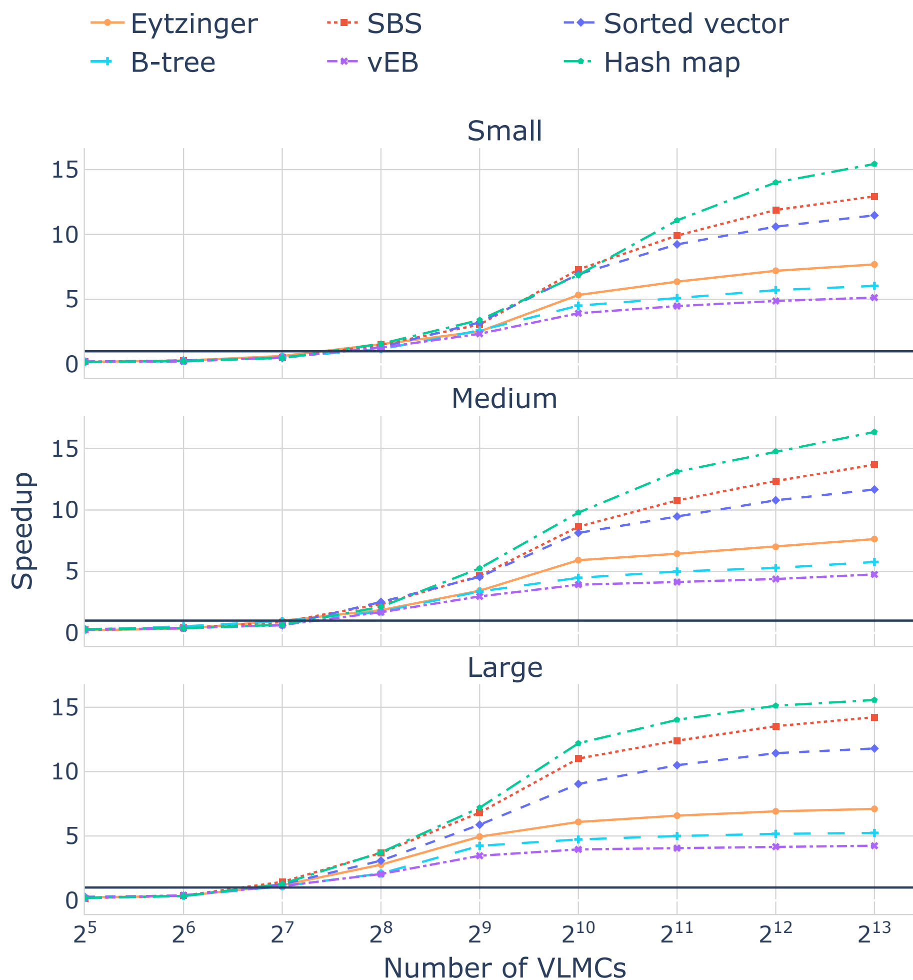


Figure 4.4: Desktop speedup versus PST for the different containers when comparing the increasing number of VLMCs from the virus dataset.

4.2 Memory & Cache

This section provides results on cache behavior, specifically how the cache miss rate is different between containers and data sets. It also shows the heap usage of the

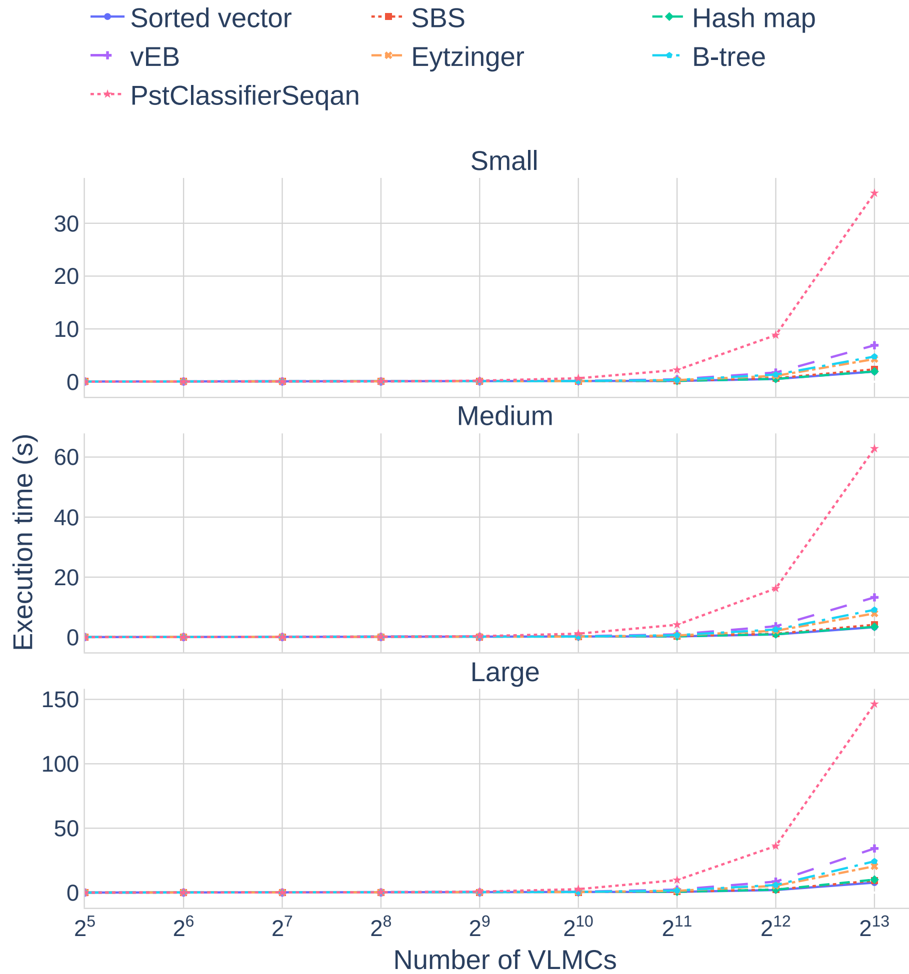


Figure 4.5: Desktop execution times for the different containers when comparing the increasing number of VLMCs from the virus dataset.

	Elapsed Time Laptop (s)		
	Small	Medium	Large
PST	1.23	85.15	DNF
Sorted Vector	0.07	8.05	34.49
SBS	0.08	7.06	26.55
Hash map	0.10	13.06	44.18
vEB	0.29	18.46	60.60
Eytzinger	0.18	17.80	65.23
B-tree	0.24	22.65	87.67

Table 4.1: Elapsed time on Laptop for distance computation on the Human, Turkey, and Corn data sets combined. DNF is a crash, killed by the OS.

	Elapsed Time Laptop (s)		
	Small	Medium	Large
PST	172.97	287.87	697.41
Sorted Vector	7.85	15.80	38.40
SBS	10.69	14.37	41.86
Hash map	6.24	11.34	38.95
vEB	22.49	43.27	131.30
Eytzinger	11.67	22.31	66.81
B-tree	20.64	49.72	120.97

Table 4.2: Elapsed time on Laptop for distance computation on the Virus-to-Virus data set.

different implementations.

4.2.1 Cache misses

Cache misses is the ratio of cache references that are not present in the cache (and need to be fetched from main memory) to total cache references in the last level cache, or L3 (Level 3) on all computer types. The Laptop and HPC setups produce more than 50% cache misses on Medium and Large version of Human, Turkey and Corn data sets, whilst the Desktop has fewer cache misses for all data sets and implementations, see Figures 4.6, 4.8 and 4.7.

For the Small version of Human, Turkey, and Corn the Laptop show a decrease in cache misses compared to PST for all implementations, see Figure 4.6. For the larger versions of these data sets, the cache miss rate become very high. However, the SBS and Sorted vector show a constant improvement in cache misses compared to the other implementations. For the Virus to Virus data set, all implementations show a great improvement compared to PST.

The HPC has the worst cache misses out of all the setups, see Figure 4.8. Similar to the Laptop results, all implementations attain better cache miss ratio for the Virus to Virus data sets. On the Human, Turkey, and Corn data sets, the Small version is inconclusive, while for the Medium and Large, most implementations generate fewer cache misses compared to PST.

Figure 4.7 show the results of cache misses on the Desktop. SBS and Sorted vector have the lowest cache misses on all data sets. The Hash map and B-tree have the highest cache misses for most data sets. The Virus to Virus data sets generate the lowest amount of cache misses compared to the other data sets. For the Human, Turkey and Corn data sets, the Larger data sets have higher cache misses for the B-tree and Hash map implementations. Together, the iterative containers (SBS and Sorted vector) generally produce the fewest cache misses on the Desktop.

Overall, for larger data sets (Medium and Large Human, Turkey and Corn) the

iterative implementations tend to produce fewer cache misses out of all implementations, see Figures 4.6, 4.8 and 4.7. On small data sets (Virus, and Small versions of all data sets) it is more difficult to make conclusions. For the Desktop, SBS and Sorted vector produce the fewest cache misses. On HPC and Laptop Eytzinger seem to produce the fewest overall cache misses.

The reason for the cache behavior is not straightforward. From Table 3.2-3.5, Large Corn has the greatest average number of k-mers of the data sets. Since each read-in k-mer is 40 bytes, the size of the VLMCs is roughly the number of k-mers times 40 since all containers except the Hash map are flat. From this, VLMCs in the Large Corn data set are about 192 MB. This is greater than the L3 cache size of the Laptop, HPC, and Desktop, which could create Capacity cache misses, see Section 2.2.2. However, only Laptop and HPC produce a high rate of cache misses on the large Corn data set. The Desktop results thereby make conclusions of the type of cache misses difficult.

There could be a number of reasons for the inconsistency. The L3 cache associativity of the Desktop is higher compared to the Laptop and HPC, which could indicate that the cache misses are caused by Conflict misses. Further, we found that fewer cache misses occurred on Laptop when running in single-thread mode (a decrease of 10 % units on the large Human data set). This type of behavior could indicate Coverage misses as the cache coherence directory is likely smaller compared to the Desktop. Since the Desktop did not show the same behavior, and threads work almost vexclusively independently (there are no invalidating writes) Coherence misses are probably not the cause.

Another cause could be the cache memory latencies since the work performed for each matching k-mer is very small. In this case, the Desktop cache latency would be just low enough to supply the CPU with data in time, while HPC and Laptop cache latencies are just slow enough to not serve the CPU in time. This was discussed previously in Section 4.1 with the two possible read-in k-mer designs using either vectorizations or beeing of smaller size.

Lastly, the benchmarking machines use very different architectures, by which they could have different hardware prefetchers. The memory access pattern should however be simple enough for all types of prefetchers for the Sorted vector and to some degree SBS. This is likely what results in these containers having the fewest cache misses overall on Medium and Large data sets.

To get a better understanding of the cache misses ratio, Figure 4.9 shows the number of cache references produced for the different implementations on the Laptop. Overall PST has the most references, especially for the Virus data set. This is probably due to moving the background order normalization, see Section 3.3.2. Another reason is the reduction in size of the k-mer. The search implementations also produce more cache references. Especially Eytzinger, which uses the "all the way to the bottom search" and then reset bits, requiring more cache references.

Figure 4.6: Laptop cache misses for different containers and data sets.

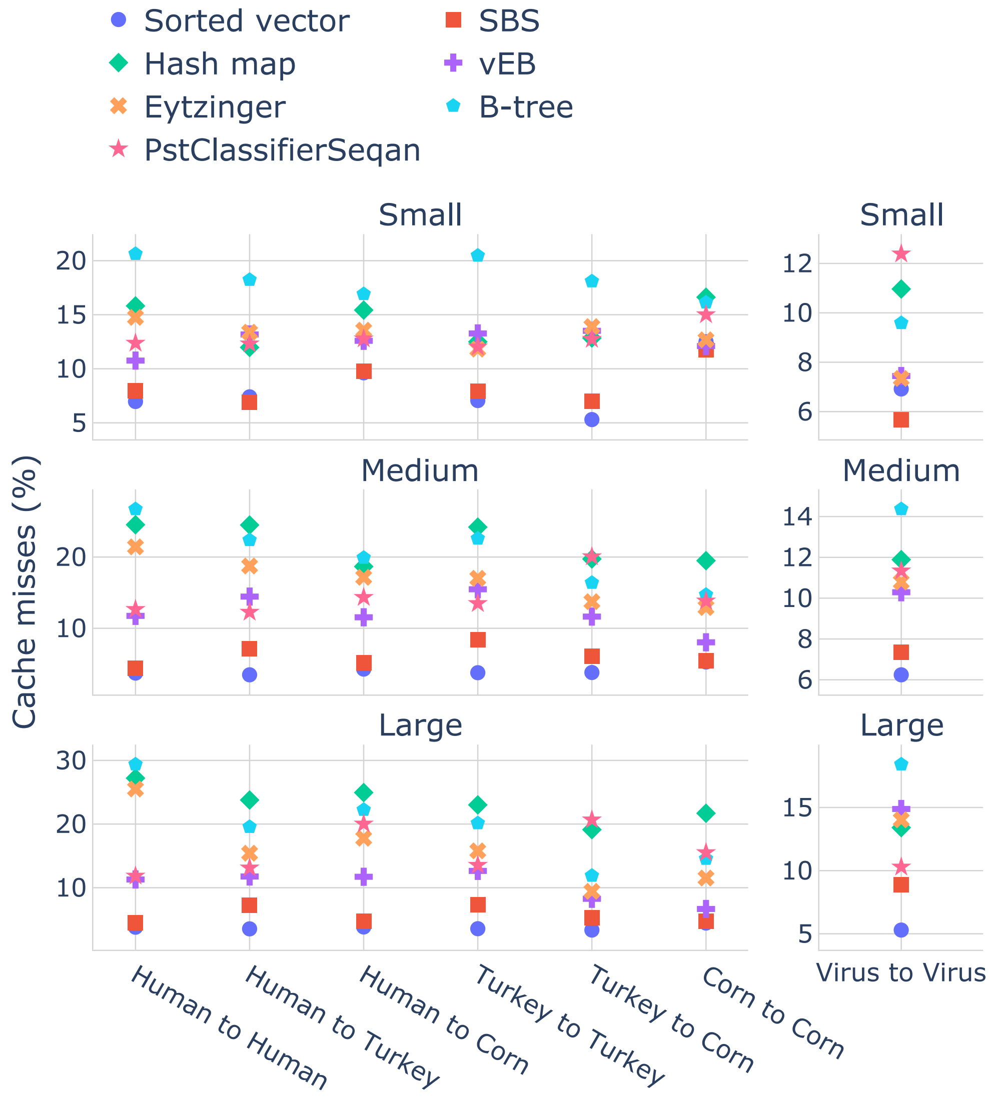


Figure 4.7: Desktop cache misses for different containers and data sets.

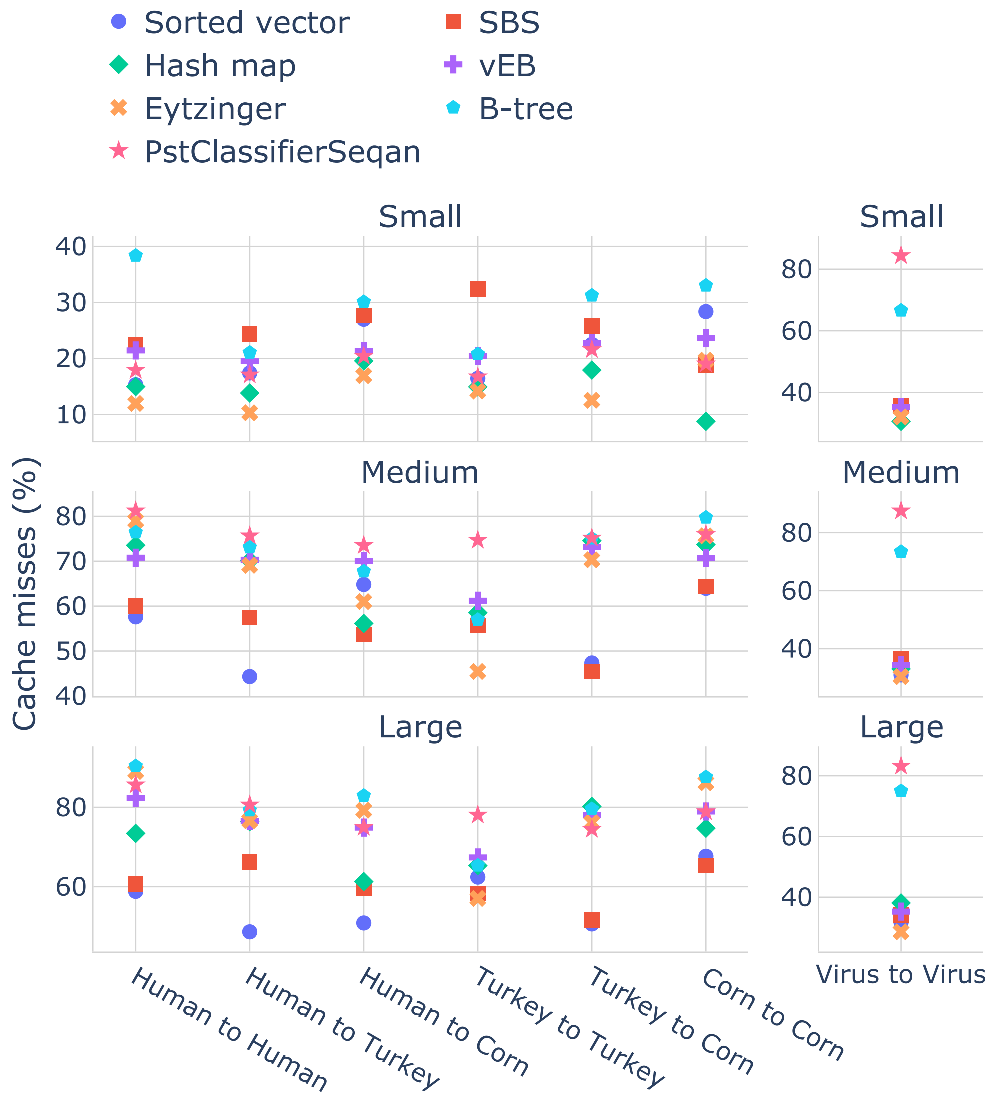


Figure 4.8: HPC cache misses for different containers and data sets.

Figure 4.9: Laptop cache reference counts for different containers and data sets.

4.2.2 Space usage

Peak heap usage of different implementations is provided with both the total heap allocation and the share of unused heap allocation. The unused heap is memory that is allocated to the program by the OS but not actually used. Table 4.3 shows less memory usage for all new implementations in comparison to PST. Only the small versions of two data sets are shown as the larger data sets have too long an execution time to evaluate using `Valgrind`.

For Small Human, Eytzinger, B-tree, vEB, Sorted vector, and SBS require the least amount of heap memory. Kmer-bucket allocates almost as much memory as PST and it has the most unused memory allocated. Hash map allocates less memory than PST but still about 1.5 times as much as the other implementations (except for Kmer-bucket).

Considering Small Virus, Eytzinger, B-tree, vEB, Sorted vector, and SBS require the least amount of heap memory. Compared to these PST uses almost exactly twice as much memory. The Hash map and Kmer-bucket implementations are worse but not to the same extent as for the Small Human data set. The difference between Sorted vector and SBS is very small, indicating that the summary structure in SBS does not require much extra memory as it is the only difference between the two. All implementations except for the Sorted vector and Hash map have a high amount of unused heap.

	Small Human		Small Virus	
	Heap (MB)	Unused Heap (KB)	Heap (MB)	Unused Heap (KB)
Sorted vector	10.6	14.7	46.7	63.1
PST	36.7	23.3	95.1	603.6
SBS	10.7	15.2	47.4	116.0
Eytzinger	10.5	16.8	47.3	254.7
B-tree	10.6	16.1	47.3	252.3
vEB	10.5	16.6	48.4	253.5
Hash map	15.8	14.8	56.0	80.5
Kmer-bucket	20.8	2000.0	61.4	1300.0

Table 4.3: Heap memory usage for the different implementations on Small Human (24 VLMCS to 24 VLMCs) and Small Virus (2000 VLMCs to 2000 VLMCs). The Heap column is the total heap allocated, and the Unused heap is the extra memory allocated to the program which is not used. The heap memory evaluation was generated on the Laptop.

From Table 4.3 PST likely uses more heap since the old k-mer data structure occupies more memory in addition to hash maps utilizing more memory than is needed. The latter can be seen in the difference between Sorted vectors heap allocation compared to the Hash map implementation. Eytzinger, B-tree, and vEB have a lot of unused heap memory in the case of small VLMCs (Small Virus). This is likely due to

the fact that these three containers copy their data from a sorted vector into their own structure, a procedure that requires two vectors of the same size to be kept in memory at the same time. Kmer-bucket uses more memory compared to the other new containers. As `std::vector` allocates more memory than necessary to make back insertion fast and has a large number of `std::vectors`.

The larger heap allocation of PST on the small Human data set, see Table 4.3, could be the reason why this implementation fails to finish executing the distance between all large VLMCs from the Human, Turkey, and Corn data sets, see Section 4.1.

4.3 Parallelization

The parallel speedup of the different containers in comparison to the theoretical maximum of Ahmdal's law (see Section 2.7) is shown in Figure 4.10. The container's parallel speedup is the speedup achieved compared to their single-thread execution time when increasing the number of cores. The results were generated on the HPC cluster with 32 cores as the maximum amount of cores available. Note that all containers except the Hash map provide comparable speedup, while the Hash map has the least gain of more cores.

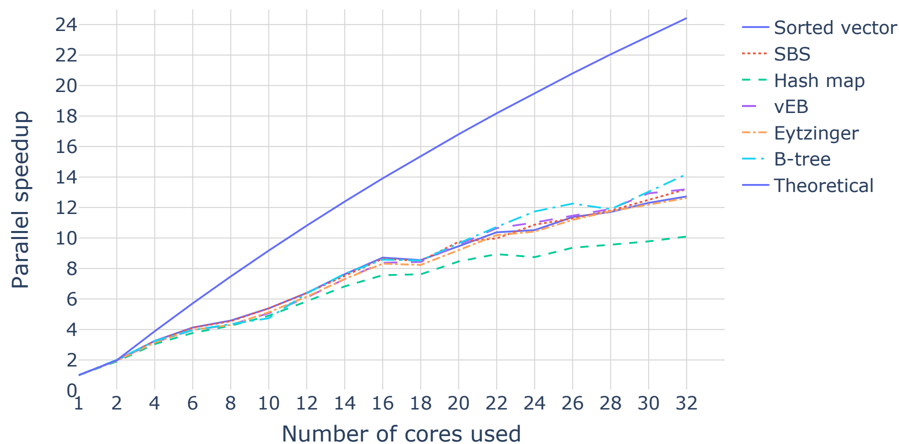


Figure 4.10: Parallel speedup of the program in relation to the theoretical maximum speedup. Theoretical maximum parallel speedup is almost infinite since distance computation takes about 99% of execution time on large enough data sets.

There does not seem to be a clear reduction in parallel speedup for any of the containers, which is a promising result for smaller HPC systems. As all containers use the same parallelization policy (split along the largest axis of the distance matrix, see Section 3.4) which does not use any mutex. Further, as work is split along the largest axis, the program can utilize as many cores as the length of the largest axis (theoretically). This policy implies that we are mainly limited by hardware threads, scheduling, and memory size and bandwidth as long as there are fewer cores than the dimensions of the axis. The experiment was limited to 32 cores, by which conclusions

for greater core counts (and HPC systems) should be carefully considered. It is likely the parallel speedup no longer increases beyond some core count, but this can not be seen from this small experiment.

When loading VLMCs (see Section 3.3.2), the number of cores is limited to a maximum of four, see Section 3.4. However, since this part of the program is linear in execution time, and the distance computation is quadratic, the ratio of time spent in the distance computation is much greater than the loading phase for large enough input sizes. Hence, the more VLMCs compared the better the program will be at scaling to the number of cores utilized.

4.4 Matrix recursion

Beyond changing the VLMC data structure, the algorithm for selecting the order of VLMC pairs to compute the distances between was also changed. The Matrix recursion algorithm (see section 2.3.1) in combination with the Sorted vector container shows clear improvements in run time and cache misses in comparison to using a typically nested for-loop, see Table 4.4.

Nested for loop	Time (s)	366	±21.3
	Cache misses (%)	72.4	±18.5
Matrix Recursion	Time (s)	46	±9.75
	Cache misses (%)	6.6	±0.96

Table 4.4: Performance difference when using matrix recursion and a nested for-loop when computing the distances between all VLMCs in the Virus data set. Time and cache misses are averaged over 5 runs, with the standard deviation in the rightmost column.

This result is exciting since it is common to apply a function to all pairs formed from sets of elements. Given that these elements usually are more complex than the basic types, this solution could probably be applied to many problems.

4.5 Effects of Downsampling

Investigating the downsampling strategy for speed was performed by removing a percentage of all k-mers from all VLMCs, as described in Section 3.5.

Downsampling was performed by first removing a percentage x of all k-mers in each VLMC and applying the distance with the vEB container. Removing a random amount of k-mers will reduce the number of shared k-mers between VLMCs, and there by should a search container be used. Compared to Sorted vector which iterates through the VLMCs, where the gain of removing k-mers is not as substantial since non-shared k-mers will still need to be iterated over.

Figure 4.11 shows some small speedup gained while sacrificing a small change in the distance computed. However, this distance is relative and the actual value of the

distance computed is not necessarily useful. Instead, the most important factor is that the relative distance between the different VLMCs remains.

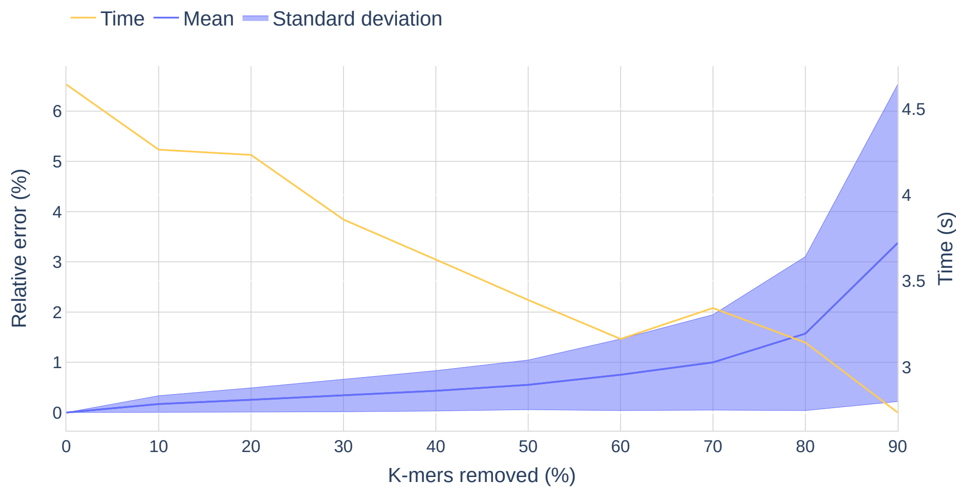


Figure 4.11: Relative error and execution time when comparing distance for the turkey dataset with the vEB container.

To measure the effect on the relative distance between the VLMCs, different levels of downsampling were tried when calculating the 10 closest VLMCs when comparing one VLMC to all others for the Virus data set. In Figure 4.12 the results of this are shown where the relative position of the 10 closest VLMCs are also tracked.

It is clear that this rudimentary test of downsampling does not show great promise. There is little improvement in execution time and the classification test shows the practical usefulness is limited. Even though the distance measure shows a somewhat small relative error, the distance will most likely be used in a scenario with classification, where the downsampling approach clearly misclassifies the most similar samples even at a small downsampling rate.

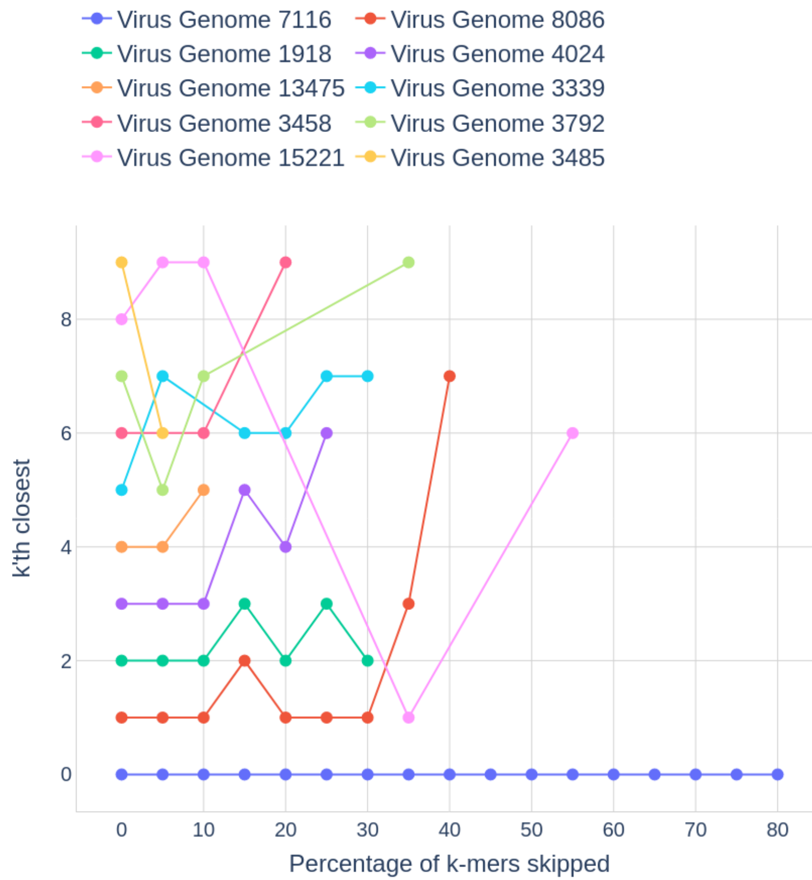


Figure 4.12: Classification test on the Virus dataset. The 0th closest is the VLMC compared to itself. After 10% downsampling the three closest samples are misclassified and after 30% only the identity is correctly identified.

5

Conclusion

This thesis provides several optimizations to compute the d_v^* dissimilarity between Variable Length Markov Chains (VLMC) generated from DNA sequences. We show speedup compared to the current state-of-the-art Probabilistic Suffix Tree (PST) in the range of 2-20 times the PST's execution time depending on hardware and data sets.

One main issue with the PST implementation highlighted by Gustafsson et al. was the percentage of cache misses. We improved the cache miss ratio when comparing many VLMCs from the NCBI Virus data set. This was achieved by utilizing a transformation of the cache-oblivious Matrix Transpose algorithm. For data sets where the VLMCs are larger (i.e. containing 77 000 k-mers or more), we found it difficult to reduce the percentage of cache misses on two of our three benchmarking setups.

The most promising data structure is a simple sorted vector of k-mers with an additional summary data structure which we call Sorted Block Search (SBS). It allows fast distance computations when VLMCs are similar and share many k-mers. It also handles the worst-case scenarios for a sorted vector approach when large VLMCs share few k-mers. The fastest data structure for small VLMCs (a few 1000s of k-mers), a Hash map solution is the fastest solution.

Several proposed cache-efficient data structures implemented also show comparable, albeit less, speedup to the iterative approach (Sorted vector and SBS). These data structures do not utilize the fact that many k-mers appear in both VLMCs, making many search calls redundant compared to incrementing approaches. In some cases, they provide a lower cache miss ratio but are overall slower. This is not surprising as these data structures must re-reference multiple k-mers during the search, whereas incrementing approaches only load each k-mer once during the dissimilarity computation of two VLMCs.

A further interesting discovery was the use (or non-use) of SIMD instructions. Our implementations opted for small k-mers of 40 bytes with no SIMD instruction used in arithmetic. This solution was faster than using SIMD instructions on a Laptop and an High Performance Compute cluster (HPC) on data sets with many hundred thousands or millions of k-mers. However, on smaller data sets, e.g. Virus, SIMD instructions can about double the performance (from 15 times faster than PST to 30 times). On Desktop, the difference was less noticeable, mainly the speedup decreased

with the smaller k-mers. As this machine also had the fewest cache misses, memory access is most likely not the bottleneck, and SIMD instructions plainly sped up execution time.

The algorithm improvements show acceptable scaling for large data sets for up to 32 cores in comparison to Ahmdals' law. This indicates that the solution can be applied to larger data sets on compute clusters containing a database of VLMCs.

As future work, there are some parts that the project could not fully investigate. Mainly, the different implementations given by Khuong et al. [16] should be further tested as it is possible some data structures could work better than the chosen implementations. The most interesting future investigation is however the tradeoff of vectorization and compact representations of k-mers. This was not properly parameterized and measured and could give insights into what the algorithm is dependent on and when, and possibly switch implementation based on architectures or data.

In summary, our implementation allows faster approximate classification of sequenced viruses or contagious diseases. Further, it allows the investigation of species' evolution on a larger scale with VLMCs than before.

Bibliography

[1] R. K. Wayne and E. A. Ostrander, "Lessons learned from the dog genome," *TRENDS in Genetics*, vol. 23, no. 11, pp. 557–567, 2007.

[2] A. Zielezinski, S. Vinga, J. Almeida, and W. M. Karlowski, "Alignment-free sequence comparison: Benefits, applications, and tools," *Genome biology*, vol. 18, pp. 1–17, 2017.

[3] S. B. Needleman and C. D. Wunsch, "A general method applicable to the search for similarities in the amino acid sequence of two proteins," *Journal of molecular biology*, vol. 48, no. 3, pp. 443–453, 1970.

[4] P. Bühlmann and A. J. Wyner, "Variable length markov chains," *The Annals of Statistics*, vol. 27, no. 2, pp. 480–513, 1999.

[5] D. Ron, Y. Singer, and N. Tishby, "The power of amnesia," *Advances in neural information processing systems*, vol. 6, 1993.

[6] J. Gustafsson, P. Norberg, J. R. Qvick-Wester, and A. Schliep, "Fast parallel construction of variable-length markov chains," *BMC bioinformatics*, vol. 22, no. 1, pp. 1–23, 2021.

[7] J. Gustafsson, Personal communication, 2022.

[8] J. Gustafsson and E. Norlander, "Clustering genomic signatures - a new distance measure for variable length markov chains," Thesis, 2018. [Online]. Available: `https://odr.chalmers.se/items/b4fc7ac2-d52a-4c96-a05e-beb6fb0733c1`.

[9] S. Vinga and J. Almeida, "Alignment-free sequence comparisona review," *Bioinformatics*, vol. 19, no. 4, pp. 513–523, 2003.

[10] J. Gustafsson, *Vlmc construction using a lazy suffix tree*, Feb. 2021. [Online]. Available: `https://github.com/Schlieplab/PstClassifierSeqan` (visited on 10/27/2022).

[11] Bevin Brett, *Memory performance in a nutshell*, `https://www.intel.com/content/www/us/en/developer/articles/technical/memory-performance-in-a-nutshell.html`, Accessed: 2023-05-04, 2016.

[12] M. Frigo, C. E. Leiserson, H. Prokop, and S. Ramachandran, "Cache-oblivious algorithms," in *40th Annual Symposium on Foundations of Computer Science (Cat. No. 99CB37039)*, IEEE, 1999, pp. 285–297.

[13] E. D. Demaine, "Cache-oblivious algorithms and data structures," *Lecture Notes from the EEF Summer School on Massive Data Sets*, vol. 8, no. 4, pp. 1–249, 2002.

[14] R. Fleischer, B. Moret, and E. M. Schmidt, *Experimental algorithmics: From algorithm design to robust and efficient software*. Springer, 2003, vol. 2547.

[15] S. Chatterjee and S. Sen, "Cache-efficient matrix transposition," in *Proceedings Sixth International Symposium on High-Performance Computer Architecture. HPCA-6 (Cat. No. PR00550)*, IEEE, 2000, pp. 195–205.

[16] P.-V. Khuong and P. Morin, "Array layouts for comparison-based searching," *ACM J. Exp. Algorithmics*, vol. 22, May 2017, ISSN: 1084-6654. DOI: 10.1145/3053370. [Online]. Available: https://doi.org/10.1145/3053370.

[17] G. S. Brodal, R. Fagerberg, and R. Jacob, "Cache oblivious search trees via binary trees of small height," *BRICS Report Series*, no. 36, 2001.

[18] M. A. Bender, E. D. Demaine, and M. Farach-Colton, "Cache-oblivious b-trees," in *Proceedings 41st Annual Symposium on Foundations of Computer Science*, IEEE, 2000, pp. 399–409.

[19] G. M. Amdahl, "Validity of the single processor approach to achieving large scale computing capabilities," in *Proceedings of the April 18-20, 1967, spring joint computer conference*, 1967, pp. 483–485.

[20] M. Leitner-Ankerl, *Robin-hood-hashing*, https://github.com/martinus/robin-hood-hashing/tree/363753c1fdba7ac14362aefb4b9803ab60a70e49, 2021.

[21] M. Leitner-Ankerl, *Ankerl:unordered_dense*, https://github.com/martinus/unordered_dense/tree/a3fea4afc6ecfab0bbfd401aded786a72a8e2b3b, 2023.

[22] G. Guennebaud, B. Jacob, *et al.*, *Eigen v3*, http://eigen.tuxfamily.org, 2010.

A

Code listings

```
void matrix_recursion(size_t start_index_left,
                      size_t stop_index_left,
                      size_t start_index_right,
                      size_t stop_index_right,
    const std::function<void(size_t &left, size_t &right)> &fun){
    int diff_left = stop_index_left - start_index_left;
    int diff_right = stop_index_right - start_index_right;
    if (diff_left == 1 && diff_right == 1){
        fun(start_index_left, start_index_right);
    } else if (diff_right > diff_left){
        int new_right_index = (stop_index_right + start_index_right) / 2;
        matrix_recursion(start_index_left, stop_index_left,
                            start_index_right, new_right_index, fun);
        matrix_recursion(start_index_left, stop_index_left,
                            new_right_index, stop_index_right, fun);
    } else {
        int new_left_index = (stop_index_left + start_index_left) / 2;
        matrix_recursion(start_index_left, new_left_index,
                         start_index_right, stop_index_right, fun);
        matrix_recursion(new_left_index, stop_index_left,
                         start_index_right, stop_index_right, fun);
}
```

Listing 7: The matrix recursion algorithm.

```
void iterate_kmers(VLMC_Container &left_kmers,
                   VLMC_Container &right_kmers,
    const std::function<void(const RI_Kmer &left, const RI_Kmer &right)> &f)
    override {
      RI_Kmer left_kmer = left_kmers.find(this->min_index);
      RI_Kmer right_kmer = right_kmers.find(this->min_index);
      while(true){
        // Check if right_kmers has this succeeding left_kmer
        // if, apply f
        if(left_kmer == right_kmer){
          f(left_kmer, right_kmer);
        }
        // Get the next k-mer
        left_kmer = veb::succ(veb, left_kmer);
        if(left_kmer.integer_rep == -1){
          return;
        }
        // Retrieve the corresponding k-mer in the other container
        right_kmer = right_kmers.find(left_kmer.integer_rep);
      }
    }
```

Listing 8: Algorithm to iterate to VLMCs, denoted left_kmers and right_kmers in code.

```
void iterate_kmers(VLMC_Container &left_kmers, VLMC_Container &right_kmers,
    const std::function<void(const RI_Kmer &left, const RI_Kmer &right)> &f) {
    auto right_it = right_kmers.begin();
    auto right_end = right_kmers.end();
    auto left_it = left_kmers.begin();
    auto left_end = left_kmers.end();
    while(left_it != left_end && right_it != right_end){
        if(*left_it == *right_it){
            f(*left_it, *right_it);
            ++left_it;
            ++right_it;
        } else if(*left_it < *right_it) {
            ++left_it;
        }
        else ++right_it;
      }
    }
```

Listing 9: Iteration of two VLMCs with sorted vectors. f is Eq 2.3, without the denominator. Iteration is done by moving the `std::iterators` forward.

```
static int child(unsigned c, int i) {
    return (B+1)*i + (c+1)*B;
}

std::vector<container::RI_Kmer>::iterator
    construct(std::vector<container::RI_Kmer>::iterator a0, int i) {
    if (i >= size) return a0;

    for (unsigned c = 0; c <= B; c++) {
        // visit c'th child
        a0 = construct(a0, child(c,i));
        if (c < B && i+c < size) {
            a[i+c] = *a0++;
        }
    }
    return a0;
}
```

Listing 10: Algorithm to recursively construct a B-tree using Khuong et al.'s approach [16].. `size` is the number of elements and `B` is the selected number of elements in each node, here set to 2.

```
int unrolled_branchfree_search(int x) const {
  int j = size;
  int i = 0;
  while (i + B <= size) {
    __builtin_prefetch(a.data()+child(i, B/2), 0, 0);
    const container::RI_Kmer *base = &a[i];
    const container::RI_Kmer *pred = branchfree_inner_search<B>(base, x);
    unsigned int nth = (*pred < x) + pred - base;
    {
        const container::RI_Kmer current = base[nth % B];
        int next = i + nth;
        j = (current >= x) ? next : j;
    }
    i = child(nth, i);
    }
if (__builtin_expect(i < size, 0)) {
    // last block
    const container::RI_Kmer *base = &a[i];
    int m = size - i;
    while (m > 1) {
        int half = m / 2;
        const container::RI_Kmer *current = &base[half];

        base = (*current < x) ? current : base;
        m -= half;
    }

    int ret = (*base < x) + base - a.data();
    return (ret == size) ? j : ret;
    }
return j;
}
```

Listing 11: Algorithm to do an unrolled and branch-free search in a B-tree using Khuong et al.'s approach [16].. `size` is the number of elements and `B` is the selected number of elements in each node, here set to 2.

```
struct tree_data {
    h_type h0;
    h_type h1;
    h_type dummy[2];
    int m0;
    int m1;
};

void sequencer(int h, tree_data *s, unsigned d) {
    if (h == 0) return;
    int h0 = h/2;
    int h1 = h-h0-1;
    sequencer(h0, s, d);
    s[d+h0].h0 = h0;
    s[d+h0].m0 = (2<<h0)-1;
    s[d+h0].h1 = h1;
    s[d+h0].m1 = (2<<h1)-1;
    sequencer(h1, s, d+h0+1);
    }

container::RI_Kmer*
        construct(container::RI_Kmer* a0, int* rtl, int path, unsigned d) {
    if (d > height || rtl[d] >= n) return a0;

    // visit left child
    path <<= 1;
    rtl[d+1] = rtl[d-s[d].h0] + s[d].m0 + (path&s[d].m0)*(s[d].m1);
    a0 = construct(a0, rtl, path, d+1);

    a[rtl[d]] = *a0++;

    // visit right child
    path += 1;
    rtl[d+1] = rtl[d-s[d].h0] + s[d].m0 + (path&s[d].m0)*(s[d].m1);
    a0 = construct(a0, rtl, path, d+1);

    return a0;
}
```

Listing 12: Algorithm to construct the vEB layout using Khuong et al.'s approach [16].

```
int search(int x) {
    int rtl[MAX_H+1];
    int j = n;
    int i = 0;
    int p = 0;
    for (int d = 0; i < n; d++) {
        rtl[d] = i;
        if (x < a[i].integer_rep) {
            p <<= 1;
            j = i;
        } else if (x > a[i].integer_rep) {
            p = (p << 1) + 1;
        } else {
            return i;
        }
        i = rtl[d-s[d].h0] + s[d].m0 + (p&s[d].m0)*(s[d].m1);
    }
    return j;
    }
}
```

Listing 13: Algortihm to find the index of k-mer with index representation `x` using Khuong et al.'s approach [16].

```
vmulpd %ymm2, %ymm3, %ymm5
vmulpd %ymm2, %ymm2, %ymm2
addq $64, %rdx
addq $64, %rax
vextractf128 $0x1, %ymm5, %xmm1
vmulpd %ymm3, %ymm3, %ymm3
vaddpd %xmm5, %xmm1, %xmm1
vunpckhpd %xmm1, %xmm1, %xmm5
vaddsd %xmm5, %xmm1, %xmm1
vaddsd %xmm1, %xmm7, %xmm7
vextractf128 $0x1, %ymm2, %xmm1
vaddpd %xmm2, %xmm1, %xmm1
vunpckhpd %xmm1, %xmm1, %xmm2
vaddsd %xmm2, %xmm1, %xmm1
vaddsd %xmm1, %xmm0, %xmm0
vextractf128 $0x1, %ymm3, %xmm1
vaddpd %xmm3, %xmm1, %xmm1
vunpckhpd %xmm1, %xmm1, %xmm2
vaddsd %xmm2, %xmm1, %xmm1
vaddsd %xmm1, %xmm6, %xmm6
cmpq %rdx, %rcx
```

Listing 14: Updating Equation 2.2 and numerator of Equation 2.3 when using an Eigen vector for the next-character probabilities. The assembly is produced with optimization `-O3` and the `-march=native` flag.

```
movq 16(%rsp), %rdx
movq 24(%rsp), %r9
popq %rax
.cfi_def_cfa_offset 88
subq %rdx, %r9
movq %r9, %rax
shrq $3, %rax
popq %rcx
.cfi_def_cfa_offset 80
je .L15870
movq 24(%rsp), %r8
movq 24(%rbx), %rdi
movq 32(%rbx), %rsi
movq 40(%rbx), %rcx
xorl %eax, %eax
.p2align 4,,10
.p2align 3
.L15869:
vmovsd (%rdx,%rax), %xmm0
vmovsd (%rdi), %xmm1
vfmadd132sd (%r8,%rax), %xmm1, %xmm0
vmovsd %xmm0, (%rdi)
vmovsd (%rdx,%rax), %xmm0
vfmadd213sd (%rsi), %xmm0, %xmm0
vmovsd %xmm0, (%rsi)
vmovsd (%r8,%rax), %xmm0
addq $8, %rax
vfmadd213sd (%rcx), %xmm0, %xmm0
vmovsd %xmm0, (%rcx)
cmpq %rax, %r9
jne .L15869
```

Listing 15: PST implementation updating Equation 2.2 and numerator of Equation 2.3 in the old implementation. Note the loop and 10 extra instructions in comparison to the new implementation. Optimization `-O3` and the `-march=native` flag.

B

Additional table

	Human to Human Execution Time (sec)			Virus to Virus Execution Time (sec)		
	Double	Eigen	Float	Double	Eigen	Float
Sorted Vector	5.53	8.64	4.52	57.17	36.88	50.06
SBS	5.53	9.44	4.63	46.55	49.83	43.77
Hash Map	9.25	12.88	8.03	46.60	45.80	37.46
Eytzinger	11.91	18.06	9.58	80.87	94.48	77.78
vEB	12.34	18.51	11.31	154.91	157.00	156.89
B-tree	18.03	18.19	12.26	141.24	122.55	130.11

Table B.1: Execution time on Laptop for the Large Human and Virus data sets when using `Double`, `Eigen`, and `Float`.