UNIVERSITY OF OKLAHOMA

GRADUATE COLLEGE

Compressing Massive Sequencing Data with Multiple Attribute Tree

A THESIS

SUBMITTED TO THE GRADUATE FACULTY

in partial fulfillment of the requirements for the

Degree of

MASTER OF SCIENCE

By

Dorian Selimovic

Norman, Oklahoma

2019

Compressing Massive Sequencing Data with Multiple Attribute Tree

A THESIS APPROVED FOR THE
SCHOOL OF COMPUTER SCIENCE

BY THE COMMITTEE CONSISTING OF

Abstract

The significant drop in DNA Sequencing costs caused by Next-Generation Sequencing has led to the production of massive amounts of raw sequencing data. This data is stored in FASTQ files, which are text files containing a large number of reads, each composed of a short DNA sequence and its associated identifier and quality score. The DNA sequence is a string of fixed length over the alphabet $\Sigma = \{A, C, T, G, N\}$, the identifier is an arbitrary string that is sequencer-dependent, and the quality score is a string of the same length as the DNA sequence, indicating for each base how confident the sequencer was when determining it. These files can range from a few gigabytes to hundreds of gigabytes, which poses a Big Data challenge, as the growth of generated sequencing data now exceeds the decrease of storage hardware price. Therefore, storing and transmitting such data requires more performant compression algorithms than general purpose compressors such as gzip, the de facto standard. Many different specialized compressors have been proposed to tackle this problem.

In this thesis, we review currently existing compressors for FASTQ files and we propose a novel compression algorithm for DNA sequences, MATC, for Multiple Attribute Tree Compression. Our algorithm divides DNA sequences into k-mers, i.e., substrings of length k, and performs column-wise compression using a multiple attribute tree. In our case the multiple attribute tree is a complete tree where each node is a k-mer and each leaf represents the sequence formed by the concatenation of its parent k-mers. The tree is then stored using level-order traversal and k-mers are compressed using Huffman encoding.

We show that our algorithm offers compression ratios comparable to the current specialized compressors. Moreover, we propose a distributed version of our algorithm, allowing the compression of larger files across a cluster of machines. This allows compression to be processed in the cloud, rather than on commodity hardware, which will become less and less suited to handle the growing size of generated sequencing data.

Acknowledgements

My first thanks go to my advisor and mentor Dr. Sridhar Radhakrishnan, for his continuous support and guidance throughout my research at the University of Oklahoma.

Then, I would like to thank my fellow students in Dr. Sridhar's research group, Addison Womack, Aditya Narasimhan, Sudhindra Gopal, Michael Nelson, Aaron Morris, Dwaine Kenney, and Jonathan Leslie, for their rich insights and feedback on this research.

Additionally, I would like to thank my professors from ISIMA, Eva Hassinger and Dr. Hervé Kerivin, for their help during my time as an international student here at OU.

Finally, I would like to thank my family for their unwavering support.

Contents

Chapter 1

Introduction

In this chapter, we introduce the concepts of DNA sequencing and data compression, as well as our contributions.

1.1 DNA Sequencing

1.1.1 DNA

DNA (DeoxyriboNucleic Acid) is the molecule encoding genetic information. It is composed of elementary bricks, called nucleotides, which contain a unique nucleobase that can be of 4 types : Adenine, Cytosine, Guanine and Thymine. These nucleobases (or bases for short) are usually referred to by the first letter of their names (A, C, G and T). Therefore a fragment of DNA can be written as a string on the alphabet A, C, T, G.

1.1.2 Sequencing

DNA Sequencing is a technique consisting in determining the sequence of nucleotides composing a given DNA sample. The DNA sample is fed to a machine, called DNA sequencer, such as the Illumina MiSeq illustrated in Figure 1.1. The sequencer will then produce a FASTA or FASTQ file, containing the genetic data. These files contain a high number of small fragments of sequences called reads, and can range from a few gigabytes to hundreds of gigabytes for a single sequencing experiment.

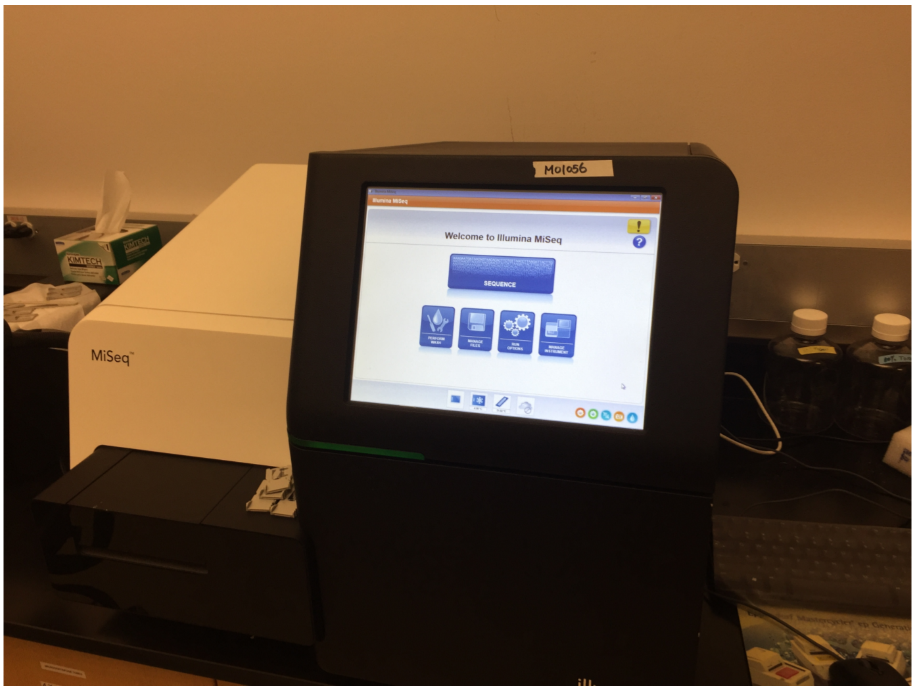


Figure 1.1: Illumina MiSeq DNA sequencer at the University of Oklahoma

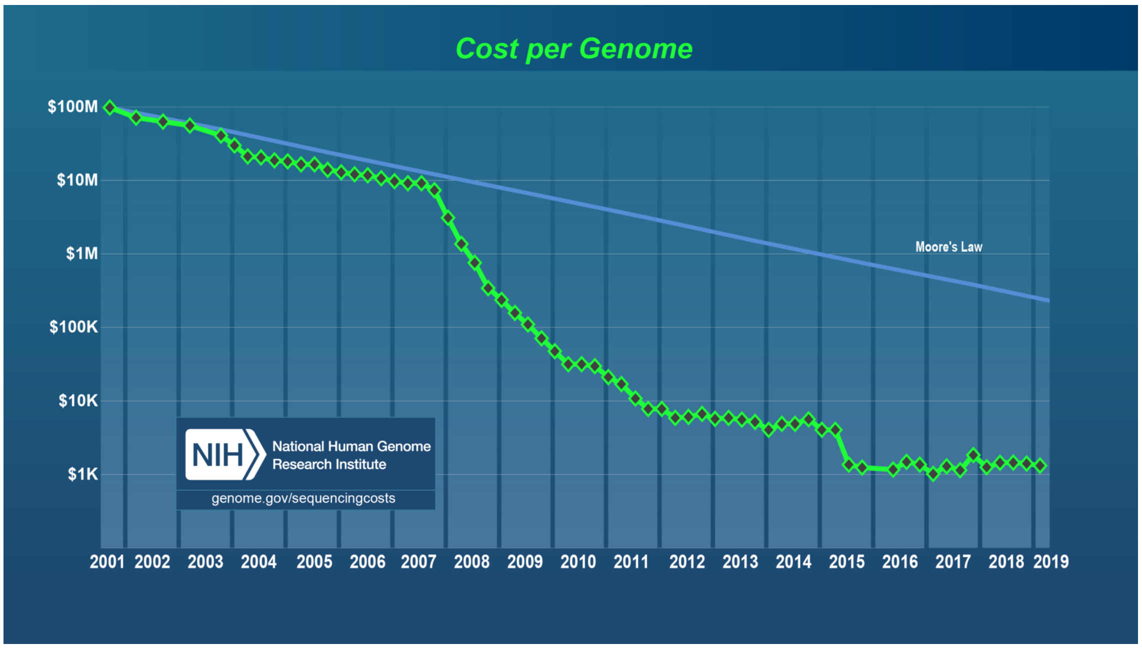


Figure 1.2: Evolution of DNA sequencing price

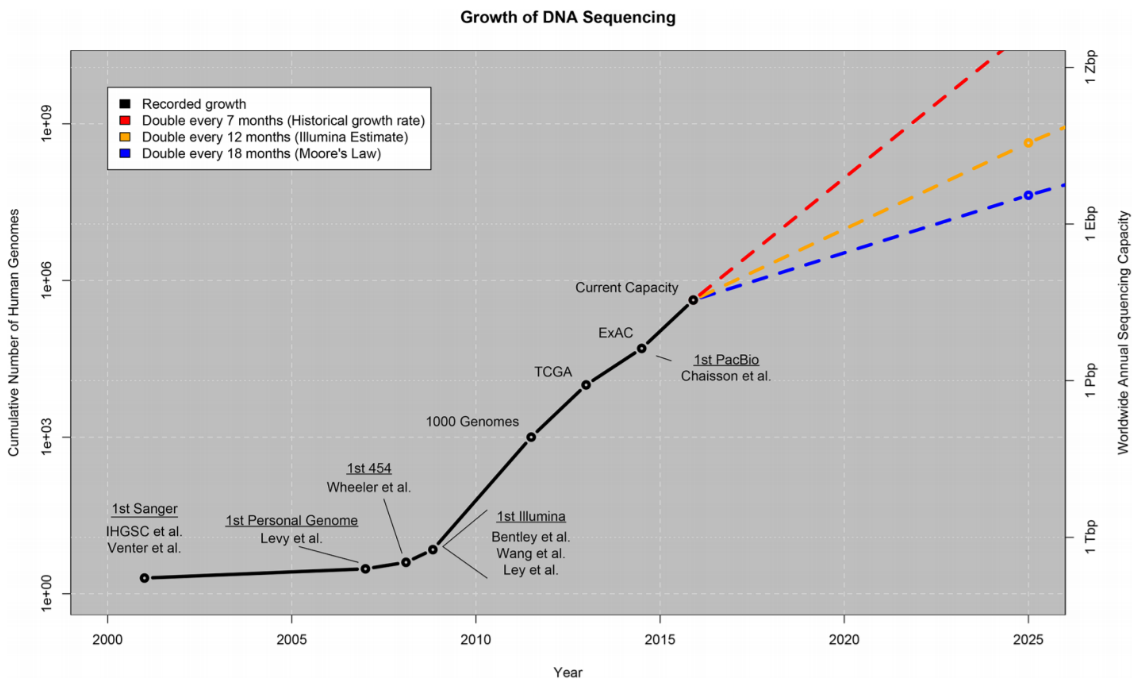


Figure 1.3: Evolution of DNA sequencing data

Since the advent of Next-Generation Sequencing, the cost of sequencing has drastically dropped, passing from a hundred million dollars for the first sequencing to only a thousand in less than twenty years (cf Figure 1.2). This technological revolution has led to an explosion of the amount of DNA sequencing data generated worldwide (cf Figure 1.3), which poses a new kind of Big Data challenge. According to Deorowicz et al. [9], this challenge is highlighted by the fact that the storage cost per byte is shrinking about three times slower than the current genomic data is growing.

This problem highly motivates research for better, specialized compressors for sequencing data. Many specialized compression algorithms have been developed and proven to be more efficient than existing, general purpose compressors such as gzip of bzip2.

1.1.3 FASTA and FASTQ File Formats

FASTA and FASTQ files are text files that contain a large number of short sequences of some fixed length, sampled randomly from the genome, called reads. In a FASTA file, each read is represented in two lines, the first line being the identifier of the read, and the second line the sequence itself. The sequence is a string over the alphabet $\Sigma = \{A, C, G, T, N\}$ where symbols A, C, G and T represent their respective bases and symbol 'N' represents a base that the sequencing machine was unable to identify. FASTQ files add a third line to each read, called the quality score, indicating for each base in the sequence, the confidence with which the machine determined it. This line is also preceded by the identifier. The identifier for the sequence is usually prefixed with a '@' character, while the same identifier for the quality score is prefixed with a '+' character. An example of a few FASTQ reads :

```
@SRR566546.970 HWUSI-EAS1673_11067_FC7070M:4:1:2299:1109 length=50
TTGCCTGCCTATCATTTTAGTGCCTGTGAGGTGGAGATGTGAGGATCAGT
+SRR566546.970 HWUSI-EAS1673_11067_FC7070M:4:1:2299:1109 length=50
hhhhhhhhhhghhghhhhhfhhhhhfffffe`ee[`X]b[d[ed`[Y[^Y
@SRR566546.971 HWUSI-EAS1673_11067_FC7070M:4:1:2374:1108 length=50
GATTTGTATGAAAGTATACAACTAAAACTGCAGGTGGATCAGAGTAAGTC
+SRR566546.971 HWUSI-EAS1673_11067_FC7070M:4:1:2374:1108 length=50
hhhhgfhhcghghggfcffdhfehhhhcehdchhdhahehffffde`bVd
@SRR566546.972 HWUSI-EAS1673_11067_FC7070M:4:1:2438:1109 length=50
TGCATGATCTTCAGTGCCAGGACCTTATCAAGCGGTTTGGTCCCTTTGTT
+SRR566546.972 HWUSI-EAS1673_11067_FC7070M:4:1:2438:1109 length=50
dhhhgchhhghhhfhhhhhdhhhhehhghfhhhchfddffcffafhfghe
```

***k*-mers:** A *k*-mer is a sequence substring of length *k*. A DNA sequence of length m has therefore (m - k + 1) *k*-mers. *k*-mers are heavily used in bioinformatics, as well as by many DNA compressors, including the algorithm we propose here.

1.2 Compression

1.2.1 Definitions

Data compression consists in, given some data, finding an encoding for that data that is more memory-efficient, i.e., that can be stored with fewer bits, while still being able to retrieve part or all of the original data. Its main idea is to eliminate redundancy of information in the original data. Its counter-part, decompression, consists in retrieving the original data from the compressed one. The compression and decompression algorithms respectively converts the original data to its encoded version and vice versa. The term *compressor* is often used to refer to a compression algorithm (and, implicitly, its corresponding decompression algorithm).

Compression can either be lossless or lossy. With lossless compression, the exact original data can be retrieved from the compressed data, whereas with lossy compression, parts of the information is lost during the compression process. Lossy compression is mostly used in applications where a good enough approximation of the original data is sufficient, such as in image, audio and video compression. Examples of lossy compressors include JPEG (image), MP3 (audio) and MP4 (video). Examples of general-purpose, lossless compressors include gzip and bzip2.

There are three main aspects to consider when evaluating the performance of a compression algorithm.

Compression ratio: It is the ratio between the size of the compressed data and the size of the original data, usually expressed in percentage. In the case of DNA compression, it may also be expressed in bits per base (bpb), i.e., the ratio between the number of bits of the compressed file, and the number of nucleobases of the original file. It represents how much space is saved when compressing, the smaller the compression ratio, the better the compression is. It is important to note that the compression ratio may vary with the size of the original data. Its variation with respect to the size of the different datasets should therefore be studied.

Compression speed: Usually expressed in megabytes per second, it is the ratio between the size of the original file and the time it takes to compress it. The higher the compression speed, the faster the compression is. It is harder to compare algorithms with respect to their compression speed, since fair comparison implies running the algorithm with the same hardware. Also, as some algorithms may offer parallel or distributed implementations, this factor must also be taken into account.

Decompression speed: Same as for compression speed, but for decompression. It is the ratio between the size of the original, or uncompressed, file and the time it takes to uncompress it. Decompression speed is usually much higher than compression speed.

Most often, there is a tradeoff between compression ratio and compression speed. Faster algorithms are more likely to have a worse compression ratio, while algorithms with a good compression ratio are more likely to be slower. Algorithms may also offer control over this tradeoff via a parameter. The two aforementioned general-purpose compressors, gzip and bzip2, both offer nine different levels of compression in this regard.

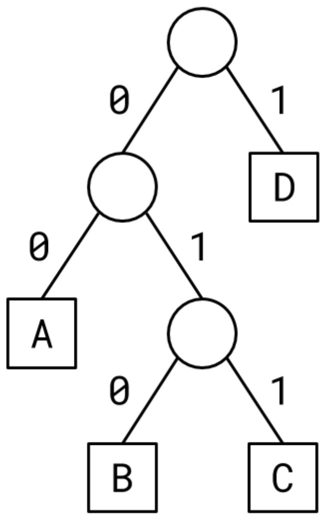
1.2.2 Huffman Coding

Huffman coding is a type of general-purpose lossless compression encoding and algorithm. Given an alphabet (e.g., the ASCII table) and the occurrence frequency of each letter of the alphabet in the input data, the Huffman algorithm finds a binary coding table to assign to each letter a variable-length binary string, such that more frequent letters have a smaller binary representation. More precisely, the algorithm builds a binary tree, called a Huffman Tree, where each leaf node represents a letter in the alphabet, and where a bit is associated with each edge (e.g., zero for left-edges, one for right-edges). Then, the binary code for each letter corresponds to the sequences of bits formed by the path between the root node and the corresponding leaf node. An example of Huffman Tree and associated coding table is given in Figure 1.4.

The algorithm for building the Huffman tree consists of the following steps:

1. A minimum priority queue is created and initialized with a leaf node for each letter, where the weight of the node in the priority queue corresponds to the number of occurrences of this letter.

2. The two lowest-weight nodes are removed from the queue, a new internal node is created with them as children and put back in the queue, its weight is the



Coding table :

Letter	Bit code
A	00
B	010
C	011
D	1

Figure 1.4: Example of a Huffman Tree and its associated coding table

sum of the weights of its children.

3. Repeat the previous step until the queue only contains a single node, which is then the root node of the so-constructed Huffman tree.

The complexity of this algorithm depends on the data structure used for the priority queue. Using a heap data structure such as a binary heap can give us at most $O(\log n)$ time complexity for inserting and removing the minimum element of the priority queue. As this operation must be repeated n times, where n is the number of letters, the construction of the Huffman tree has a time complexity of $O(n \log n)$.

The basic Huffman compression algorithm is therefore composed of four steps:

1. Count the number of occurrences of each letter in the input data.
2. Build the Huffman tree.
3. Build the coding table by applying a depth-first search algorithm on the tree.
4. Replace each letter in the input data by its corresponding bit code in the coding table.

1.3 Contributions

In this thesis we make the following contributions :

1. We present a novel compression algorithm for DNA sequencing data based on multiple attribute trees.

2. We provide an open-source, cross-platform implementation of our algorithm.

3. We also provide a distributed implementation of our algorithm, based on the master-worker paradigm.

4. We show that our distributed algorithm as an ideal linear speedup.

Chapter 2

Previous Work

In this chapter, we review two existing compressors for DNA sequencing data, namely FaStore [9] and CIGARCoil [10].

2.1 FaStore

FaStore is an algorithm proposed by Roguski et al. [9] for FASTQ file compression, i.e., it can compress both DNA sequences along with their identifiers and quality score. Here, we will only focus on its DNA sequence compression mechanism, which is based on the ORCOM [4] (for Overlapping Reads COmpression with Minimizers) algorithm with some improvements.

2.1.1 Minimizers and Signatures

ORCOM is an external memory algorithm that will first iterate through all the sequences in the input file, and dispatch them into bins so they can be compressed independently in parallel. It introduces the idea of minimizers and signatures to be able to cluster similar sequences together so they can be compressed more efficiently.

The minimizer for a sequence of length m is the lexicographically smallest of its k-mers. For example, the sequence GTAACGTT has five 4-mers: GTAA, TAAC, AACG, ACGT and CGTT. Among these, the lexicographically smallest one – assuming we use alphabetic order on the nucleobases – is AACG. It is therefore the minimizer for this sequence. As the letter N is way less frequent than the others, all sequences that have a minimizer containing at least one N letter are clustered together. This helps reduce the total number of possible minimizers from 5^k to $4^k + 1$. Additionally, a "skip zone" is introduced, which consists in ignoring the sequence suffixes of length z when looking for the minimizers (by default $z = 12$).

As DNA sequences can also be read backwards with their nucleotides complements – A becomes T, C becomes G and vice versa – each sequence is processed twice when searching for its minimizers.

This means that the minimizer is sought over $2(m - z - k + 1)$ k-mers for a given sequence of length m. These minimizers are called *canonical minimizers*.

Even distribution of sequences in bins is critical for compression, for both memory usage and efficient parallelization. In order to cluster sequences more evenly, in ORCOM, canonical minimizers are restricted by excluding those containing any of the triplets AAA, CCC, GGG, and TTT. FaStore goes one step further by enforcing neither to start with ACA and neither to contain AA anywhere except at the beginning of the minimizer. Such restricted minimizers are further called signatures.

2.1.2 Binning and Compression

The algorithm clusters sequences into bins, grouping them by their signatures. An extra sequence reordering step is applied independently for each bin. In order to move overlapping sequences close to each other, all sequences s in the bin are sorted according to the lexicographic order of the string $s[i : m] \circ s[1 : i - 1]$, where i is the position of the signature in the sequence, m is the length of the sequence and $\circ$ is the concatenation operator.

Compression is then processed on a per-bin basis. Several interleaving data streams are produced, and then compressed with a general purpose compressor. The goal is, for each bin, to construct a forest where each node corresponds to a sequence. Only root sequences will be hard coded, as the others will be encoded as the operations to be applied on their parent to reconstruct them.

While iterating through the sequences in a bin, a buffer – or sliding window – of n previous sequences is maintained. For each sequence, we look for the sequence in the buffer which maximizes the overlap. Both sequences are conceptually aligned on their signatures, and a distance measure is introduced, which consists of the weighted sum of the offset and the number of mismatches. For example, given sequences TTGCCTxxxxATAATTT and CATxxxxATGGTTTTAG where xxxx is their signature, by aligning them :

```
TTGCCTxxxxATATTT
   CATxxxxATGTTTTAG
```

we can see that they have an offset of three letters, and two mismatches (letter C becomes A before the signature and letter A becomes G after). Their distance is therefore $3 \times c_i + 2 \times c_m$ where c_i is the insertion cost (for the offset) and c_m is the mismatch cost. These values have been chosen experimentally to $c_i = 1$ and $c_m = 2$. For a given sequence, we search in the sliding window for the reference sequence that minimizes this distance.

Then, the compressed data is encoded into different data streams :

- *Flags* contains one of the following values for each sequence :
  - f_{copy} if the current sequence is identical to the previous one,
  - f_{diss} if the current sequence has no reference sequence because its similarity distance exceeds a specified threshold *max_dist* with all sequences in the sliding window,
  - f_{ex} if the current sequence has a reference sequence with no mismatch (only offset letters must be encoded),
  - f_{mis} if the current sequence has a reference sequence with exactly one mismatch at its last position,
  - f_{oth} if the current sequence has a reference sequence with one mismatch (not at its last position) or more.
- *Lengths* used to encode the sequence lengths if they are variables.
- *LettersX* for $X \in \{A, C, G, T, N\}$ (used for flags f_{ex}, f_{mis}, and f_{oth}) :

  - stores the mismatch in the current sequence, where the corresponding letter in the reference sequence is X (the alphabet size for these letters is 4 as it is $\{A, C, G, T, N\} \setminus \{X\}$),
  - if $X = N$, also store the trailing letters after the match

- *Prev* (used for flags f_{ex}, f_{mis}, and f_{oth}), stores the id of the reference sequence for the current one.
- *Shift* stores the offset of the current sequence against the reference (may be negative).
- *Matches* (used for flag f_{oth}) stores the mismatch positions. The signature is ignored as its position can be retrieved from *Shift*. The data is encoded as a form of run-length encoding, by storing a list of matching lengths.
- *HReads* (used for flag f_{diss}) stores the hard-coded sequences with no reference sequence, the signature is replaced with a special character to save more space.
- *Rev* indicates whether the current read is processed reverse-complemented.

As mentioned above, these stream are then compressed using a general purpose compressor. Decompression then consists in reading the *Flags* stream, taking from other streams depending on the flags read for each sequence. The advantage here is that decompression is a streaming operation, which is an important feature as data can be processed while it is being uncompressed.

2.2 CIGARCoil

CIGARCoil is an algorithm proposed by Womack et al. [10]. It is based on the idea of CIGAR strings [3], the representation of a DNA sequence as the operations needed to be applied on another sequence to reconstruct it, and the ReCoil compressor [11], which constructs a similarity graph between sequences. Its distinctive feature is to provide random-access to the DNA sequences directly from the compressed data.

2.2.1 CIGAR

The concept of CIGAR was first introduced by Fritz et al. [3]. It can be viewed as a list of operations to be applied on a sequence – or more formally, on a string defined on the alphabet $\Sigma = \{A, C, T, G, N\}$ – to construct a new one. There are four types of operations :

- M for match
- D for deletion
- I for insertion
- S for substitution

Operation M and D have an associated strictly positive integer value, corresponding to the length of the match or deletion. Operations I and S have an associated word defined on Σ, corresponding to the letters to be inserted or substituted.

Given a CIGAR string and a source sequence, the reconstruction process consists in iterating through the CIGAR operations and :

- if operation is M of length n, copy n letters from the source sequence to the output sequence,
- if operation is D of length n, move forward n letters in source sequence,
- if operation is I with word w, concatenate w to output sequence,
- if operation is S with word w, concatenate w to output sequence and move forward $|w|$ letters in source sequence.

For example, given the sequence **AAACCTGGG** and CIGAR **D[3]M[2]I[AA]S[G]M[3]**, the reconstruction process will consist of the following steps :

step	source sequence	CIGAR	output sequence
0	AAACCTGGG	D[3]M[2]I[AA]S[G]M[3]	
1	CCTGGG	M[2]I[AA]S[G]M[3]	
2	TGGG	I[AA]S[G]M[3]	CC
3	TGGG	S[G]M[3]	CCAA
4	GGG	M[3]	CCAAG
5			CCAAGGGG

which produces sequence **CCAAGGGG**.

In their original implementation, Womack et al. use two bytes – i.e., 16 bits – to encode a single CIGAR operation. The first bit is reserved, the next three bits indicate the type of operation, and the remaining twelve bits encode either up to four letters (three bits per letter) for operations S and I or an unsigned integer value for operations M and D.

CIGARCoil uses a modified version of the Wagner-Fischer algorithm to find the CIGAR string of minimal length between two sequences.

2.2.2 Wagner-Fischer Algorithm

The original Wagner-Fischer algorithm is a dynamic programming algorithm to compute the edit distance between two strings, i.e., the minimum number of single-character deletion, insertion or substitution operations to be applied on one string to obtain the other. More precisely, weights can be associated with each of these operations when computing the distance.

Let u and v be two words defined on a given alphabet. Let $u[: i]$ represent the substring formed by the i first letters of u. The Wagner-Fischer algorithm builds a matrix of size $|u|+1$ by $|v|+1$ where each cell $d_{i,j}$ will contain the distance between words $u[: i-1]$ and $v[: j-1]$. The cell $d_{1,1}$ is initialized to zero, and each cell is computed with respect to its upper-left neighbors according to the following rule :

$$d_{i,j} = \begin{cases} d_{i-1,j-1} & \text{if } u[i-1] = v[j-1] \\ min \begin{cases} d_{i-1,j} + C_d \\ d_{i,j-1} + C_i \\ d_{i-1,j-1} + C_s \end{cases} & \text{otherwise} \end{cases}$$

where C_d, C_i, and C_s, respectively correspond to the cost of a deletion, insertion and substitution operation. Once every cell in the matrix has been computed, the lower-right cell, $d_{|u|+1,|v|+1}$, contains the edit distance between u and v.

CIGARCoil uses a modified version of the Wagner-Fischer algorithm to compute the CIGAR string between two sequences. The notion of edit distance now corresponds to the length of the CIGAR string, i.e., the total number of operations including matches. Moreover, as operations in a CIGAR string can span across more than one letter, the algorithm is modified accordingly.

Concretely, the computation of cell $d_{i,j}$ depends on two cases :

- if letters $u[i]$ and $v[j]$ are equal, the CIGAR string in cell $d_{i-1,j-1}$ is copied and we look at its last operation :
  - if it's a match, then its value is incremented,
  - otherwise, a new match operation of value 1 is added,
- if letters $u[i]$ and $v[j]$ are different, we look at the new size of CIGAR strings in cells $d_{i-1,j-1}$, $d_{i,j-1}$ and $d_{i-1,j}$ if we add to them, respectively, a substitution, insertion or deletion operation (incrementing the operation when possible), and choose the smallest one.

For example, applying the modified Wagner-Fischer algorithm to find the minimal CIGAR string from sequence AAGGACCC to sequence GAAAACCCC would produce the matrix illustrated in Figure 2.1, and the final CIGAR string **S[GAAA]M[4]I[C]**.

We can note that due to the asymmetry between deletion and insertion operations, the distance is not symmetric anymore – and therefore cannot be called a distance in the mathematical sense – so the CIGAR string from u to v can be of

	ϵ	**A**	**A**	**G**	**G**	**A**	**C**	**C**	**C**
ϵ	0	1	1	1	1	2	2	2	2
G	1	1	2	2	2	2	3	3	3
A	1	2	1	2	2	3	2	3	3
A	1	2	2	1	2	3	3	2	3
A	1	2	2	2	1	2	2	2	2
A	1	2	2	2	2	2	3	3	3
C	1	2	2	2	2	3	2	3	3
C	1	2	2	2	2	3	3	2	3
C	1	2	2	2	2	3	3	3	2
C	1	2	2	2	2	3	3	3	3

Figure 2.1: Example of a modified Wagner-Fischer matrix

different length than the CIGAR string from v to u. However, in practice, the vast majority of minimal CIGAR strings have the same length from one sequence to the other and vice versa.

We can clearly see that the more similar two sequences are, the shorter the CIGAR string will be to express one against the other. It is with this idea in mind that, in CIGARCoil, a similarity graph is constructed between all the sequences from the input data.

2.2.3 Similarity Graph

The similarity graph idea consists in considering the sequences from the input data as nodes in an undirected graph where edges correspond to the length of the CIGAR string between two sequences.

From this graph, we compute a Minimum Spanning Tree (MST), to have a tree where the sum of the edge weights is minimal. As edge weights correspond to the length of the CIGAR strings, computing the MST minimizes the size of storing those, which is what we want to achieve in the context of compression.

This tree is then encoded and stored as the output data. Each internal node is stored as the CIGAR string to reconstruct it from the parent, and only the root node is stored as a plain sequence. Each sequence can then be reconstructed by successively applying CIGAR reconstruction on the path from the root of the tree to its corresponding node.

Yet, the question remains as to how to construct the graph from which we will produce the MST. A first naive approach would be to take the complete graph of all the sequences. However, since a complete graph with n nodes has $n(n-1)/2$ edges, and since the number of nodes here is the number of sequences, which is very large, this approach is not sustainable. That is why CIGARCoil introduces a heuristic to find a graph with limited number of edges, based on a *ad hoc* "hash-buckets" data structure.

2.2.4 Hash-Buckets Heuristic

This heuristic consists in constructing the n nodes graph by adding promising edges, that are likely to produce short CIGAR strings, i.e., between similar sequences. To this end, CIGARCoil introduces a "hash-buckets" data structure. The idea is to count the number of bases in each k-mer of a sequence (Womack et al. use the term "partition"). Given a sequence of length m and a number of partitions Δ, we count the number of letters A, C, G and T (letter N is rare enough to be ignored), in each partition of length $\lfloor \frac{m}{\Delta} \rfloor$. An example is illustrated in Figure 2.2.

Sequence	**Hash (#A,#C,#G,#T)**
AAACTGGCCT	[(3,1,0,1), (0,2,2,1)]
ACGTACGTAC	[(2,1,1,1), (1,2,1,1)]
TTTTTCCCCC	[(0,0,0,5), (0,5,0,0)]

Figure 2.2: Example of hash values for Δ=5

A three-dimensional array of buckets H is then constructed where $H[p][l][\delta]$ is the bucket that contains the list of all sequences that have δ letters l in partition p. An illustration of this data structure is provided in Figure 2.3.

Once all buckets are filled, for each sequence, we take the intersection of all

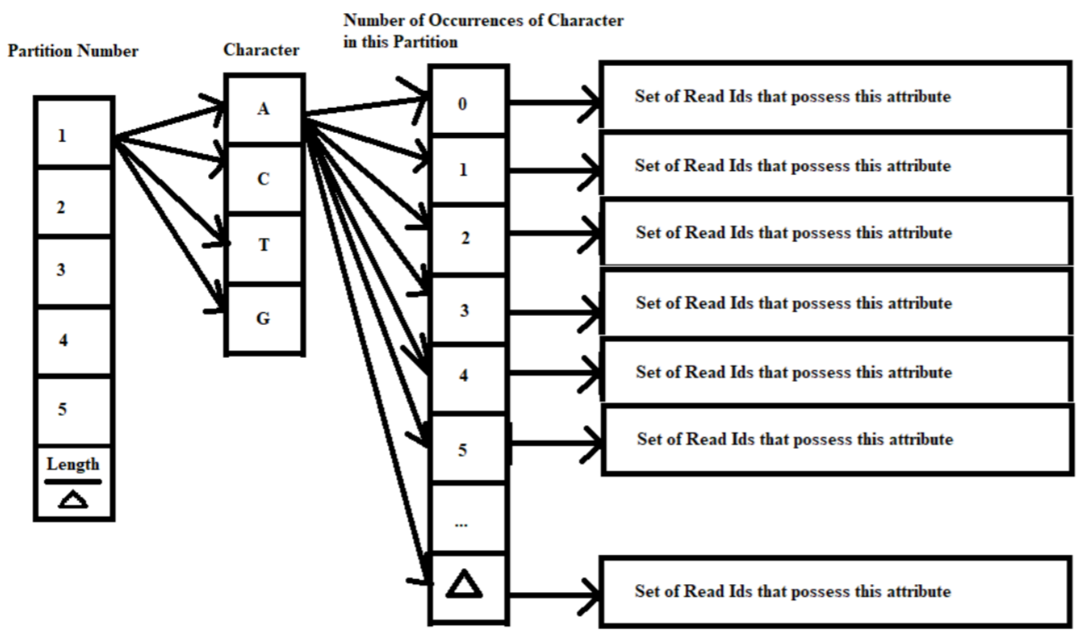

Partition Number
1
2
3
4
5
Length
Δ
Character
A
C
T
G
Number of Occurrences of Character in this Partition
0
1
2
3
4
5
...
Δ
Set of Read Ids that possess this attribute

Figure 2.3: Illustration of the hash buckets data structure

buckets it is in. This gives us a list of sequences that share the same number of letters in each partition. We then add the edges between each of the sequences to form the graph. If at some point in the intersection process, we have an empty set, we take the last non-empty intersection.

The hypothesis made here is that sequences that share the same number of letters in their partitions are more likely to be similar, and thus produce a smaller CIGAR string. This hypothesis is supported by the fact that, in modern DNA sequencing – so called "next-generation sequencing" – sequences are randomly sampled from the input genome, and consequently share a lot of substrings with other sequences.

This hash-buckets heuristic allows us to construct a graph which can then be used to compute the MST as mentioned above.

2.2.5 Decompression

As the compressed data is stored as a CIGAR tree with only the root node being stored as a plain sequence, decompressing simply consists of a traversal of the tree, reconstructing the sequence one after the other along the way. As the reconstruction process can be done in linear time with respect to the size of the sequence, the overall time complexity of decompression is $O(nm)$ where n is the total number of sequences and m is the length of a sequence.

Moreover, this structure provides us with partial decompression. CIGARCoil introduces a query operator to reconstruct a single sequence, by simply traversing the path from the root node to the sequence node, and applying the CIGAR reconstruction. It also offers a caching mechanism to be used for a repeated use of the query operator, and goes even one step further by using Q-learning for predictive caching. This offers significant improvement for working directly on the compressed file, and is a distinctive feature of the CIGARCoil algorithm.

Chapter 3

Multiple Attribute Tree Compression

In this chapter, we present our algorithm, MATC (for Multiple Attribute Tree Compression).

3.1 K-mer Huffman Compression

The first concept we introduce is the idea of per-k-mer Huffman compression. We split each sequence of length m in the input data in $\frac{m}{k}$ k-mers. If m is not divisible by k, we take its first $\lfloor \frac{m}{k} \rfloor$ k-mers, and its last (m mod k)-mer. Then, we apply a basic Huffman compression algorithm on a per-k-mer basis.

We count the number of occurrences of each k-mer to construct a Huffman tree. This gives us a coding table to associate each k-mer with a bit string. Then, we write in the output file the Huffman tree encoded via a pre-order traversal, detailed in Algorithm 1, as well as each sequence as the concatenation of the Huffman codes for each of its k-mers.

Algorithm 1: Encode Huffman Tree

```
Input  : T a Huffman tree
Output: Bit string encoding of T
if T is a leaf node then
    Write bit 1;
    Write k-mer of T;
else
    Write bit 0;
    Encode left child of T;
    Encode right child of T;
end
```

The idea of using Huffman compression on k-mers is to capture in the k-mers the overlaps of the sequences. As the sequences are randomly sampled from the same DNA fragment, the share common substrings that correspond to the areas in the reference genome where they overlap. The choice of the value of k – i.e. the length of the k-mers – is key, because if k is too small, k-mers will not be large enough to capture repeating substrings, but if k is too large, the variance of k-mers occurrence will plummet and the Huffman compression will not be efficient.

We made some experimental investigations to determine the best value of k. We tested this basic k-mer compression idea on two small datasets. The first one contained 1 million sequences of length 151, the second contained 8.2 million sequences of length 36. Compression size for different values of k for both are reported in Figures 3.1 and 3.2.

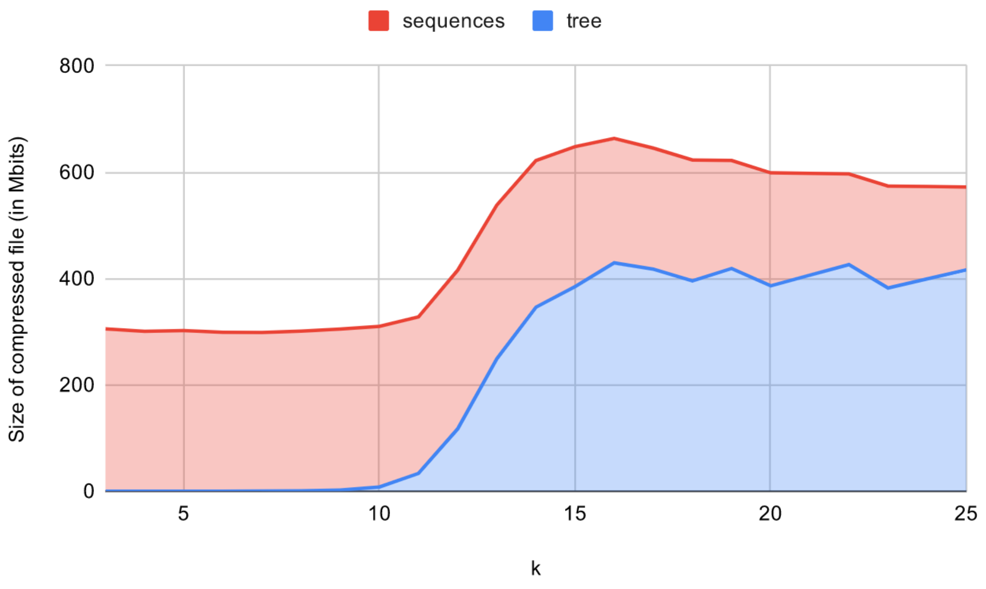


Figure 3.1: Evolution of compression size with respect to k for basic k-mer compression on a dataset of 1 million sequences of length 151. In red the total size of Huffman-encoded sequences, in blue the size of the Huffman tree.

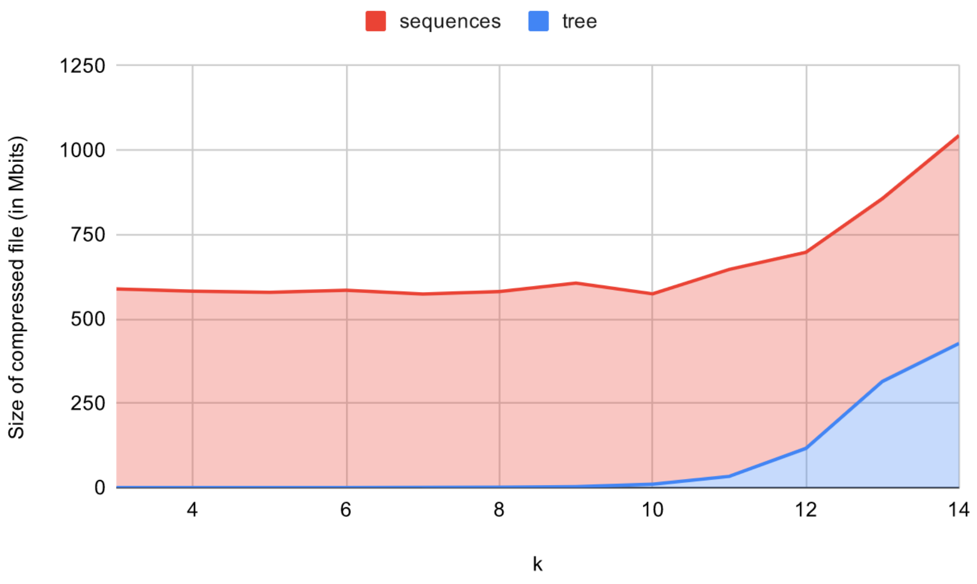


Figure 3.2: Evolution of compression size with respect to k for basic k-mer compression on a dataset of 8.2 million sequences of length 36. In red the total size of Huffman-encoded sequences, in blue the size of the Huffman tree.

These results show that the best value for k is between 6 and 10. We clearly see that after 10, the size of the Huffman tree grows rapidly, and causes the compression ratio to deteriorate.

Interestingly, with this simple idea alone, we achieve compression ratios comparable to those obtained with general purpose compressors such as gzip or bzip2 (i.e. around 30% of the original file size). This supports the argument that specialized compressors are much more suited for this kind of data.

3.2 Multiple Attribute Tree

We build upon the concept of k-mer Huffman compression by using a Multiple Attribute Tree (MAT) to represent the sequences. The idea is to construct a complete tree where each node represents a k-mer – except the root node – and each path from the root node to any leaf node represent the sequence formed by the concatenation of the k-mers of each node in the path. For example, let us consider the following sequences :

```
TTTTAGGATTTT
TTTTAGGACCAA
TTTTAGGAGGTC
AGGATTTTAGGA
AGGACCAATTTT
AGGACCAAAGGA
GGTCTTTTAGGA
```

Let $k = 4$, we obtain four different k-mers : K_1=TTTT, K_2=AGGA, K_3=CCAA, and K_4=GGTC. We can represent those sequences as the following list of k-mers :

- $K_1K_2K_1$
- $K_1K_2K_3$
- $K_1K_2K_4$
- $K_2K_1K_2$
- $K_2K_3K_1$

- $K_2K_3K_2$
- $K_4K_1K_2$

Then, we can construct the multiple attribute tree illustrated in Figure 3.2.

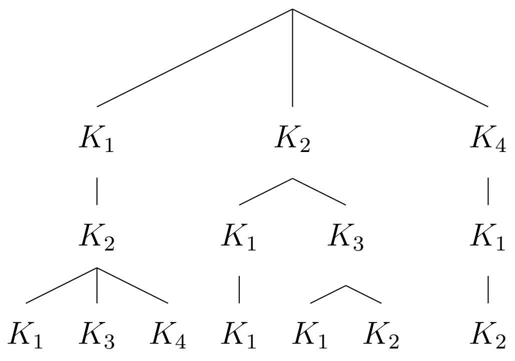


Figure 3.3: Example of a Multiple Attribute Tree

We construct the tree by inserting sequences one by one. The method for inserting a sequence – i.e. an array of k-mers – is detailed in Algorithm 2. The principle is, for each k-mer, to search in the current node's children for the current k-mer, and insert a new node if it was not found, while updating the current node to advance in the tree.

Algorithm 2: Insert k-mers in multiple attribute tree

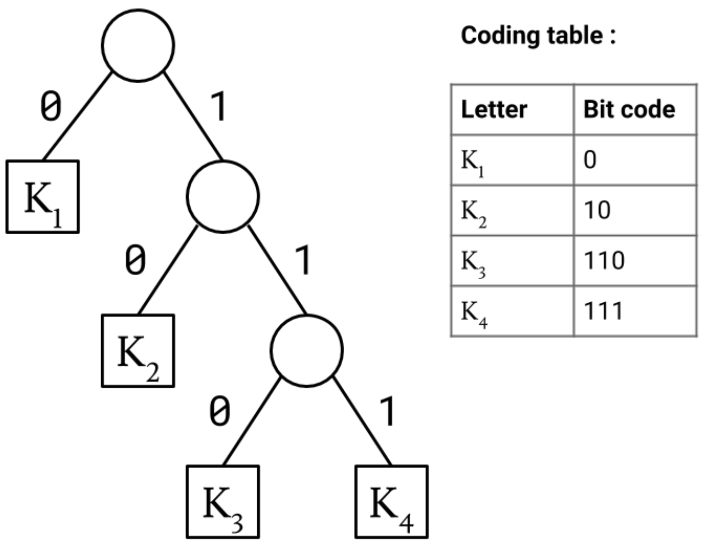
```
Input  : T a multiple attribute tree root node
Input  : kmers an array of k-mers representing a sequence
for each kmer in kmers do
    use binary search to find kmer in T.children
    if kmer was not found then
        insert new node in T.children for kmer
    end
    T ← child node of kmer
end
```

As we create the tree, we count the number of occurrences of each k-mer for Huffman encoding. We then build the Huffman tree and corresponding k-mer coding table.

The output data consists of the encoded Huffman tree and multiple attribute tree. For example, the multiple attribute tree illustrated above in Figure 3.2 would produce the following k-mer frequencies :

- $K_1 \to 6$
- $K_2 \to 4$
- $K_3 \to 2$
- $K_4 \to 2$

which produces the Huffman tree and coding table in Figure 3.4.



Coding table :

Letter	Bit code
K_1	0
K_2	10
K_3	110
K_4	111

Figure 3.4: Huffman tree produced by k-mer frequencies in Figure 3.2.

The depth-first traversal encoding of this Huffman tree would be: $01K_101K_201K_31K_4$, where K_i is replaced by its hard-coded sequence (3 bits per base).

Then, to encode the multiple attribute tree, we use a depth-first traversal where we encode each node's k-mer with the Huffman coding table, followed by its number of children. We detail this process in Algorithm 3.

Algorithm 3: Encode multiple attribute tree

```
Input  : T a multiple attribute tree root node
Input  : Huffman coding table
Encode size(T.children)
S ← new stack initialized with nodes of T.children
for size(S) > 0 do
    T ← S.pop()
    Encode T.kmer with coding table
    if size(T.children) > 0 then
        Encode size(T.children)
        for child in T.children do
            S.push(child)
        end
    end
end
```

The number of children of each node is encoded using a variable-width scheme. We noticed that single-child nodes are extremely frequent, while nodes with a large number of children are rare. More specifically, the deeper the node is in the tree, the less children it has. This can be explained by the fact that the children of a given node in the tree represent all the sequences that have all k-mers from the root node to this node. Therefore, we encode the number of children n of a given internal node with the following rule :

- if $n = 1$, then write 0,
- if $\log n < 4$, then write 10, and n encoded on 4 bits,
- if $\log n < 8$, then write 110, and n encoded on 8 bits,
- etc.

The number of ones before the first zero tells us on how many chunks of 4 bits is the number encoded, with a special case for 1. This method has significantly reduced the size taken by the number of children – i.e. the multiple attribute tree structure – in the output file.

For example, given the multiple attribute tree in Figure 3.2, its depth-first traversal encoding would produce the following output:
$3K_1 1K_2 3K_1 K_3 K_4 K_2 2K_1 1K_1 K_3 2K_1 K_2 K_4 1K_1 1K_2$. Using the Huffman coding table in Figure 3.4, and our children number encoding scheme, the resulting bits would be (with spaces between each number and k-mer):

```
100011 0 0 10 100011 0 110 111 10 100010 0 0 0 110 100010 0 10
111 0 0 0 10
```

3.3 Complexity Analysis

We can see that the time complexity for compression depends on three steps :

- multiple attribute tree construction
- Huffman tree construction
- multiple attribute tree encoding

Let n be the number of sequences in the input file, m the length of those sequences, and k the length of the k-mers. To construct the multiple attribute tree, we need to insert each sequence in the tree. Each insertion takes constant time, as it is linear with respect to the height of the tree $\frac{m}{k}$, which is constant. As we need to do n insertions, multiple attribute tree construction can be done in $O(n)$ time.

Huffman tree construction complexity depends on the number of different k-mers in the file. This number n_{k-mers} is bounded by 5^k because the alphabet size is 5. As this number is constant, Huffman tree construction can be done in constant time (more precisely in $O(n_{k-mers} \log n_{k-mers})$ time which is constant).

Finally, multiple attribute tree encoding can be done in $O(n)$ as it simply consists of a depth-first traversal of the tree, and each k-mer encoding can be done in constant time if we store the coding table as a hash table.

In summary, the time complexity of our algorithm is $O(n)$, i.e., linear with respect to the number of sequences.

In terms of space complexity, our algorithm is particularly memory-intensive, as it needs to store the whole multiple attribute tree in memory. This gives us a space complexity of $O(n)$, as it is equivalent to storing all sequences in memory. One way to address this is to use an external memory approach. However we rather decided to opt for a distributed approach, as we will see in the following chapter.

3.4 Decompression

Decompression consists of the following steps. First, we read and decode the Huffman tree, which allows us to decode the k-mers. Then we decode the multiple attribute tree by decoding the number of children of each node, followed by its k-mer, except for leaf nodes which we know do not have any children. As a multiple attribute tree is a complete tree, we can know that we have reached a leaf node by keeping track of the depth. The recursive process is detailed in Algorithm 4.

Algorithm 4: Decompression

```
Input  : HT, a Huffman tree
Input  : height, the height of the multiple attribute tree
Input  : depth, the current depth, initially 0
Input  : kmers, the current array of k-mers, initially empty
Output: stream of sequences
$n \leftarrow$ Decode number of children;
for $i$ from 1 to $n$ do
    kmer $\leftarrow$ Decode k-mer with HT;
    kmers.append(kmer);
    if $depth < height$ then
        Decompression(HT, height, depth+1, kmers);
    else
        construct sequence from kmers and write it to output;
```

We note that decompression is a streaming operation, meaning that sequences can be processed as they are decompressed. We do not need to reconstruct the whole multiple attribute tree, as we just need to follow the depth-first traversal of the tree. We believe it is an important feature of our algorithm.

3.5 Implementation

Our algorithm was implemented using the Go programming language.

Go (or golang) is a programming language developed at Google by Robert Griesemer, Rob Pike, and Ken Thompson in 2007. The initial motivation was to create a language in-between systems and application programming, for highly concurrent and scalable software. Performance and developer-productivity are core concerns of the language.

Although Go does not seem to be highly adopted by the research community, its expressiveness and simplicity – allowing fast source code iterations, which are very common in research – encourage us to advocate for its broader adoption.

Go is a compiled, statically typed, and garbage collected language, with a C-inspired syntax. It also offers a CSP-style concurrent paradigm, via *green threads* called "goroutines".

Green threads are lightweight threads handled one abstraction layer above operation system threads. They are usually created and managed by a virtual machine – such as the Java Virtual Machine (JVM) or the Erlang virtual machine (BEAM) – or, in the case of Go, a runtime, whereas standard threads are handled by the operating system. Each Go program, when compiled, ships with a small runtime that handles garbage collection and goroutines management. The advantage of green threads over operation system threads is that they have a much smaller footprint, especially memory-wise (a goroutine initially starts with a stack size of only 2kB), which means that many of them can be heavily created and destroyed without affecting performances.

For example, in our implementation, a goroutine is responsible for reading the sequences in the input file, and sends them via a channel – a first-class value in

Go which allows goroutine memory communication – to the main goroutine which processes them. This means that our program does not wait for I/O operations as it handles sequences asynchronously. Moreover, we found out that adding an additional goroutine in the middle, that converts lines from the input file to arrays of k-mers, further improved performances.

Implementation of multiple attribute tree construction was done using a dedicated data structure. Inserting k-mers in the multiple attribute tree was done recursively using binary search on each node's children. However, we found out that splitting the top level k-mers in a hash table improved performances. The hash table associates to each top k-mer (i.e. first k-mer of each sequence) a multiple attribute tree. When encoding the whole multiple attribute tree, we first encode the number of top k-mers, and each k-mer in the hash table followed by its encoded tree.

As we needed to read and write at the bit level, we used an open-source library called go-bitstream (`https://github.com/dgryski/go-bitstream`). More precisely, we forked this Go package to modify it to better suit our needs. We namely added a bit buffer data structure based on the bit stream reader and writer provided by this package. We also added buffered versions of these bit stream reader and writer.

Chapter 4

Distributed Multiple Attribute Tree Compression

In this chapter, we present a distributed implementation for our algorithm.

4.1 Master-Worker Paradigm

Our distributed implementation follows the well-known master-worker paradigm. A single master process reads the input file, and distributes the compression work to several worker processes. These processes can run on the same machine, or on a cluster of machines communicating over the network.

We note that the construction of the multiple attribute tree can actually be viewed as the construction of a forest, as the root node does not store any k-mer. Therefore, our idea consists in dispatching the sequences among the workers depending on the first k-mer of each sequence. When a sequence is read, the master assigns its first k-mer to a specific worker so that all other sequences with the same first k-mer will be sent to this worker.

The question that arises from this idea is how to ensure an even distribution of the sequences. To solve this problem, we assign to each worker a *workload* variable that corresponds to the number of sequences previously sent to this worker. This variable is initialized to zero and incremented each time a sequence is sent to this worker. Then, if the first k-mer of a newly read sequence as no assigned worker, we assign it to the worker for which the *workload* value is minimal.

Each worker will receive the sequences sent by the master and build its own attribute tree. By design, as no worker share the same k-mer at the first level of the tree, the concatenation of all the trees stored by each worker forms the complete multiple attribute tree for the entire input file.

However, the Huffman tree for k-mer encoding cannot be constructed by a single worker, as it depends on the frequency of k-mers in all trees. A synchronisation step is thus necessary. After dispatching the sequences to the workers, the master process receives the k-mer frequencies from all workers, construct the Huffman tree by merging them, and sends it back to all workers. Each worker then encodes its multiple attribute tree and sends it to the master, which writes them to the output file.

This communication protocol is illustrated in Figure 4.1.

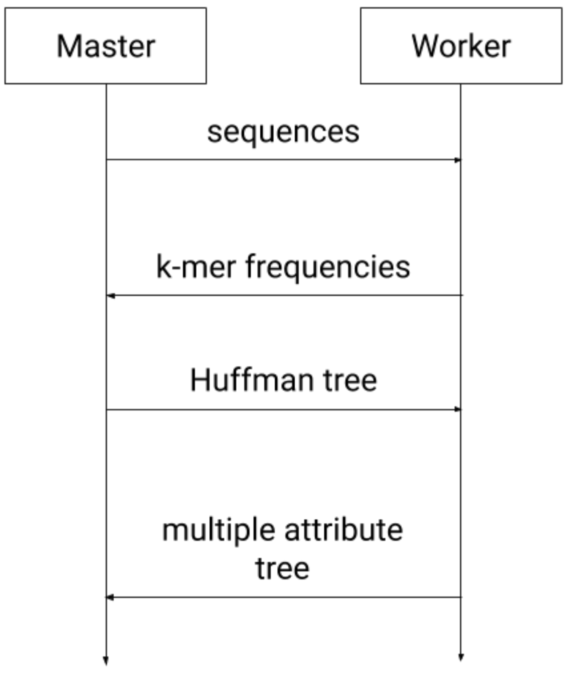


Figure 4.1: Master-worker protocol for distributed MATC

4.2 Algorithms

We detail algorithms for the master and worker processes in Algorithms 5 and 6.

Algorithm 5: Master

```
workers ← connect to each worker;
workloads ← new array of size of size(workers);
dispatchedKmers ← new hash table (kmer → worker);
for sequence in inpute file do
    kmer ← first kmer in sequence;
    if kmer is in dispatchedKmers then
        send sequence to worker dispatchedKmers[kmer];
    else
        worker ← workers[min(workloads)];
        send sequence to worker;
        dispatchedKmers[kmer] ← worker;
    end
end
kmerFreq ← receive k-mer frequencies;
huffmanTree ← build Huffman tree from kmerFreq;
for worker in workers do
    send huffmanTree to worker;
end
write number of top k-mers in output file;
for worker in workers do
    mat ← receive multiple attribute tree from worker;
    write mat to output file;
end
```

Algorithm 6: Worker

```
master $\leftarrow$ listen for incoming connection;
mat $\leftarrow$ initialize multiple attribute tree;
kmerFreq $\leftarrow$ new hash table (kmer $\rightarrow$ unsigned integer);
for each sequence from master do
    insert sequence in mat and update kmerFreq;
end
send kmerFreq to master;
huffmanTree $\leftarrow$ receive Huffman tree from master;
encode mat with huffmanTree and send it to master;
```

4.2.1 Speedup

In parallel computing, speedup is defined as the ratio of the sequential execution time to the parallel execution time. It is usually expressed as $S_p = \frac{T(n,1)}{T(n,p)}$. We saw in the previous chapter that our sequential algorithm runs in $O(n)$. Our parallel algorithm distributes the multiple attribute tree construction and encoding to all workers. The only sequential part is the Huffman tree construction, which, as we saw, can be done in constant time (as it only depends on the number of different k-mers, which is bounded by the constant 5^k) i.e., the sequential part that needs to be synchronised is in $O(5^k)$. Therefore, the parallel execution time is $T(n,p) = O(\frac{n}{p})$, which gives us an ideal linear speedup $S_p = p$. Our algorithm is thus highly parallelizable.

Naturally, in practice, the number of different k-mers does grow with the number of sequences. We investigated this growth experimentally, by counting the number of different k-mers (with $k = 10$) for a number of sequences between 10 and 60 million (with sequences of length 100). Results are shown in Figure 4.2. We can see that doubling the number of sequences in the input data only increases the number of different k-mers by a factor of around 8%.

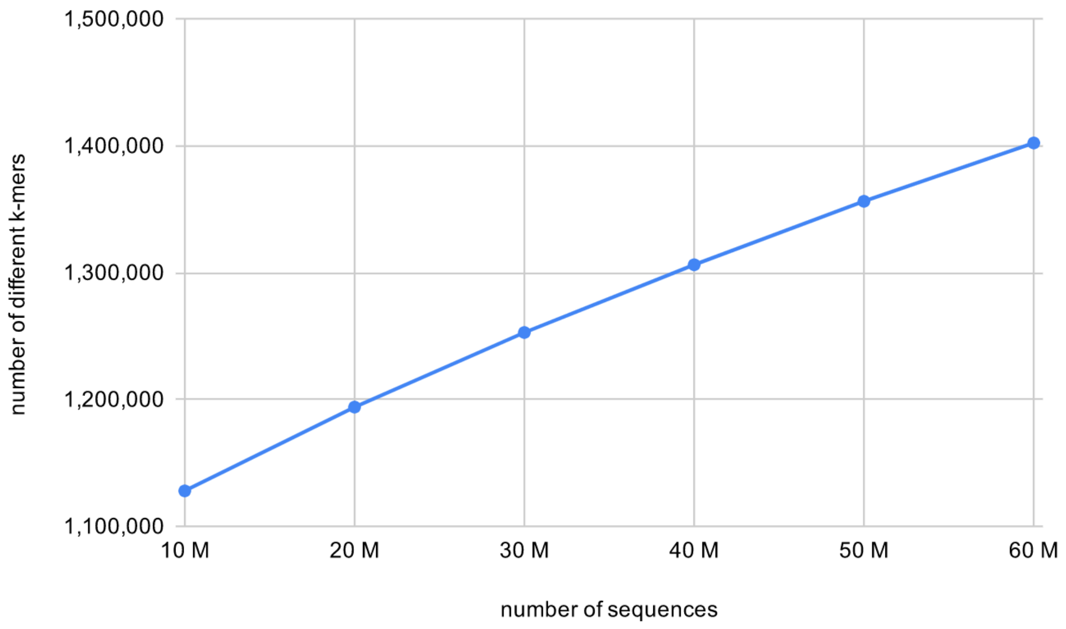


Figure 4.2: Evolution of the number of different k-mers with respect to the number of sequences on the CE dataset (sequence length = 100; k = 10).

4.3 Implementation

In our implementation, the worker process is a TCP server waiting for a connection from a master process. The master process acts as a TCP client and connects to the different worker processes.

Interestingly, an important feature of the Go programming language, interfaces, has greatly facilitated our implementation of a distributed version of MATC. In Go, a file structure implements the io.Reader and io.Writer interfaces from the standard library, which means that we can read and write bytes from/to a file. As we have previously mentioned, in our implementation, we use a modified version of an open-source library for manipulating bit streams. These streams are based on the io.Reader and io.Writer interfaces to read from/to a file.

In Go, a TCP connection also implements these interfaces, and can therefore be used, as is, as a bit stream. For example, when the master sends the Huffman tree to the worker, we use the exact same code for encoding the Huffman tree to the output file, except that we encode it to the TCP connection. Each worker can then decode the Huffman tree as it would from a compressed file. Another interesting application is when workers send their multiple attribute trees. The master simply redirects this bit stream to the output file.

Chapter 5

Results

In this chapter, we detail our experimental results on several datasets.

5.1 Datasets

In our experiments, we used a subset of the datasets used by FaStore [9], as in their paper, they provide results for all major DNA compressors, which we used to compare with our algorithm.

These datasets consists of gzipped FASTQ files, which were extracted and concatenated. We then removed identifiers and quality scores from the resulting files, to only keep DNA sequences.

5.1.1 CE Dataset

This dataset consists of two FASTQ files, which can be downloaded from :

```
ftp://ftp.sra.ebi.ac.uk/vol1/fastq/SRR065/SRR065390/SRR065390_1.
    fastq.gz
ftp://ftp.sra.ebi.ac.uk/vol1/fastq/SRR065/SRR065390/SRR065390_2.
    fastq.gz
```

5.1.2 WEX Dataset

This dataset consists of a single BAM file, which can be downloaded from :

```
ftp://ftp-trace.ncbi.nlm.nih.gov/giab/ftp/data/AshkenazimTrio/
   HG002_NA24385_son/OsloUniversityHospital_Exome/151002
   _7001448_0359_AC7F6GANXX_Sample_HG002-EEogPU_v02-KIT-
   Av5_AGATGTAC_L008.posiSrt.markDup.bam
```

We then used SAMTools [7] to convert it to a FASTQ file.

5.1.3 GG Dataset

This dataset consists of twelve FASTQ files, which can be downloaded from :

```
ftp://ftp.ddbj.nig.ac.jp/ddbj_database/dra/fastq/SRA030/SRA030308/
   SRX043656/SRR105788_1.fastq.bz2
ftp://ftp.ddbj.nig.ac.jp/ddbj_database/dra/fastq/SRA030/SRA030308/
   SRX043656/SRR105788_2.fastq.bz2
ftp://ftp.ddbj.nig.ac.jp/ddbj_database/dra/fastq/SRA030/SRA030309/
   SRX043656/SRR105789_1.fastq.bz2
ftp://ftp.ddbj.nig.ac.jp/ddbj_database/dra/fastq/SRA030/SRA030309/
   SRX043656/SRR105789_2.fastq.bz2
ftp://ftp.ddbj.nig.ac.jp/ddbj_database/dra/fastq/SRA030/SRA030312/
   SRX043656/SRR105792_1.fastq.bz2
ftp://ftp.ddbj.nig.ac.jp/ddbj_database/dra/fastq/SRA030/SRA030312/
   SRX043656/SRR105792_2.fastq.bz2
ftp://ftp.ddbj.nig.ac.jp/ddbj_database/dra/fastq/SRA030/SRA030314/
   SRX043656/SRR105794_1.fastq.bz2
ftp://ftp.ddbj.nig.ac.jp/ddbj_database/dra/fastq/SRA030/SRA030314/
   SRX043656/SRR105794_2.fastq.bz2
ftp://ftp.ddbj.nig.ac.jp/ddbj_database/dra/fastq/SRA036/SRA036382/
   SRX043656/SRR197985_1.fastq.bz2
ftp://ftp.ddbj.nig.ac.jp/ddbj_database/dra/fastq/SRA036/SRA036382/
   SRX043656/SRR197985_2.fastq.bz2
ftp://ftp.ddbj.nig.ac.jp/ddbj_database/dra/fastq/SRA036/SRA036383/
   SRX043656/SRR197986_1.fastq.bz2
```

```
ftp://ftp.ddbj.nig.ac.jp/ddbj_database/dra/fastq/SRA036/SRA036383/
    SRX043656/SRR197986_2.fastq.bz2
```

5.1.4 WGS-14 Dataset

This dataset is a subset of a larger dataset called WGS-235x. It consists of its first two FASTQ files which can be downloaded from :

```
ftp://ftp.sra.ebi.ac.uk/vol1/fastq/ERR174/ERR174324/ERR174324_1.
    fastq.gz
ftp://ftp.sra.ebi.ac.uk/vol1/fastq/ERR174/ERR174324/ERR174324_2.
    fastq.gz
```

5.1.5 Summary

We provide in Table 5.1 a detailed summary of the datasets we used, including sequence length, number of sequences, and the raw size of the resulting file containing only the DNA sequences.

Dataset	**Organism**	**Seq. length**	**Seq. count**	**Raw size**
CE	C. Elegans	100	67.62 M	6.83 GB
WEX	H. Sapiens	126	151.80 M	19.28 GB
GG	G. Gallus	100	347.74 M	35.12 GB
WGS-14	H. Sapiens	101	447.14 M	45.61 GB

Table 5.1: Summary of the datasets used in our experiments

5.2 Compression Ratio and Speed Evolution

5.2.1 Non-distributed MATC

We studied the evolution of compression ratio and speed with respect to the number of sequences in the input file. Our approach was to randomly sample sequences from the CE dataset to produce artificial files with different number of sequences. We produced 6 files containing between 10 and 60 million sequences. We then ran our algorithm on a single machine for each file. Results are reported in Figure 5.1.

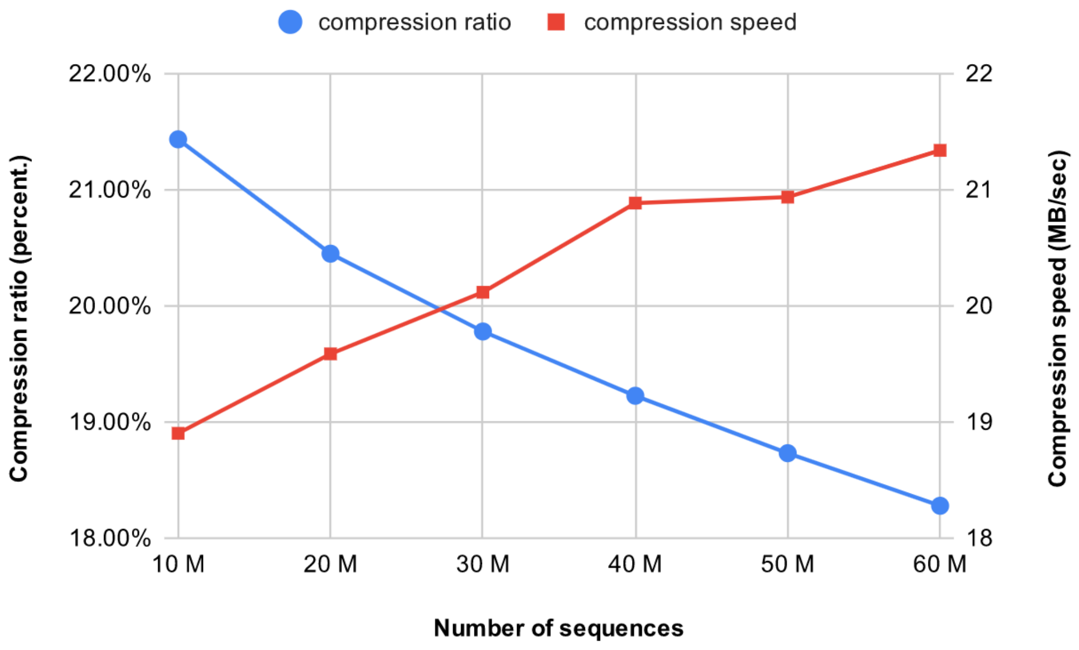


Figure 5.1: Evolution of the compression ratio and speed with respect to the number of sequences randomly sampled from the CE dataset

We can clearly see that compression ratio decreases as the number of sequences – i.e., improves, as lower is better. Likewise, the compression speed increases with the number of sequences.

This results suggest that the bigger the input file, the more efficient our algorithm will be both in terms of compression ratio and speed.

5.2.2 Distributed MATC

We also studied how our distributed algorithm behaves with respect to the number of workers. We ran our algorithm on the full CE dataset with between 2 and 10 workers, each running in a single machine on a cluster of 10 machines with each 16 GB of RAM and 1 vCPU. The evolution of compression speed is reported in Figure 5.2.

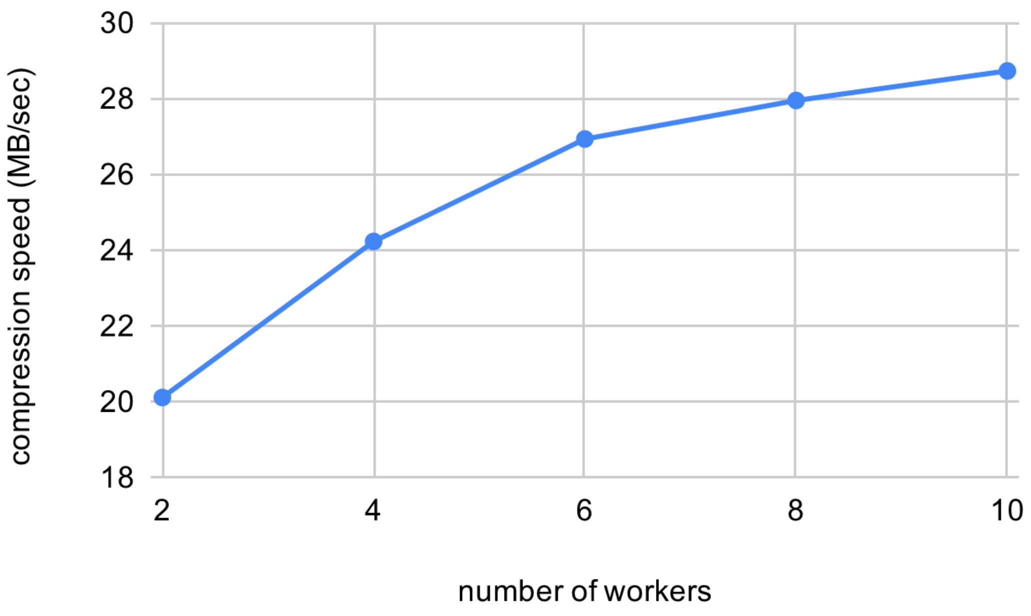


Figure 5.2: Evolution of the compression time with respect to the number of workers for our distributed algorithm on the CE dataset. [6.83GB; 67.62M sequences]

Doubling the number of workers seems to increase the compression speed by around 15%. However we can see that as the number of workers grow, the improvement diminishes. This can be explained by the growing number of different k-mers, which increases the time taken by the Huffman tree construction, as we saw in Section 4.2.1, and by the increasing cost of network communication. We also studied the evolution of network communication cost, results are shown in Figures 5.3 and 5.4. We can see that these variations are quite small, representing less than 1% of the total data transfer. They are actually only due to the Huffman tree synchronisation step. Indeed, the master needs to receive k-mer frequencies and send the Huffman tree to all workers, hence more workers means more network load. However, since the size of the k-mer frequencies and Huffman tree is quite small, the does not induce excessive growth of network communication costs.

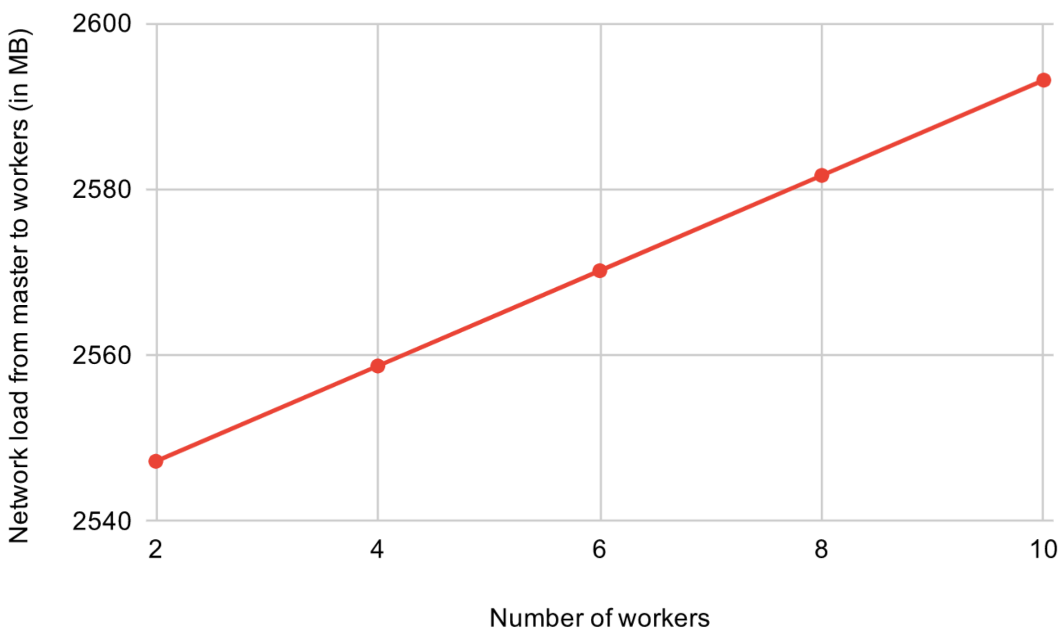


Figure 5.3: Evolution of the network load from master to workers with respect to the number of workers for our distributed algorithm on the CE dataset. [6.83GB; 67.62M sequences]

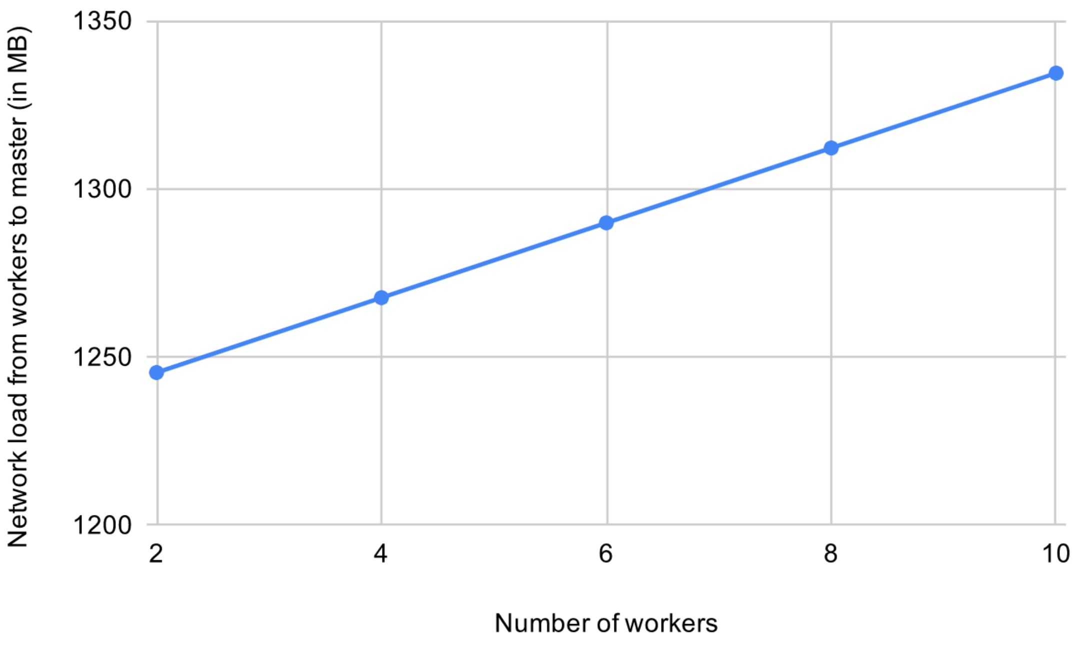


Figure 5.4: Evolution of the network load from workers to master with respect to the number of workers for our distributed algorithm on the CE dataset. [6.83GB; 67.62M sequences]

5.3 Comparison with other DNA Compressors

5.3.1 Experimental Setup

We ran our distributed implementation of MATC on all datasets, in the cloud, on a cluster of 10 machines with each 16 GB of RAM and 1 vCPU of an Intel® Xeon® Gold 6140 @2.30GHz processor, which can be considered as commodity hardware as they are single-core machines.

We set $k = 10$ for all datasets, except for the WEX dataset for which $k = 9$ (because the sequences length, 126, was divisible by 9).

The detailed results, including compression ratio – i.e., the size of the output file divided by the size of the input file, in percentage – and compression speed – i.e., the size of the input file divided by the compression time – are provided in Table 5.2.

Dataset	**In. size**	**Out. size**	**Time** (sec)	**Comp. Ratio**	**Speed** (MB/sec)
CE	6.83 GB	1.23 GB	237.69	17.97%	28.44
WEX	19.28 GB	2.98 GB	530.01	15.44%	35.75
GG	35.12 GB	6.68 GB	1614.17	19.01%	21.51
WGS-14	45.61 GB	9.15 GB	2014.20	20.07%	22.42

Table 5.2: MATC compression ratio and speed for all datasets

5.3.2 Compression Ratio

We compare our compression ratio for each dataset with other compressors in Table 5.3 and Figure 5.5. Namely, we compare compression ratio with the following DNA compressors :

- FaStore [9]
- Scalce [5]
- LEON [1]
- Fqzcomp [2]
- Quip [6]
- DSRC 2 [8]

as well as the general purpose compressors bzip2 and gzip.

Results for those DNA compressors were taken from the FaStore paper supplementary materials. Results for bzip2 and gzip were run by us.

	Datasets			
Compressors	**CE**	**WEX**	**GG**	**WGS-14**
MATC	17.97%	15.44%	19.01%	20.07%
FaStore	6.34%	5.38%	8.41%	10.28%
Scalce	7.94%	6.81%	10.30%	11.98%
LEON	8.31%	9.04%	12.21%	15.93%
Fqzcomp	13.19%	11.01%	17.74%	18.74%
Quip	21.37%	22.14%	21.69%	21.70%
DSRC 2	23.03%	23.93%	22.75%	22.76%
bzip2	27.61%	28.04%	26.98%	27.40%
gzip	29.88%	30.17%	29.31%	29.90%

Table 5.3: Comparison of compression ratio with several other compressors for all datasets

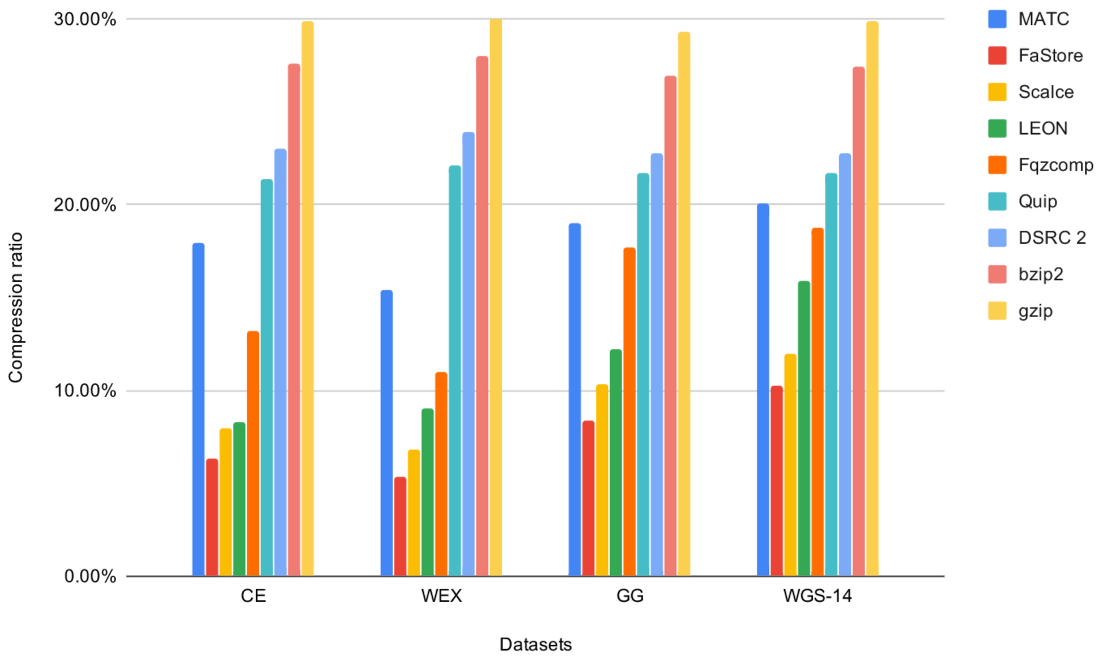


Figure 5.5: Comparison of compression ratio with several other compressors for all datasets

These results show that, in terms of compression ratio, our algorithm performs worse than FaStore, Scalce, LEON and Fqzcomp. However, our algorithm also performs better than Quip and DSRC 2, as well as bzip2 and gzip.

5.3.3 Compression Speed

Compression speed is harder to compare for two reasons. First, it depends on the hardware on which the algorithm is executed. Secondly, as our algorithm is distributed, comparing its speed with non-distributed algorithm is less legitimate.

We report the compression speeds provided in the FaStore paper for the aforementioned algorithms in Table 5.4.

	Datasets			
Compressors	**CE**	**WEX**	**GG**	**WGS-14**
FaStore	6.0	8.4	8.3	7.6
Scalce	39.4	42.8	49.0	43.2
LEON	18.1	16.3	19.7	15.6
Fqzcomp	13.9	18.0	24.3	14.0
Quip	46.2	50.6	60.2	41.9
DSRC 2	127.8	159.2	169.6	124.3

Table 5.4: Compression speed of other compressors for all datasets (in MB/sec)

According to the FaStore paper, these compression speeds were obtained on an 8-core machine. This type of setup is comparable to our cluster of 10 machine with 1 vCPU each. As we report compression speeds in Table 5.2 between 21.51 and 35.75 MB/sec, it is reasonable to believe that our algorithm performs on par with these other compressors, if not faster than those which have better compression ratio.

5.3.4 Decompression Speed

Likewise, decompression speed is reported in Table 5.5. These results were obtained on a single machine from our cluster of single-core workstations.

Dataset	**Decompression speed** (MB/sec)
CE	24.92
WEX	32.58
GG	23.74
WGS-14	22.66

Table 5.5: MATC decompression speed for all datasets

Chapter 6

Conclusion

In this thesis, we proposed MATC, a novel compression algorithm for DNA sequencing data based on multiple attribute trees. Our algorithm offers streaming decompression, which allows programs to start processing sequences as they are uncompressed.

We provided both local and distributed implementation for this algorithm, allowing compression to happen in the cloud on a cluster of machines. We have shown that our algorithm perform moderately well in terms of compression ratio, around 18%, while other specialized compressors perform between 6% and 24%, but with fairly good compression speed thanks to its distributed implementation.

We believe that there are still rooms for improvements for our algorithm. Notably, we tackled its high memory complexity with a distributed approach, but we believe an external memory approach would also be worth considering.

Bibliography

[1] Gaëtan Benoit, Claire Lemaitre, Dominique Lavenier, Erwan Drezen, Thibault Dayris, Raluca Uricaru, and Guillaume Rizk. Reference-free compression of high throughput sequencing data with a probabilistic de bruijn graph. *BMC bioinformatics*, 16(1):288, 2015.

[2] James K Bonfield and Matthew V Mahoney. Compression of fastq and sam format sequencing data. *PloS one*, 8(3):e59190, 2013.

[3] Markus Hsi-Yang Fritz, Rasko Leinonen, Guy Cochrane, and Ewan Birney. Efficient storage of high throughput dna sequencing data using reference-based compression. *Genome research*, 21(5):734–740, 2011.

[4] Szymon Grabowski, Sebastian Deorowicz, and Łukasz Roguski. Disk-based compression of data from genome sequencing. *Bioinformatics*, 31(9):1389–1395, 12 2014.

[5] Faraz Hach, Ibrahim Numanagić, Can Alkan, and S Cenk Sahinalp. Scalce: boosting sequence compression algorithms using locally consistent encoding. *Bioinformatics*, 28(23):3051–3057, 2012.

[6] Daniel C Jones, Walter L Ruzzo, Xinxia Peng, and Michael G Katze. Compression of next-generation sequencing reads aided by highly efficient de novo assembly. *Nucleic acids research*, 40(22):e171–e171, 2012.

[7] Heng Li, Bob Handsaker, Alec Wysoker, Tim Fennell, Jue Ruan, Nils Homer, Gabor Marth, Goncalo Abecasis, and Richard Durbin. The sequence alignment/map format and samtools. *Bioinformatics*, 25(16):2078–2079, 2009.

[8] Łukasz Roguski and Sebastian Deorowicz. DSRC 2—Industry-oriented compression of FASTQ files. *Bioinformatics*, 30(15):2213–2215, 04 2014.

[9] Łukasz Roguski, Idoia Ochoa, Mikel Hernaez, and Sebastian Deorowicz. FaStore: a space-saving solution for raw sequencing data. *Bioinformatics*, 34(16):2748–2756, 03 2018.

[10] Addison Womack. Cigarcoil: A new algorithm for the compression of dna sequencing data. Master's thesis, University of Oklahoma, Norman, Oklahoma, 2019.

[11] Vladimir Yanovsky. Recoil-an algorithm for compression of extremely large datasets of dna data. *Algorithms for Molecular Biology*, 6(1):23, 2011.

Appendix A

Source code

Source code for MATC was written in the Go programming language. It mainly consists of two Go packages : the matc library and the executable. Each is composed of several ".go" files.

A.1 MATC library

A.1.1 letter.go

```
package matc

import "fmt"

const (
  // LetterBitSize correspond to the number of bits
  // required to encode a single letter
  LetterBitSize = 3
  // LetterMask has its LetterBitSize least-significant
  // bits set to ones
  LetterMask = 0x7

  // these are the DNA sequencing letters from the
  // alphabet {A, C, G, T, N}
  LetterN uint8 = iota
  LetterA
  LetterC
  LetterG
  LetterT
)

// LetterFromRune converts a rune (character) to its letter value
func LetterFromRune(r rune) (l uint8, ok bool) {
  switch r {
  case 'N':
    return LetterN, true
  case 'A':
    return LetterA, true
  case 'C':
    return LetterC, true
  case 'G':
    return LetterG, true
  case 'T':
    return LetterT, true
  }
  return 0, false
}

// LetterFromRune converts a letter value to a rune (character)
func LetterToRune(l uint8) rune {
  switch l {
  case LetterN:
    return 'N'
  case LetterA:
    return 'A'
  case LetterC:
    return 'C'
  case LetterG:
    return 'G'
  case LetterT:
    return 'T'
  }
  panic(fmt.Sprintf("cannot convert letter to rune: invalid letter
     value ‘%02x‘", l))
}
```

A.1.2 kmer.go

```
package matc

import (
  "bufio"

  "github.com/dselim/go-bit"
)

// Kmer represents a k-mer encoded on a uint32
// each letter is encoded on LetterBitSize bits
type Kmer uint32

// maxK represents the maximum value of k (10)
const maxK = 32 / LetterBitSize

// KmersFromString converts a sequences string
// to a slice of k-mers
func KmersFromString(str string, k int) (kmers []Kmer, ok bool) {
  n := len(str)
  nk := n / k
  kr := n % k
  if kr > 0 {
    kmers = make([]Kmer, nk+1)
  } else {
    kmers = make([]Kmer, nk)
  }
  for i := 0; i < nk; i++ {
    kmers[i], ok = KmerFromString(str[i*k : (i+1)*k])
    if !ok {
      return nil, false
    }
  }
  if kr > 0 {
    kmers[nk], ok = KmerFromString(str[nk*k:])
    if !ok {
      return nil, false
    }
  }
  return
}

// KmerFromString converts a sequences to a single k-mer
func KmerFromString(str string) (Kmer, bool) {
  var kmer Kmer
  for _, r := range str {
    kmer <<= LetterBitSize
    l, ok := LetterFromRune(r)
    if !ok {
      return 0, false
    }
    kmer += Kmer(l)
  }
  return kmer, true
}

// Len returns the number of letters in this k-mer
func (kmer Kmer) Len() (n int) {
  for ; kmer != 0; kmer >>= LetterBitSize {
    n++
  }
  return
}

// EncodeBits writes the k-mer to the provided bit.Writer
func (kmer Kmer) EncodeBits(w bit.Writer, k int) (n int, err error)
    {
  n = LetterBitSize * k
  err = w.WriteBits(uint64(kmer), n)
  return
}

// EncodeBits decodes the k-mer from the provided bit.Writer
func (kmer *Kmer) DecodeBits(r bit.Reader, k int) (err error) {
  if k > maxK {
    panic("k too large for storing kmer in uint32")
  }
  var v uint64
  v, err = r.ReadBits(LetterBitSize * k)
  if err != nil {
    return
  }
  *kmer = Kmer(uint32(v))
  return
}

var bytebuf = make([]byte, 0, 10)

// WriteToBuffer writes the k-mer as a plain string
// to the provided buffered writer
func (kmer Kmer) WriteToBuffer(w *bufio.Writer) error {
  bytebuf = bytebuf[:0]
  for l := kmer & LetterMask; l != 0; l = kmer & LetterMask {
    bytebuf = append(bytebuf, byte(l))
    kmer >>= LetterBitSize
  }
  for i := len(bytebuf) - 1; i >= 0; i-- {
    _, err := w.WriteRune(LetterToRune(bytebuf[i]))
    if err != nil {
      return err
    }
  }
  return nil
}

// String converts the k-mer to a plain string
func (kmer Kmer) String() string {
  s := ""
  for ; kmer != 0; kmer >>= LetterBitSize {
    s = s + string(LetterToRune(uint8(kmer&LetterMask)))
  }
  return s
}
```

A.1.3 file_io.go

```go
package matc

import (
  "bufio"
  "errors"
  "os"
  "runtime"
)

// ScanFileFirst reads the first sequence in the input file
// and returns its k-mers as well as its length
func ScanFileFirst(filepath string, k int) ([]Kmer, int, error) {
  f, err := os.Open(filepath)
  if err != nil {
    return nil, 0, err
  }
  defer f.Close()

  scanner := bufio.NewScanner(f)
  for scanner.Scan() {
    kmers, ok := KmersFromString(scanner.Text(), k)
    if !ok {
      continue
    }
    return kmers, len(scanner.Text()), nil
  }
  return nil, 0, errors.New("no valid sequence in input stream")
}

// ScanFileFirst reads all sequences in the input file
// and returns a channel in which it will send batches
// of the k-mers for each sequence
func ScanFileKmers(filepath string, batchSize, k int) <-chan [][]
   Kmer {
  resc := make(chan [][]Kmer, runtime.NumCPU())
  linec := scanLines(filepath, batchSize)
  go func() {
    defer close(resc)
    batch := make([][]Kmer, 0, batchSize)
    seqLen := -1
    for lines := range linec {
      for _, line := range lines {
        kmers, ok := KmersFromString(line, k)
        if !ok {
          continue
        }
        if seqLen == -1 {
          seqLen = len(line)
        } else if len(line) != seqLen {
          panic("found sequences of different lengths")
        }
        batch = append(batch, kmers)

        if len(batch) >= batchSize {
          resc <- batch
          batch = make([][]Kmer, 0, batchSize)
        }
      }
    }
    if len(batch) > 0 {
      resc <- batch
    }
  }()
  return resc
}

// scanLines reads all lines in the input file
// and sends them by batch of batchSize in
// the channel it returns
func scanLines(filepath string, batchSize int) <-chan []string {
  linec := make(chan []string, runtime.NumCPU())
  go func() {
    defer close(linec)
    f, err := os.Open(filepath)
    if err != nil {
      panic(err)
    }
    defer f.Close()

    scanner := bufio.NewScanner(f)
    lines := make([]string, 0, batchSize)
    for scanner.Scan() {
      lines = append(lines, scanner.Text())
      if len(lines) >= batchSize {
        linec <- lines
        lines = make([]string, 0, batchSize)
      }
    }
    if len(lines) > 0 {
      linec <- lines
    }
    err = scanner.Err()
    if err != nil {
      panic(err)
    }
  }()
  return linec
}
```

A.1.4 huffman_tree.go

```go
package matc

import (
  "errors"

  "github.com/dgryski/go-bitstream"
  "github.com/dselim/go-bit"
  go_heaps "github.com/theodesp/go-heaps"
  "github.com/theodesp/go-heaps/fibonacci"
)

// HuffmanTree represents a Huffman tree node
// for internal nodes: kmer == 0 and left and right are defined
// for leaf nodes: kmer != 0 and left and right are nil
type HuffmanTree struct {
  kmer  Kmer
  left  *HuffmanTree
  right *HuffmanTree
}

// heapElem encapsulates a Huffman tree node
// for priority queue
type heapElem struct {
  priority int
  tree     *HuffmanTree
}

// Compare implements the go_heaps.Item interface
func (e heapElem) Compare(other go_heaps.Item) int {
  return e.priority - other.(heapElem).priority
}

// BuildHuffmanTree builds a Huffman tree from
// k-mer frequencies (or occurences) using a
// fibonacci heap as a priority queue data structure
func BuildHuffmanTree(kmers map[Kmer]int) *HuffmanTree {
  queue := fibonacci.New()
  n := len(kmers)
  for kmer, occ := range kmers {
    queue.Insert(heapElem{
      occ,
      &HuffmanTree{kmer: kmer},
    })
  }
  for n > 1 {
    n--
    e1 := queue.DeleteMin().(heapElem)
    e2 := queue.DeleteMin().(heapElem)
    queue.Insert(heapElem{
      e1.priority + e2.priority,
      &HuffmanTree{
        left:  e1.tree,
        right: e2.tree,
      },
    })
  }
  return queue.DeleteMin().(heapElem).tree
}

// Bits represent a bitstring of at most 64 bits
type Bits struct {
  n int
  v uint64
}

// Size returns the length of the bitstring
func (b Bits) Size() int {
  return b.n
}

// AddBit adds a bit to the bitstring and returns
// its new value
func (b Bits) AddBit(bit bitstream.Bit) (bits Bits) {
  v := b.v << 1
  if bit == bitstream.One {
    v++
  }
  return Bits{
    n: b.n + 1,
    v: v,
  }
}

// EncodeBits writes the bitstring to a bit.Writer
// it returns the number of bits written
func (b Bits) EncodeBits(w bit.Writer) (n int, err error) {
  n = b.n
  err = w.WriteBits(b.v, b.n)
  return
}

// CodingTable associate with each k-mer in the
// Huffman tree its corresponding bitstring
type CodingTable map[Kmer]Bits

// CodingTable computes the coding table for
// a Huffman tree
func (t *HuffmanTree) CodingTable() CodingTable {
  type stackElem struct {
    t *HuffmanTree
    b Bits
  }
  stack := []stackElem{stackElem{t: t}}

  table := make(CodingTable)
  for len(stack) > 0 {
    elem := stack[len(stack)-1]
    stack = stack[:len(stack)-1]

    if elem.t.kmer != 0 {
      table[elem.t.kmer] = elem.b

    } else {
      stack = append(stack, stackElem{
        t: elem.t.right,
        b: elem.b.AddBit(bitstream.One),
      }, stackElem{
        t: elem.t.left,
        b: elem.b.AddBit(bitstream.Zero),
      })
    }
  }
  return table
}

// EncodeKmer writes the Huffman code for the
// provided k-mer to a bit.Writer
// it returns the number of bits written
// and throws an error if the k-mer is not
// in the coding table
func (t CodingTable) EncodeKmer(w bit.Writer, kmer Kmer) (int,
   error) {
  bits, ok := t[kmer]
  if !ok {
    return 0, errors.New("encode kmer: coding table does not
       contain kmer")
  }
  return bits.n, w.WriteBits(bits.v, bits.n)
}

// DecodeKmer uses the Huffman tree to read
// a k-mer from the provided bit.Reader
func (t *HuffmanTree) DecodeKmer(r bit.Reader) (Kmer, error) {
  cur := t
  for cur.kmer == 0 {
    b, err := r.ReadBit()
    if err != nil {
      return 0, err
    }
    if b == bit.Zero {
      cur = cur.left
    } else {
      cur = cur.right
    }
  }
  return cur.kmer, nil
}

// EncodeBits encodes the Huffman tree using a
// depth-first traversal encoding
func (t *HuffmanTree) EncodeBits(w bit.Writer, k int) (n int, err
   error) {
  var m int
  stack := []*HuffmanTree{t}
  for len(stack) > 0 {
    tree := stack[len(stack)-1]
    stack = stack[:len(stack)-1]
    if tree.kmer != 0 {
      err = w.WriteBit(bit.One)
      if err != nil {
        return
      }
      n++
      m, err = tree.kmer.EncodeBits(w, k)
      if err != nil {
        return
      }
      n += m
    } else {
      err = w.WriteBit(bit.Zero)
      if err != nil {
        return
      }
      n++
      stack = append(stack, tree.right, tree.left)
    }
  }
  return
}

// DecodeBits decodes a Huffman tree from the
// provided bit.Reader
func (t *HuffmanTree) DecodeBits(r bit.Reader, k int) (err error) {
  stack := []*HuffmanTree{t}
  for len(stack) > 0 {
    tree := stack[len(stack)-1]
    stack = stack[:len(stack)-1]

    var b bit.Bit
    b, err = r.ReadBit()
    if err != nil {
      return
    }
    if b == bit.One {
      err = tree.kmer.DecodeBits(r, k)
      if err != nil {
        return
      }
    } else {
      tree.right = new(HuffmanTree)
      tree.left = new(HuffmanTree)
      stack = append(stack, tree.right, tree.left)
    }
  }
  return
}
```

A.1.5 var_uint64.go

```
package matc

import (
  "github.com/dselim/go-bit"
)

// pow2 return 2^p
func pow2(p uint64) (v uint64) {
  v = 1
  for p > 0 {
    v *= 2
    p--
  }
  return
}

const widthPerOne = 4

// EncodeVarWidthUint64 encodes v in the provided bit.Writer
// as a variable width unsigned integer such that :
// - if n = 1, then write 0
// - if logn < widthPerOne, then write 10, and v encoded on
   widthPerOne bits
// - if logn < 2*widthPerOne, then write 10, and v encoded on 2*
   widthPerOne bits
// - etc.
func EncodeVarWidthUint64(w bit.Writer, v uint64) (n int, err error
   ) {
  if v == 0 {
    n, err = 1, w.WriteBit(bit.Zero)
    return
  }
  ones := 1
  for v >= pow2(uint64(ones*widthPerOne)) {
    ones++
  }
  err = w.WriteBits(^uint64(0), ones)
  if err != nil {
    return 0, err
  }
  err = w.WriteBit(bit.Zero)
  if err != nil {
    return 0, err
  }
  n = ones + 1 + ones*widthPerOne
  err = w.WriteBits(v, ones*widthPerOne)
  return
}

// DecodeVarWidthUint64 decodes from the provided bit.Reader
// a variable width unsigned integer encoded with
   EncodeVarWidthUint64
func DecodeVarWidthUint64(r bit.Reader) (uint64, error) {
  ones := 0
  b, err := r.ReadBit()
  if err != nil {
    return 0, err
  }
  for b == bit.One {
    ones++
    b, err = r.ReadBit()
    if err != nil {
      return 0, err
    }
  }
  if ones == 0 {
    return 0, nil
  }
  return r.ReadBits(ones * widthPerOne)
}
```

A.1.6 mat.go

```
package matc

import (
  "sort"

  "github.com/dselim/go-bit"
)

// MAT represents a multiple attribute tree
type MAT struct {
  Kmer
  Children []MAT
}

// AddLeaf adds the provided k-mers to the multiple
// attribute tree, and updates the provided k-mer
// frequency hash-maps
func (t *MAT) AddLeaf(kmers []Kmer, kmerOcc, trailKmerOcc map[Kmer]
   int) {
  if len(kmers) == 0 {
    return
  }
  i, found := t.SearchChild(kmers[0])
  if !found {
    // insert
    t.Children = append(t.Children, MAT{})
    copy(t.Children[i+1:], t.Children[i:])
    t.Children[i] = MAT{Kmer: kmers[0]}

    if len(kmers) == 1 && trailKmerOcc != nil {
      trailKmerOcc[kmers[0]]++
    } else {
      kmerOcc[kmers[0]]++
    }
  }
  t.Children[i].AddLeaf(kmers[1:], kmerOcc, trailKmerOcc)
  return
}

// SearchChild searches in the MAT node's children
// for a provided k-mer using binary search
// it returns the index at which the k-mer was
// found or should be inserted
func (t MAT) SearchChild(kmer Kmer) (int, bool) {
  if len(t.Children) < 8 {
    for i, child := range t.Children {
      if child.Kmer >= kmer {
        return i, child.Kmer == kmer
      }
    }
    return len(t.Children), false
  }
  i := sort.Search(len(t.Children), func(i int) bool { return t.
     Children[i].Kmer >= kmer })
  return i, i < len(t.Children) && t.Children[i].Kmer == kmer
}

// EncodeBits encodes the multiple attribute tree
// to the provided bit.Writer, using the provided
// coding tables to encode the k-mers
func (t MAT) EncodeBits(w bit.Writer, table, trailTable CodingTable
   ) (nkmers, nchildren int, err error) {
  var m int
  m, err = EncodeVarWidthUint64(w, uint64(len(t.Children)-1))
  if err != nil {
    return
  }
  nchildren += m

  stack := t.Children
  for len(stack) > 0 {
    t = stack[len(stack)-1]
    stack = stack[:len(stack)-1]
    if len(t.Children) == 0 && trailTable != nil {
      m, err = trailTable.EncodeKmer(w, t.Kmer)
    } else {
      m, err = table.EncodeKmer(w, t.Kmer)
    }
    if err != nil {
      return
    }
    nkmers += m

    if len(t.Children) > 0 {
      m, err = EncodeVarWidthUint64(w, uint64(len(t.Children)-1))
      if err != nil {
        return
      }
      nchildren += m

      stack = append(stack, t.Children...)
    }
  }
  return
}

// MATDecodeBits decodes a multiple attribute tree from the
// provided bit.Reader, using the provided Huffman trees
// calling fn for each decoded sequence
func MATDecodeBits(r bit.Reader, hft, trailHft *HuffmanTree, height
    int, fn func([]Kmer) error) error {
  return matDecodeBitsRec(r, hft, trailHft, fn, make([]Kmer, height
     ), 0)
}

// matDecodeBitsRec is the recursive function
// called by MATDecodeBits
func matDecodeBitsRec(r bit.Reader, hft, trailHft *HuffmanTree, fn
   func([]Kmer) error, kmers []Kmer, i int) error {
```

```
  v, err := DecodeVarWidthUint64(r)
  if err != nil {
    return err
  }
  numChildren := int(v) + 1
  for j := 0; j < numChildren; j++ {
    if i == len(kmers)-1 && trailHft != nil {
      kmers[i], err = trailHft.DecodeKmer(r)
    } else {
      kmers[i], err = hft.DecodeKmer(r)
    }
    if err != nil {
      return err
    }

    if i < len(kmers)-1 {
      err = matDecodeBitsRec(r, hft, trailHft, fn, kmers, i+1)
      if err != nil {
        return err
      }
    } else {
      err = fn(kmers)
      if err != nil {
        return err
      }
    }
  }
  return nil
}
```

A.2 Executable

A.2.1 main.go

```
package main

import (
  "fmt"
  "log"
  "os"
  "time"

  "github.com/spf13/pflag"
)

var (
  k            int
  batchSize    int
  inFile       string
  outFile      string
  uncompress   bool
  workersAddr  []string
  listenAddr   string
)

func init() {
  pflag.IntVarP(&k, "kmer-length", "k", 0, "length of K-mers")
  pflag.IntVarP(&batchSize, "batch-size", "b", 10000, "size of
     batches for goroutine communication")
  pflag.StringVarP(&inFile, "in", "i", "", "input file")
  pflag.StringVarP(&outFile, "out", "o", "", "output file")
  pflag.BoolVarP(&uncompress, "uncompress", "x", false, "uncompress
      mode")
  pflag.StringArrayVarP(&workersAddr, "worker", "w", nil, "worker
     addresses (distributed master mode)")
  pflag.StringVarP(&listenAddr, "listen", "l", "", "address to
     listen to (distributed worker mode)")
```

```
}

func main() {
  pflag.Parse()

  if listenAddr != "" {
    err := runWorker()
    if err != nil {
      log.Fatalln(err)
    }
    return
  }

  if inFile == "" || outFile == "" {
    pflag.Usage()
    return
  }
  if uncompress {
    decompression(runUncompress)
    return
  }
  if k == 0 {
    pflag.Usage()
    return
  }
  if len(workersAddr) > 0 {
    compression(runMaster)
  } else {
    compression(runCompress)
  }
}

// compression runs the provided function
// and logs the compression size, ratio,
// time, and speed
func compression(runFunc func() error) {
  log.Println("compressing file", inFile)
  t := time.Now()

  err := runFunc()
  if err != nil {
    log.Fatalln(err)
  }

  dt := time.Since(t)
  inSize, err := fileSize(inFile)
  if err != nil {
    log.Fatalln("could not open input file")
  }
  outSize, err := fileSize(outFile)
  if err != nil {
    log.Fatalln("could not open output file")
  }
  percent := float64(outSize) / float64(inSize) * 100
  ratio := float64(inSize) / float64(outSize)
  speed := (float64(inSize) / 1000000) / dt.Seconds()
  log.Println("---------- SUMMARY ----------")
  log.Println("compression time:", dt)
  log.Println("output file size:", outSize/1000000, "MB")
  log.Println("compression speed:", speed, "MB/seconds")
  log.Println("compression ratio:", ratio)
  log.Println("compression percent:", percent, "%")
}

// decompression runs the provided function
// and logs the decompression time and speed
func decompression(runFunc func() error) {
  log.Println("uncompressing file", inFile)
  t := time.Now()

  err := runFunc()
  if err != nil {
    log.Fatalln(err)
  }
  dt := time.Since(t)
  outSize, err := fileSize(outFile)
  if err != nil {
    panic(err)
  }
  outSizeMega := float64(outSize) / 1000000
  speed := outSizeMega / dt.Seconds()
  log.Println("---------- SUMMARY ----------")
  log.Println("compression time:", dt.Seconds(), "seconds")
  log.Println("decompression speed:", speed, "MB/seconds")
}

// fileSize returns the size in bytes of
// the file at filepath
func fileSize(filepath string) (int64, error) {
  f, err := os.Open(filepath)
  if err != nil {
    return 0, err
  }
  stat, err := f.Stat()
  if err != nil {
    return 0, err
  }
  return stat.Size(), nil
}

// Clock is a utility for logging the time
// taken by some tasks
type Clock time.Time

func NewClock() *Clock {
  clock := Clock(time.Now())
  return &clock
}

func (c Clock) Logf(format string, v ...interface{}) {
  log.Println(fmt.Sprintf(format, v...), "in:", time.Since(time.
     Time(c)))
}

func (c *Clock) LogfReset(format string, v ...interface{}) {
  c.Logf(format, v...)
  *c = Clock(time.Now())
}
```

A.2.2 run_compress.go

```go
package main

import (
  "fmt"
  "log"
  "os"

  "github.com/dselim/cigarcoil/matc"

  "github.com/dselim/go-bit"
)

// runCompress locally compresses a file with the
// MATC algorithm
func runCompress() error {
  clock := NewClock()

  _, n, err := matc.ScanFileFirst(inFile, k)
  if err != nil {
    return err
  }
  nk := n / k
  if nk < 2 {
    panic(fmt.Sprintf("nk is %v, should be at least 2", nk))
  }
  kr := n % k
  log.Println("n", n, "k", k, "nk", nk, "kr", kr)

  numSeq := 0
  kmerOcc := make(map[matc.Kmer]int)
  var trailKmerOcc map[matc.Kmer]int
  if kr > 0 {
    trailKmerOcc = make(map[matc.Kmer]int)
  }
  mats := make(map[matc.Kmer]*matc.MAT)
  for batch := range matc.ScanFileKmers(inFile, batchSize, k) {
    for _, kmers := range batch {
      numSeq++
      if numSeq == 1000000 || numSeq%10000000 == 0 {
        clock.Logf("read %vM sequences", numSeq/1000000)
      }

      t, ok := mats[kmers[0]]
      if !ok {
        t = new(matc.MAT)
        mats[kmers[0]] = t
        kmerOcc[kmers[0]]++
      }
      t.AddLeaf(kmers[1:], kmerOcc, trailKmerOcc)
    }
  }
  clock.LogfReset("read all sequences")

  hft := matc.BuildHuffmanTree(kmerOcc)
  var trailHft *matc.HuffmanTree
  if kr > 0 {
    trailHft = matc.BuildHuffmanTree(trailKmerOcc)
  }
  kmerOcc = nil
  trailKmerOcc = nil
  clock.LogfReset("computed huffman tree(s)")

  codingTable := hft.CodingTable()
  var trailCodingTable matc.CodingTable
  if kr > 0 {
    trailCodingTable = trailHft.CodingTable()
  }
  clock.LogfReset("computed coding table(s)")

  // open ouput file for writing
  // and create buffered bit writer
  f, err := os.Create(outFile)
  if err != nil {
```

```
  return err
}
defer f.Close()
bw := bit.NewBufWriter(f)
defer bw.Flush()

// write headers to file
err = bw.WriteBits(uint64(n), 16)
if err != nil {
  return err
}
err = bw.WriteBits(uint64(k), 8)
if err != nil {
  return err
}

// write huffman trees
_, err = hft.EncodeBits(bw, k)
if err != nil {
  return err
}
if kr > 0 {
  _, err = trailHft.EncodeBits(bw, kr)
  if err != nil {
    return err
  }
}
hft = nil
trailHft = nil

// write multiple attribute trees
_, err = matc.EncodeVarWidthUint64(bw, uint64(len(mats)-1))
if err != nil {
  return err
}
matWritten := 0
for kmer, mat := range mats {
    delete(mats, kmer)

    _, err = codingTable.EncodeKmer(bw, kmer)
    if err != nil {
      return err
    }
    _, _, err = mat.EncodeBits(bw, codingTable, trailCodingTable)
    if err != nil {
      return err
    }

    matWritten++
    if matWritten%100000 == 0 {
      clock.Logf("wrote %vk multiple attribute trees", matWritten
         /1000)
    }

    mat = nil
  }
  clock.LogfReset("wrote all multiple attribute trees")

  return nil
}
```

A.2.3 run_uncompress.go

```
package main

import (
  "bufio"
  "os"

  "github.com/dselim/cigarcoil/matc"
  "github.com/dselim/go-bit"
)

// runUncompress locally decompresses a file with the
// MATC algorithm
func runUncompress() error {
  clock := NewClock()

  f, err := os.Open(inFile)
  if err != nil {
    return err
  }
  defer f.Close()
  br := bit.NewBufReader(f)

  v, err := br.ReadBits(16)
  if err != nil {
    return err
  }
  n := int(v)
  v, err = br.ReadBits(8)
  if err != nil {
    return err
  }
  k := int(v)
  clock.LogfReset("read headers")

  nk := n / k
  kr := n % k
```

```
height := nk
if kr > 0 {
  height++
}

hft := new(matc.HuffmanTree)
err = hft.DecodeBits(br, k)
if err != nil {
  return err
}
var trailHft *matc.HuffmanTree
if kr > 0 {
  trailHft = new(matc.HuffmanTree)
  err = trailHft.DecodeBits(br, kr)
  if err != nil {
    return err
  }
}
clock.LogfReset("read huffman tree(s)")

f, err = os.Create(outFile)
if err != nil {
  return err
}
defer f.Close()
bw := bufio.NewWriter(f)
defer bw.Flush()

matc.MATDecodeBits(br, hft, trailHft, height, func(kmers []matc.
   Kmer) error {
  for _, kmer := range kmers {
    err := kmer.WriteToBuffer(bw)
    if err != nil {
      return err
    }
  }
  return bw.WriteByte('\n')
  })
  clock.LogfReset("decoded sequences")

  return nil
}
```

A.2.4 run_master.go

```
package main

import (
  "fmt"
  "log"
  "net"
  "os"

  "github.com/dselim/cigarcoil/matc"
  "github.com/dselim/go-bit"
)

type Worker struct {
  conn net.Conn
  bit.ReadWriter
}

// runMaster compresses a file with the
// distributed MATC algorithm by connecting
// to worker processes over TCP
func runMaster() error {
  clock := NewClock()
  var kmer matc.Kmer

  _, n, err := matc.ScanFileFirst(inFile, k)
  if err != nil {
    return err
  }
  nk := n / k
  if nk < 2 {
    panic(fmt.Sprintf("nk is %v, should be at least 2", nk))
  }
  kr := n % k
  log.Println("n", n, "k", k, "nk", nk, "kr", kr)

  workers := make([]*Worker, len(workersAddr))
```

```
for i, addr := range workersAddr {
  conn, err := net.Dial("tcp", addr)
  if err != nil {
    return err
  }
  defer conn.Close()
  workers[i] = &Worker{
    conn:       conn,
    ReadWriter: bit.NewBufReadWriter(bit.NewBufReader(conn), bit.
       NewBufWriter(conn)),
  }
}

for _, worker := range workers {
  b, err := worker.ReadByte()
  if err != nil {
    return err
  }
  if b == 0 {
    return fmt.Errorf("worker ‘%v‘ is already running", worker.
       conn.RemoteAddr())
  }

  err = worker.WriteBits(uint64(n), 16)
  if err != nil {
    return err
  }
  err = worker.WriteBits(uint64(k), 8)
  if err != nil {
    return err
  }

  if err != nil {
    return err
  }
  err = worker.Flush()
  if err != nil {
    return err
  }
}
clock.LogfReset("established connections with workers")

numSeq := 0
workerKmers := make(map[matc.Kmer]int)
workerLoads := make([]int, len(workers))
kmers := make([]matc.Kmer, nk)
if kr > 0 {
  kmers = append(kmers, 0)
}
for batch := range matc.ScanFileKmers(inFile, batchSize, k) {
  for _, kmers := range batch {
    numSeq++
    if numSeq == 1000000 || numSeq%10000000 == 0 {
      clock.Logf("dispatched %vM sequences", numSeq/1000000)
    }

    i, hasAttrWorker := workerKmers[kmers[0]]
    if !hasAttrWorker {
      i, _ = minInt(workerLoads)
      workerLoads[i]++
      workerKmers[kmers[0]] = i
    }
    // send these kmers to worker i
    for j := 0; j < nk; j++ {
      _, err = kmers[j].EncodeBits(workers[i], k)
      if err != nil {
        return err
      }
    }
    if kr > 0 {
      _, err = kmers[nk].EncodeBits(workers[i], kr)
      if err != nil {
        return err
      }
    }
  }
}
numTopKmers := len(workerKmers)
workerKmers = nil
workerLoads = nil
kmer = 0
for _, worker := range workers {
  _, err = kmer.EncodeBits(worker, k)
  if err != nil {
    return err
  }
  err = worker.Flush()
  if err != nil {
    return err
  }
}
clock.LogfReset("dispatched kmers")
log.Println("number of top kmers:", numTopKmers)

kmerOcc := make(map[matc.Kmer]int)
var trailKmerOcc map[matc.Kmer]int
if kr > 0 {
  trailKmerOcc = make(map[matc.Kmer]int)
}
for _, worker := range workers {
  err = recvKmerOcc(worker, k, kmerOcc)
  if err != nil {
    return err
  }
  if kr > 0 {
    err = recvKmerOcc(worker, kr, trailKmerOcc)
    if err != nil {
      return err
    }
  }
  worker.Reset()
```

```
}
clock.LogfReset("received kmer occurences")
log.Println("number of different kmers:", len(kmerOcc)+len(
    trailKmerOcc))

hft := matc.BuildHuffmanTree(kmerOcc)
var trailHft *matc.HuffmanTree
if kr > 0 {
  trailHft = matc.BuildHuffmanTree(trailKmerOcc)
}
codingTable := hft.CodingTable()
for _, worker := range workers {
  _, err = hft.EncodeBits(worker, k)
  if err != nil {
    return err
  }
  if kr > 0 {
    _, err = trailHft.EncodeBits(worker, kr)
    if err != nil {
      return err
    }
  }
  err = worker.Flush()
  if err != nil {
    return err
  }
}
clock.LogfReset("computed and sent huffman tree")

// open ouput file for writing
// and create buffered bit writer
f, err := os.Create(outFile)
if err != nil {
  return err
}
defer f.Close()
bw := bit.NewBufWriter(f)
```

```
defer bw.Flush()

// write headers to file
err = bw.WriteBits(uint64(n), 16)
if err != nil {
  return err
}
err = bw.WriteBits(uint64(k), 8)
if err != nil {
  return err
}

// write huffman trees
_, err = hft.EncodeBits(bw, k)
if err != nil {
  return err
}
if kr > 0 {
  _, err = trailHft.EncodeBits(bw, kr)
  if err != nil {
    return err
  }
}
hft = nil
trailHft = nil

// write multiple attribute trees
_, err = matc.EncodeVarWidthUint64(bw, uint64(numTopKmers-1))
if err != nil {
  return err
}
matWritten := 0
bitbuf := bit.NewBuffer()
for _, worker := range workers {
  for {
    err = kmer.DecodeBits(worker, k)
    if err != nil {
```

```
    return err
  }
  if kmer == 0 {
    break
  }

  numTopKmers--
  if numTopKmers < 0 {
    panic("numTopKmers < 0")
  }
  _, err = codingTable.EncodeKmer(bw, kmer)
  if err != nil {
    return err
  }

  m, err := worker.ReadBits(64)
  if err != nil {
    return err
  }
  // read mat from connection
  err = bitbuf.ReadFrom(worker, int(m))
  if err != nil {
    return err
  }
  err = bitbuf.Flush()
  if err != nil {
    return err
  }

  // write mat to disk
  err = bitbuf.WriteTo(bw, int(m))
  if err != nil {
    return err
  }

  bitbuf.Reset()
```

```
      matWritten++
      if matWritten%100000 == 0 {
        clock.Logf("received %vk multiple attribute trees",
            matWritten/1000)
      }
    }
    clock.Logf("received multiple attribute trees from worker %v",
        worker.conn.RemoteAddr())
  }
  bitbuf = nil
  clock.LogfReset("received multiple attribute trees and wrote them
      to disk")

  return nil
}

func recvKmerOcc(r bit.Reader, k int, kmerOcc map[matc.Kmer]int)
   error {
  var kmer matc.Kmer
  for {
    err := kmer.DecodeBits(r, k)
    if err != nil {
      return err
    }
    if kmer == 0 {
      break
    }
    occ, err := r.ReadBits(32)
    if err != nil {
      return err
    }
    kmerOcc[kmer] += int(occ)
  }
  return nil
}

func minInt(values []int) (i, n int) {
  for j, m := range values {
    if j == 0 || m < n {
      i = j
      n = m
    }
  }
  return
}
```

A.2.5 run_worker.go

```
package main

import (
  "errors"
  "io"
  "log"
  "net"
  "sync"

  "github.com/dselim/cigarcoil/matc"
  "github.com/dselim/go-bit"
)

var (
  running    bool
  runningMux sync.Mutex
)

func isRunning() bool {
  runningMux.Lock()
  defer runningMux.Unlock()
  return running
}

func setRunning(value bool) {
  runningMux.Lock()
  defer runningMux.Unlock()
  running = value
}

// runWorker launches a TCP server
// and listens for the connection of
// a distributed MATC master
func runWorker() error {
  l, err := net.Listen("tcp", listenAddr)
  if err != nil {
    return err
  }
  log.Println("listening at", listenAddr)
  for {
    conn, err := l.Accept()
    if err != nil {
      return err
    }
    log.Println("received connection from", conn.RemoteAddr())
    if isRunning() {
      log.Println("already running, terminating connection")
      conn.Write([]byte{0})
      conn.Close()
      continue
    }
    setRunning(true)
    conn.Write([]byte{1})
    go func(conn net.Conn) {
      defer setRunning(false)
      err := handleConnection(conn)
      if err != nil {
        if errors.Is(err, io.EOF) {
          log.Println("EOF while running:", err)
        } else {
          log.Println("error while running:", err)
        }
      } else {
        log.Println("finished running")
      }
    }(conn)
    conn = nil
  }
}

func handleConnection(conn net.Conn) error {
  clock := NewClock()
  log.Println("running for", conn.RemoteAddr())
```

```
rw := bit.NewBufReadWriter(bit.NewBufReader(conn), bit.
    NewBufWriter(conn))

// RECEIVE HEADERS
v, err := rw.ReadBits(16)
if err != nil {
  return err
}
n := int(v)
v, err = rw.ReadBits(8)
if err != nil {
  return err
}
k := int(v)
rw.Reset()
clock.LogfReset("received parameters")

nk := n / k
kr := n % k

log.Println("n", n, "k", k, "nk", nk, "kr", kr)

// RECEIVE KMERS
kmerOcc := make(map[matc.Kmer]int)
var trailKmerOcc map[matc.Kmer]int
if kr > 0 {
  trailKmerOcc = make(map[matc.Kmer]int)
}
mats := make(map[matc.Kmer]*matc.MAT)
kmers := make([]matc.Kmer, nk)
if kr > 0 {
  kmers = append(kmers, 0)
}
for {
  err = kmers[0].DecodeBits(rw, k)
  if err != nil {
    return err
  }
  if kmers[0] == 0 {
    break
  }
  for i := 1; i < nk; i++ {
    err = kmers[i].DecodeBits(rw, k)
    if err != nil {
      return err
    }
  }
  if kr > 0 {
    err = kmers[nk].DecodeBits(rw, kr)
    if err != nil {
      return err
    }
  }
  kmerOcc[kmers[0]]++
  t, ok := mats[kmers[0]]
  if !ok {
    t = new(matc.MAT)
    mats[kmers[0]] = t
  }
  t.AddLeaf(kmers[1:], kmerOcc, trailKmerOcc)
}
rw.Reset()
kmers = nil
clock.LogfReset("received all kmers")

err = sendKmerOcc(rw, k, kmerOcc)
if err != nil {
  return err
}
if kr > 0 {
  err = sendKmerOcc(rw, kr, trailKmerOcc)
  if err != nil {
    return err
  }
}
err = rw.Flush()
if err != nil {
  return err
}
kmerOcc = nil
trailKmerOcc = nil
clock.LogfReset("sent kmer occurences")

hft := new(matc.HuffmanTree)
err = hft.DecodeBits(rw, k)
if err != nil {
  return err
}
var trailHft *matc.HuffmanTree
if kr > 0 {
  trailHft = new(matc.HuffmanTree)
  err = trailHft.DecodeBits(rw, kr)
  if err != nil {
    return err
  }
}
rw.Reset()
clock.LogfReset("received huffman tree(s)")

codingTable := hft.CodingTable()
var trailCodingTable matc.CodingTable
if kr > 0 {
  trailCodingTable = trailHft.CodingTable()
}
hft = nil
trailHft = nil
clock.LogfReset("computed coding table(s)")

bitbuf := bit.NewBuffer()
for kmer, mat := range mats {
  delete(mats, kmer)

  _, err = kmer.EncodeBits(rw, k)
  if err != nil {
    return err
  }

  // write mat in bit buffer
  nkmers, nchildren, err := mat.EncodeBits(bitbuf, codingTable,
      trailCodingTable)
  if err != nil {
    return err
  }
  m := nkmers + nchildren
  err = bitbuf.Flush()
  if err != nil {
    return err
  }

  // write bit buffer length and content
  err = rw.WriteBits(uint64(m), 64)
  if err != nil {
    return err
  }
  err = bitbuf.WriteTo(rw.BitWriter, m)
  if err != nil {
    return err
  }
  bitbuf.Reset()

  mat = nil
}
mats = nil
bitbuf = nil
var kmer matc.Kmer
_, err = kmer.EncodeBits(rw, k)
if err != nil {
  return err
}
err = rw.Flush()
if err != nil {
  return err
}
rw = nil
clock.LogfReset("written multiple attribute trees")

return nil
}


func sendKmerOcc(w bit.Writer, k int, kmerOcc map[matc.Kmer]int)
 error {
var err error
for kmer, occ := range kmerOcc {
  _, err = kmer.EncodeBits(w, k)
  if err != nil {
    return err
  }
  if int(uint32(occ)) != occ {
    panic("kmer occurences value cannot be converted to uint32")
  }
  err = w.WriteBits(uint64(occ), 32)
  if err != nil {
    return err
  }
}
var kmer matc.Kmer
_, err = kmer.EncodeBits(w, k)
if err != nil {
  return err
}
return nil
}
```