UNIVERSITÀ DEGLI STUDI DI SALERNO

UNIVERSITÀ DEGLI STUDI DI SALERNO
Dipartimento di Farmacia

Dottorato di ricerca
in Biologia dei Sistemi
Ciclo XIII — Anno di discussione 2015

Coordinatore: Chiar.mo Prof. *Antonietta Leone*

Compression and indexing of genomic data with confidentiality protection

settore scientifico disciplinare di afferenza: INF/01

Dottorando

Dott. *Ferdinando Montecuollo*

Tutore

Chiar.mo Prof. *Roberto Tagliaferri*

Co-tutore

Chiar.mo Dr. *Giovanni Schmid*

Index

Abstract

The problem of data compression having specific security properties in order to guarantee user's privacy is a living matter. On the other hand, high-throughput systems in genomics (e.g. the so-called Next Generation Sequencers) generate massive amounts of genetic data at affordable costs.

As a consequence, huge DBMSs integrating many types of genomic information, clinical data and other (personal, environmental, historical, etc.) information types are on the way. This will allow for an unprecedented capability of doing large-scale, comprehensive and in-depth analysis of human beings and diseases; however, it will also constitute a formidable threat to user's privacy.

Whilst the confidential storage of clinical data can be done with well-known methods in the field of relational databases, it is not the same for genomic data; so the main goal of my research work was the design of new compressed indexing schemas for the management of genomic data with confidentiality protection.

For the effective processing of a huge amount of such data, a key point will be the possibility of doing high speed search operations in secondary storage, directly operating on the data in compressed and encrypted form; therefore, I spent a big effort to obtain algorithms and data structures enabling pattern search operations on compressed and encrypted data in secondary storage, so that there is no need to preload data in main memory before starting that operations.

Riassunto

La progettazione di metodi integrati di compressione e cifratura che garantiscano la riservatezza dei dati riducendone al contempo la dimensione è un argomento di ricerca molto attuale in Informatica.

D'altro canto, l'avvento delle nuove piattaforme di sequenziamento NGS e, più in generale, delle tecnologie cosiddette high-throughput ha consentito negli ultimi anni di ottenere velocemente, e a costi molto ridotti rispetto al passato, notevoli quantità di dati genomici.

Di conseguenza è sempre più sentita l'esigenza di costruire grandi database che integrino le informazioni genomiche con dati personali e clinici: database di questo genere permetterebbero, infatti, di ottenere risultati migliori nella cura di malattie multifattoriali e una comprensione più approfondita dei fenomeni biologici che avvengono nel nostro organismo; essi potrebbero costituire, tuttavia, un notevole problema per la privacy dei pazienti.

Mentre la confidenzialità dei dati clinici può essere ottenuta con tecniche di cifratura standard, comunemente applicate nel campo dei database relazionali, lo stesso non può dirsi per i dati genomici; il principale obiettivo del mio lavoro di ricerca è stato, quindi, la progettazione di nuovi schemi integrati di compressione e cifratura che consentissero di gestire dati genomici preservandone la riservatezza.

Affinchè tali sistemi fossero sfruttabili in applicazioni reali, ho profuso grande impegno nella progettazione di algoritmi e strutture-dati che consentissero di effettuare ricerche efficienti direttamente sui dati in forma compressa e cifrata, caricando di volta in volta in memoria RAM solo il minimo set di informazioni necessario ad eseguire ogni specifica operazione di ricerca.

Introduction

NGS technologies and Genomic sequences

DNA double helix structure was discovered by Watson and Crick in 1953: DNA is an organic polymer structured as a extremely long chain of monomers called nucleotides (deoxyribonucleotides). For example, human DNA contains approximately three billions nucleotides.

Each nucleotide contains a nitrogenous base that represents its information content and can be one of the following: adenine (A), cytosine (C), guanine (G) and thymine (T).

DNA sequencing methods allows to determine the exact sequence of an individual genome nucleotides.

The first sequence of 24bp was published in 1973, but only in 1977 the English biochemist Frederick Sanger developed and published the revolutionary *Sanger method*: thanks to his work and its enhancements, it was possibile in 2000 to complete the sequencing of the whole human genome.

Sanger's was the most used sequencing method for more than 25 years, but it suffered some problems: for example, it was very difficult to automate the samples preparation process and to handle in parallel a big number of samples.

In order to address these problems and to have faster and cheaper sequencing platforms, in 2005 first Next generation sequencing platforms, Roche 454 and Solexa Illumina, appeared on the market.

Their massive parallel sequencing methods allow now to get genomic data at much higher speed and lower cost than in the past, enabling large scale genome sequencing projects, like the famous *1000 Genomes Project*[14].

This is giving rise to a considerable amount of NGS-type sequences (short reads), for which traditional storage and processing approaches have proven inadequate: e.g., to assemble these reads in order to obtain the overall DNA sequence of the sampled individual, new alignments algorithms and tool have been developed.

Independently from the sampling method, the final product of a sequencing process is indeed the sampled individual DNA genomic sequence or a part of it, and this sequence must be stored anywhere in some format.

The most used data formats for genomic sequences are:

- *FASTA*, which is a text format for representing nucleotide or peptide sequences, encoding each sequence element with a single ASCII character;
- *FASTQ*, which is another text format for storing both a sequence and its corresponding quality scores, also encoding sequence letters and quality scores with single ASCII characters.

Both are very expensive in terms of disk space; moreover they do not support fast search and alignments versus their content, because they do not intrinsically provide any type of indexing.

In order to overcome these issues compressed full-text indexes, like FM-indexes[15], are often used in Bioinformatics to allow fast searches on compressed genomic data; they constitute the basis of many alignment tools, like *Bowtie2*.

Hashing and other indexing methods have been used instead to implement *BLAST* and other alignment and search tools which compare an input sequence with all those stored in big databases, in order to find the most similar ones.

Genomic databases and confidentiality protection

Tipically genomic data are organized in structured and searchable collections of information, named databases. There are two kinds of genomic sequence databases: public and private databases. The first ones are accessible by everyone in the world at no charge: some examples are *DDBJ*, *EMBL* and *GenBank*; the second ones contain collections of data sequenced by private companies, which let them accessible only when paying an accession fee: therefore public databases are mostly used for academical research, while private ones are used by pharmaceutical companies.

Both the above mentioned categories of databases have been designed to improve scientific research in genomics: sequence data are stored in clear format and the only applied privacy protection rule consists in anonymizing data, avoiding to associate them with personal data like surname, name and tax code. This type of anonymization well suites to big worldwide databases designed only for research purposes: not only they contain data sequenced from individuals coming from all world regions, but there is no need to access the related personal data.

My research work instead addresses a very different type of multi-user databases, designed to be used both in clinical and in research contexts: they must contain collections of genomic sequences from several individuals, potentially coming from a circumscribed region; they must implement moreover a security model enabling their users to work only on the set of genomic information to which they have been granted access.

This need arises from a simple observation: genomic sequences are indeed classifiable as sensitive data and so confidentiality and privacy concerns must be taken into account.

In fact, not only the genomic information alone could afford to trace the identity of the individual, but it could be used for discriminatory and not legitimate purposes. For example, an insurance company may deny the stipulation of a health insurance agreement depending on the predisposition to develop certain types of diseases, or a company may not hire a worker inferring from his DNA an aggressiveness higher than the average.

So, genomic data should be kept away from prying eyes, both using methods that guarantee its privacy and establishing new appropriate laws.

In recent years, new regulations were introduced at European level, which govern the access to databases enabling the so-called *Forensic DNA profiling*: this is a method used by forensic scientists and police to identify individuals on the basis of a *DNA profile*, some mutations of their DNA, which are extremely unlikely to happen together.

Genetic profiles are stored into data banks designed to compare a crime scene DNA record with DNA records contained in the database: American “Combined DNA Index System” (CODIS) is an example of such a database and of the related accession rules.

European Union States have subscribed in 2005 the Prum Convention: it "provides for the automated exchange of DNA, fingerprints and vehicle registration data, as well as for other forms of police cooperation between the 27 EU States. Access to DNA profiles and fingerprints held in national databases is granted on a hit/no-hit basis, which means that DNA profiles or fingerprints found at a crime scene in one EU State can be compared with profiles held in the databases of other EU States."

Most of the above described conventions and laws only regulate the storage and usage of DNA profiles in parentage testing and criminal investigation. However, DNA profiling should not be confused with whole genome sequencing, which has as its objectives the determination, and possibly the long-term storage, of the whole genomic sequence of an individual.

The Italian Data Protection Authority has recently released a "General authorization for genetic data processing", establishing a set of rules which regulate the several processing purposes, the security measures to adopt and the informed consent procedures to follow so that an individual can know and limit the usage of his/her genetic data to a given set of purposes.

In conclusion, currently new regulations are appearing with the aim of disciplining the treatment of genetic data, but there is a lack of technical standards adequate to enable confidential storage and fast accession to it.

This is why I started my research project: my aim was to model new indexed, compressed and encrypted data structures:

- enabling the long-term storage and retrieval of individuals genomic sequences, and not only of the limited set of genetic information needed for his/her identification;
- allowing fast pattern searches directly on encrypted and compressed data;
- easily integrable with standard relational databases containing clinical information.

Thesis outline

The aim of this thesis is to present two new compressed and encrypted self-index models for genomic sequences: indexes of this type do not currently exist in literature.

The thesis is organized in three chapters.

Chapter 1 gives some basic details on Burrows and Wheeler Transform (BWT)[9] and FM-Index[15], from whose analysis I started my research activities; furthermore, at the end of this chapter I propose my first contribution: a new fast method to efficiently compute the BWT of genomic sequences.

Chapter 2 presents my first encrypted self-index model, named **EFM-index**: it combines an approach known in literature as *Scrambled BWT*[25] with a structure similar to that of FM-index, together with a block encryption method based on the *Salsa20*[6] stream cipher. It has optimal search performance and exhibits optimal compression ratios on a single sequence and on collections of lowly similar sequences; however, it cannot fully exploit the inter-sequence redundancy that exists in collections of highly similar sequences.

Chapter 3 presents my second encrypted self-index model, named **ER-index**: when used on collections of highly similar sequences, it allows to obtain compression ratios which are an order of magnitude smaller than EFM-index, maintaining optimal search performance. Moreover, its *multi-user and multiple-keys encryption model* permits to store genomic sequences of different individuals with distinct encryption keys within the same index: this allows each user to perform search operations only on the sequences to which he/she was granted access.

Durign my PhD studies, which lasted three years, I developed, tested and incrementally refined several software prototypes of the above mentioned indexes. The experimental results obtained by the two last prototypes on several collections of genomic sequences show that both the proposed index models could be successfully used in real-world applications.

Chapter 1

Compression and indexing of genomic data

Introduction

To date, great efforts have been made to obtain compress and indexed representations of genomic sequences. In particular, methods based on the *Burrows Wheeler Transform (BWT)* [9] have been introduced that obtain excellent results in terms of compression ratio, search efficiency and sequence alignment: for example, *Bzip2* is a compressor that gets very good space-efficiency on most files thanks to the BWT followed by a move-to-front transform and Huffman coding [35], whereas *Bowtie* is a short read aligner that is capable of aligning short reads to the human genome at a rate of over 25 million 35-bp reads per hour [28].

In this chapter I explain some BWT details needed to understand the methods and algorithms that I present in the following chapters; afterwards I speak about the FM-index, from which I started my research activities, and finally I present a my first contribution: an original method to compute efficiently the BWT of genomic sequences, I called *FastGenomicBWT*.

1.1 Burrows and Wheeler transform

The Burrows and Wheeler Transform (BWT) [9] is based on the idea that a permutation of the text given in input may give rise to a more easily compressible output. Let T be an n-length sequence of characters over an alphabet Σ. Then $BWT(T)$ is a permutation of T with respect to the lexicographical order of Σ, which is obtained as follows (see Fig. 1.1):

1. A special symbol \$ is placed after text T, with the convention that \$ precedes lexicographically all the other symbols in Σ;
2. The resulting text is shifted by one place on the right for $n+1$ times, so to obtain a $n+1$-order *rotations matrix* M, whose adjacent rows differ by a single circular shift;
3. The rows of M are sorted with respect to the *canonical (lexicographical) ordering* induced by Σ, giving rise to the sorted rotations matrix M_{Σ};

4. $BWT(T) = L$, where L is the text given by reading the last column of M_Σ from the top.

$BWT(T)$ is generally much more compressible than T because adjacent rows in M_Σ start with common corresponding substrings (*contexts*), which in some cases can be quite long, and characters preceding the same contexts are more probably equals. The BWT can be compressed with a run-length coder and/or a move-to-front coder [39].

To restore T from $BWT(T) = L$, it is needed to know where T starts and which letters follow which others. The crucial observation in such respect is that, if F is the first column of M_Σ, then for each $1 \leq i \leq n+1$ the element $F(i)$ follows $L(i)$ in T. Thus, the *inversion* of $BWT(T)$ can be easily accomplished by restoring T backwards as follows (see Fig. 1.1):

1. The L column is lexicographically ordered, giving rise to F;
2. Let $M' = [F, L]$ be the $2 \times n + 1$ matrix having F and L as first and last column, respectively;
3. The last character of T is the first of L, i.e. $T(n) = L(1)$, since $L(1)$ precedes \$ and \$ is the first character in F;
4. $T(j) = L(i)$ for $j = n-1, \ldots, 1$, where i is the minimum row index in M' such that $F(i) = T(j+1)$.

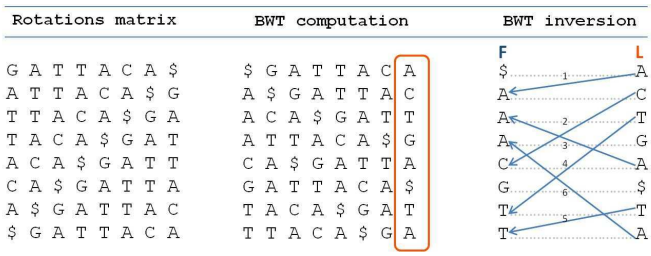


Figure 1.1: An example of BWT computation and its inversion for the text $T =$ "$GATTACA$"

1.2 FM-index

The Fast index in Minute Space (FM-index) [15]) is an example of *self-index*, namely a data-structure which replaces a given text compressing it and allowing in addition a fast search of its substrings.

The FM-index of a text T is composed of a binary sequence Z, which results from the compression of $L = BWT(T)$, plus a set of auxiliary data structures: these data structures describe the subdivision of Z into data blocks of variable length, each block being related to a single l-length block of L. For improved performance, the n/l data blocks are clustered in n/l^2 super-blocks. The length $l > 0$ is a parameter affecting compression ratio and search speed in opposite way: decreasing l usually results in a faster searching but in a bigger size index. Indeed, blocks and super-blocks increase in number as l decreases, and carry in auxiliary information which speeds up searches but results in a bigger index.

The compression is achieved by splitting $L = BWT(T)$ in l-length blocks and by applying to each of them a set of coders in the following order: a *move-to-front* (MTF) transform, a *zero-run-length encoder* (RLE0), and a *prefix-free encoding*.
The MTF transform [33, 4] is a simple way of generating a code based on the probability of occurrence of symbols in the input strings. First, an ordered sequence of symbols (*MTF table*) is constructed by following the lexicographical ordering of Σ, in such way that each symbol is mapped to a unique integer in $(0, |\Sigma| - 1)$. Then, each symbol of the input sequence is coded as the corresponding integer in the MTF table, after which the table is updated with such a symbol moved to the top (position 0). The MTF output results in a string of integers where runs of same characters are mapped to runs of zeros, with the more frequently occurring characters appearing near the top of the list.
The RLE0 simply turns each run of zeros in a single zero followed by a count (in binary) of the number of times it occurs, and leaving the encodings of the other numbers.
Lastly, the PFE encodes the sequence of binary digits produced by RLE0 using a variable-length *prefix code* [39].

Apart from the binary sequence Z obtained by concatenating all the output blocks given by the previous compression process, the FM-index consists of the following auxiliary data structures:

- an array containing, for each character, the number of occurrences of all the characters preceding it;
- a super-blocks array, whose j-th element stores:
  - the sum of the sizes of all compressed blocks until the end of super-block j;
  - a *rank table* containing, for each $c \in \Sigma$, the number of occurrences of c until the beginning of super-block j;
- an array whose i-th element corresponds to the i-th block of $L = BWT(T)$ and stores:
  - the sum of the sizes of all blocks up to the i-th block (included), starting from the super-block to which it belongs;
  - the MTF table status before applying the MTF to block i;
  - a *rank table* containing, for each $c \in \Sigma$, the number of occurrences of c up to the beginning of block i, starting from the super-block to which it belongs.

1.3 Fast Genomic BWT: a new multi-threading BWT computation algorithm

In order to perform a fast BWT computation of big genomic sequences, I designed an algorithm to parallelize it on modern multi-core and hyper-threading CPUs.

My algorithm origins from a simple observation: given a genomic sequence s, which is a string on the alphabet Σ, its rotations can be distributed uniformly in contiguous symbols ranges of an extended alphabet Σ^k, on the basis of each rotation first k characters. This happens in a useful way for our purposes yet for low k values (i.e. for $k >= 4$).

In order to obtain an additional performance increment, I reserved a special treatment to *long repetitions* of the same character, which make the ordering very difficult: the

ranges containing long repetitions are split in several subranges, which in turn are separately ordered as I show hereafter. An example of such repetitions are the very long N character regions often occurring in genomic reference sequences.

Shortly, the overall computation algorithm evolves as follows:

1. it accepts in input a sequence s, an alphabet extension degree k and the maximum number nt of threads executable in parallel on the running machine.
2. it builds the extended alphabet Σ^k, where Σ consists only of the symbols effectively appearing in s.
3. it distributes the rotations in a sufficiently high number of ranges of the extended alphabet and retrieves long repetitions. This is a multi-thread step: the input text is divided into several regions of consecutive characters and each region is analyzed by a separate thread; the final result of this step is an array R of ranges, each of which contains a set of rotations.
4. if necessary, it joins the repetitions found by the several threads of the step 2, because a repetition positioned at the end of a region could continue from the first character of the next region (single-thread step).
5. it assembles the not empty ranges in nt groups, splitting the array R into nt sub-arrays so that the number of rotations in the several sub-arrays is approximately the same (single-thread step, performed by a greedy sub-algorithm that I named *array splitting*, which permits to balance the workload of the sorting threads).
6. it sorts in parallel the rotations belonging to each group of ranges (multi-thread step).
7. finally it merges the sort results and computes BWT (single-thread step).

The sorting of each range/subrange is performed through the *multi-key quick sort* algorithm[3]; I chose this algorithm for two reasons: it is very fast and moreover its main building block compares strings starting from a given position ($depth$), which was very important in designing my strategy to treat long repetitions.

After giving an overall view of my method, I am going to explain some useful details to reproduce it, omitting the detailed description of the steps that do not require further study; finally, I will show some interesting experimental results.

1.3.1 Method details

The algorithm 1 shows the array splitting details: it splits an array A into a set sn adjacent subarrays, so that the sum of the elements in each sub-array is almost the same. In other words, if the array elements are seen as weights, this problem is equivalent to split the array in sub-arrays with almost the same total weight.

The algorithm tries to minimize, with a greedy approach, an objective function defined as the sum of square quadratic errors between each sub-arrays weigths and an optimal weight: this optimal weight corresponds to an ideal assignment of splitters in which all the subarrays have exactly the same total weight and therefore it can be computed summing all the elements of A and dividing the obtained sum by sn.

The computation stops when the number of steps exceeds $maxSteps$ or when the percentage gain obtained during last step falls below $minPercentageGain$.

Moreover, also the method used to sort each range of rotations $r \in R$ needs to be described more in detail. Indeed the sorting algorithm is trivial if r does not contain long-repetitions rotations, as it simply consists in a single execution of *multi-key quick sort* with $depth = 0$. On the contrary, if r contains long-repetitions rotations, i.e. rotations whose initial part falls within a long repetition of the same character, the algorithm splits r in:

1. a subrange containing all the rotations whose starting character precedes the repeated character;
2. a set of subranges containing only rotations starting with the repeated character, obtained as follows: if mrl is the maximum length of that character repetitions and n_{sub} is the number of desired subranges, the interval $]0, mrl]$ can be divided into n_{sub} subintervals of length mrl/n_{sub}; each of the n_{sub} subranges will correspond biunivocally to one of these subintervals and will contain only rotations starting with a number of repeated characters falling in that subinterval.

   This approach has two advantages: not only the several subranges can be ordered in parallel, but the long repetition subrange corresponding to a subinterval $]l, r]$ can be ordered avoiding to compare the first l symbols, in that they certainly match.
3. a subrange containing all the rotations whose starting character follows the repeated character.

An example will better clarify the method. Suppose that $mrl = 12$, N is the repeated character and $n_{sub} = 3$. The interval $]0, 12]$ will be divided into 3 subintervals: $]0, 4],]5, 8],]9, 12]$. Consider some rotations starting with the long repetition character N:

$$\begin{cases} N & N & N & A & C & G & T \dots \\ N & A & C & G & C & G & T \dots \\ N & N & T & C & C & A & T \dots \\ N & N & N & N & C & A & A \dots \end{cases}$$
$$\begin{cases} N & N & N & N & N & C & A \dots \\ N & N & N & N & N & T & A \dots \\ N & N & N & N & N & N & A \dots \end{cases}$$

The first four rotations will be assigned to a first subrange, corresponding to the subinterval $]l_0, r_0] =]0, 4]$: they can be sorted starting the comparisons from the position $l_0 + 1 = 1$, as their first character is certainly a N; so they can be sorted running the *multi-key quick sort* algorithm with $depth = 1$.

Similarly, the last three rotations will be assigned to a second subrange, corresponding to the subinterval $]l_1, r_1] =]4, 8]$: they can be sorted starting the comparisons from the position $l_1 + 1 = 5$, as their first 5 characters are all equal to N; so they can be sorted running the *multi-key quick sort* algorithm with $depth = 5$.

Note that the above described method ensures that, in order to compare two sequences, the sorting algorithm will do at most $mrl/n_{sub} = 12/3 = 4$ steps before exiting from the repeated character region, and it will find in a few other steps two different characters. The aforementioned depths have been computed thinking of a 0-based (C-like) string representation, in which the elements indexes start from 0.

Another key point of my method concerns the merging operation: once all ranges and subranges have been separately ordered, their rotations must be organized according a unique global ordering, before starting the BWT computation. Again, this is trivial for ranges, as they correspond to consecutive intervals of the Σ^k alphabet: therefore their rotations can

be naturally organized so that the second range suffixes follow the first one suffixes, the third range suffixes follow the second one suffixes, and so on.

A not trivial approach must instead be used when merging the long repetitions subranges. Consider the above example rotations after the sorting phase:

$$\begin{cases} N & A & C & G & C & G & T\ldots \\ N & N & N & A & C & G & T\ldots \\ N & N & N & N & C & A & A\ldots \\ \hline N & N & T & C & C & A & T\ldots \end{cases} \quad \begin{cases} N & N & N & N & N & C & A\ldots \\ N & N & N & N & N & N & A\ldots \\ N & N & N & N & N & T & A\ldots \end{cases}$$

The horizontal line marks a split point in the ordered list of rotations belonging to the first subrange; it splits the subrange rotations in two parts:

- a right side containing all rotations whose first non-repeated character follows the repeated character N in lexicographical ordering;
- a left side containing the remaining rotations.

Ultimately, the above mentioned split point is the *insertion point* of the following subranges rotations into the current subrange rotations ordered list. When reconstructing the global ordering, the left side rotations must precede the second subrange rotations, which in turn precede the right side rotations; again, the left side suffixes of second subrange must precede the third subrange rotations, which in turn precede the second subrange's right side rotations, and so on.

Therefore I used a recursive approach, reported in the algorithm 2.

Algorithm 1 SplitArray: splits an array A into a set of consecutive sub-arrays so that the sum of the elements in each sub-array is almost the same

INPUT:
A: array of weights
sn: number of desired sub-arrays
$maxSteps$: maximum number of steps
$minPercentageGain$: minimum objective function gain in a step
OUTPUT:
$splitters$: array of splitters

```
1: ▷ Returns the sum of square quadratic errors between the sub-arrays weigths and the optimal weight
2: function ComputeObjectiveFunction(A,sn,splitters,optimalWeight)
3:     result ← 0
4:     for sai ← 0 to sn − 1 do
5:         subArrayWeight ← computeSubarrayWeight(A, splitters, sai);
6:         result ← result + (subArrayWeight − optimalWeight)^2;
7:     end for
8:     return result;
9: end function

10: ▷ Returns a sub-array weight, i.e. the sum of all the weights contained in a subarray
11: function ComputeSubarrayWeight(A,sn,splitters,subArrayIndex)
12:     if subArrayIndex = 0 then
13:         startPosition = 0;
14:     else
15:         startPosition = splitters[interval − 1];
16:     end if
17:     if subArrayIndex = sn − 1 then
18:         endPosition = n − 1;
19:     else
20:         endPosition = splitters[interval] − 1;
21:     end if
22:     sum ← 0;
23:     for i ← startPosition to endPosition do
24:         sum ← sum + A[i];
25:     end for
26:     return sum;
27: end function
```

Algorithm 1 SplitArray (continued)

```
28: n ← length(A);
29: ▷ Sums all weights to compute the total weight
30: sumOfWeights ← 0;
31: for i = 0 to n − 1 do
32:     sumOfWeights ← sumOfWeights + A[i];
33: end for
34: splitters=[];                                                    ▷ Trial splitters positions
35: ▷ Do an initial assignment of splitters, so that they are equally spaced
36: intervalLength ← n/sn;
37: for j = 0 to sn − 1 do
38:     splitters[j] ← round((j + 1) ∗ intervalLength);
39: end for
40: maxGainSplitters ← []; ▷ Assignment of splitters corresponding to the best value of the objective function
41: optimalWeight ← n;
42: steps ← 0;
43: gain ← 1;
44: ▷ Try to move the splitter right so that at least one element remains in the next interval
45: while steps < maxSteps AND gain ≥ minPercentageGain do
46:     objectiveFunctionInitialValue ← computeObjectiveFunction(A, splitters,
                                                                 optimalWeight);
47:     maxGainSplitters ← splitters;
48:     for j = 0 to sn − 1 do                                          ▷ for each splitter
49:         ▷ Try to move the splitter left so that at least one element remains in the previous interval
50:         if j = 0 then
51:             leftmostPosition ← 1;
52:         else
53:             leftmostPosition ← splitters[j − 1] + 1;
54:         end if
55:         for trialPosition = splitters[j] − 1 downto leftMostPosition do
56:             trialSplitters ← splitters;
57:             trialSplitters[j] ← trialPosition;
58:             objectiveFunctionTrialValue ← computeObjectiveFunction(A, trialSplitters(A,
                                                                 splitters, optimalWeight));
59:             trialGain ← objectiveFunctionInitialValue − objectiveFunctionTrialValue;
60:             if trialGain > maxGain then
61:                 maxGainSplitters ← trialSplitters;
62:                 maxGain ← trialGain;
63:             end if
64:         end for
65:         if j = sn − 1 then
66:             rightmostPosition ← n − 1;
67:         else
68:             rightmostPosition ← splitters[j + 1] − 1;
69:         end if
```

Algorithm 1 SplitArray (continued)

```
70:         for trialPosition = splitters[j] + 1 to rightMostPosition do
71:             trialSplitters ← splitters;
72:             trialSplitters[j] ← trialPosition;
73:             objectiveFunctionTrialValue ← computeObjectiveFunction(A, trialSplitters,
                                                                       optimalWeight);
74:             trialGain ← objectiveFunctionInitialValue − objectiveFunctionTrialValue;
75:             if trialGain > maxGain then
76:                 maxGainSplitters ← trialSplitters;
77:                 maxGain ← trialGain;
78:             end if
79:         end for
80:     end for
81:     if maxGain > 0 then splitters ← maxGainSplitters; gain = maxGain;
82:     end if
83: end while
```

Algorithm 2 Permits to process long repetitions subranges, scanning their rotations according to the global ordering needed for BWT computation

```
INPUT:
    sri: subrange index
GLOBAL VARIABLES:
    longRepetitionsSubRanges: array containing all the long repetitions subranges
 1: procedure ProcessLongRepetitionsSubRange(sri)
 2:     subRange ← longRepetitionsSubRanges[sri];
 3:     ▷ Compute the number of rotations belonging to this subrange
 4:     n = size(subRange.rotations);
 5:     ▷ Compute the insertion point: it is the position of the first rotation whose first non-repeated character
 6:     ▷ follows the repeated character N in lexicographical ordering
 7:     insertionPoint ← computeInsertionPoint(subRange);
 8:     for i ← 0 to insertionPoint − 1 do
 9:         ▷ Process rotation i, taking its last character to build the BWT
10:         processRotation(i);
11:     end for
12:     ProcessLongRepetitionsSubRange(sri + 1);
13:     for i ← insertionPoint to n − 1 do
14:         ▷ Process rotation i, taking its last character to build the BWT
15:         processRotation(i);
16:     end for
17: end procedure
```

1.3.2 Experimental results

In order to test the *FastGenomicBWT* algorithm, I implemented a prototype in C++ language, comparing its execution times with those obtained through the *libdivsufsort* library, which is considered the state of the art in matter of BWT computation.

Figure 1.2 compares the BWT computation times of FastGenomicBwt and libdivsufsort on three sequences very different in length:

- the reference sequence for E. coli genome (4638690 bp);
- the reference sequence for human chromosome 20, from hs37d5 assembly (63025520 bp);
- the reference sequence for human chromosome 1, from hs37d5 assembly (249250621 bp).

The experiments were conducted on a small-sized server equipped with two Intel(R) Xeon(R) CPU X5680 3.33GHz 6-core hyper-threading processors.

The experimental results show that already on a machine with a low actual parallelism (the machine in question can only run 24 threads in parallel) our algorithm is significantly faster than libdivsufsort; the speed-up could be increased running *FastGenomicBWT* on machines with an higher number of cores.

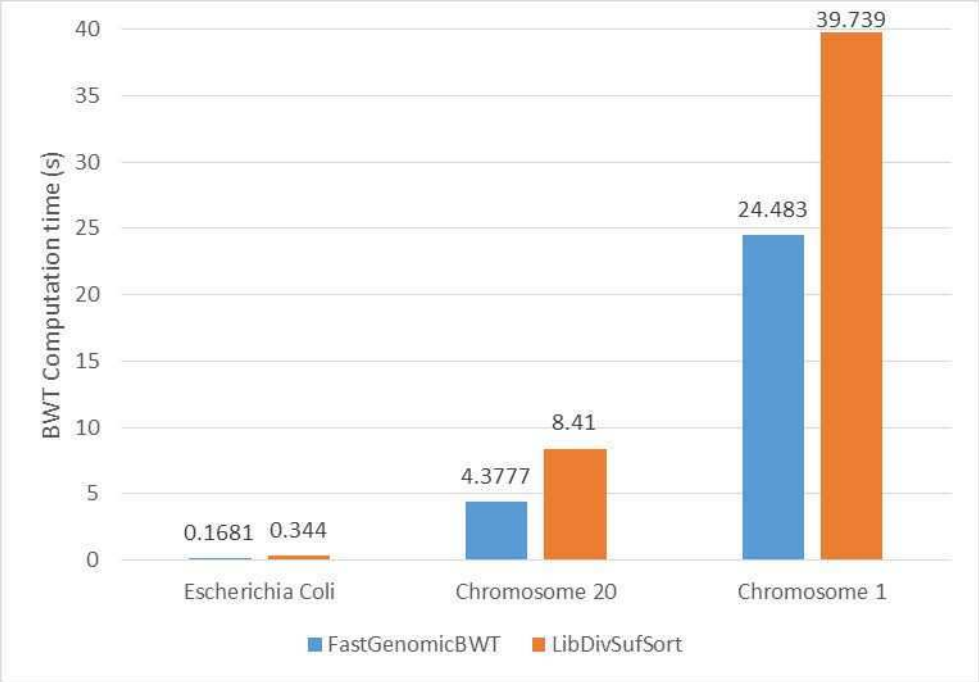


Figure 1.2: Comparison between FastGenomicBWT and libdivsufsort

Chapter 2

EFM-index

Introduction

The availability of low-cost digitized genome is a great opportunity for Life sciences: it will allow to conduct in silico more and more genomic applications and tests, enabling a real personalized medicine. However, such uses require protection from disclosure of digitized genome, whose misuse could have repercussions not only on the life of individuals to whom it belongs, but also on those of their relatives.

State-of-the-art techniques for performing efficiently various operations on compressed data do not offer natively any secrecy, and a traditional way to get confidentiality protection is the so called "compress-then-encrypt" paradigm, in which encryption is performed through a dedicated algorithm after data indexing and compression steps have taken place. For example, compress-then-encrypt methods have been documented in the ZIP File Format Specification since version 5.2 [19], and an AES-based standard has been developed for WinZip [41] and are used also in other file archivers (e.g. 7-Zip [32]). However, this approach has the drawback that one must first decrypt the file or the archive in its entirety before to operate on the compressed file. For massive data amounts, as in case of genomic data, this can lead to big downgrades in performance; moreover, it exposes data during operations, which can be an issue if the databases are in outsourcing or in multi-tenants environments (e.g. cloud environments). Therefore, it takes relevance the study of new approaches to the representation and storage of genomic data that:

- achieve high compression rates;
- preserve confidentiality, and;
- make it possible to quickly perform calculations on compressed and encrypted data.

In this chapter I present a new full-text index that allows to perform searches and queries on both encrypted and compressed genomic sequences: like the *Fast index in Minute space (FM) index* [15], this index is structured in data blocks and super-blocks, and compression on data blocks is performed through a pipeline in which a BWT is followed by a *move-to-front (MTF)* transform [33, 4] and a *run-length Encoding* of zero's patterns (RLE0). Achieving lossless compression through such kind of coders is customary in BWT-based compressors [39], but in addition:

- I carefully optimized my index for genomic sequences;

- I interspersed some high speed encryption transformations in the compression pipeline.

Using the BWT-MTF two-stage process to ensure confidentiality of compressed text was first considered in [24, 25]: the idea was to scramble the lexicographical ordering used in the construction of the BWT, a method referred by the author as *scrambled BWT* or *sBWT* for short. It is easy to show that the sBWT results in a permutation of the text given in input, and that the reconstruction of the original data from its sBWT or the backward search of a pattern requires the knowledge of the order given to the characters composing the alphabet. Scrambling the BWT induces also a second garbling by the MTF transform, since the MTF output also depends on the ordering given to the letters of the alphabet.
This simple approach results in a poly-alphabetic substitution cipher that, for alphabets of suitable size and homophonic input data, can thwart exhaustive key-search attacks and cryptanalytic attacks based only on ciphertext knowledge (*ciphertext-only attacks*). Unfortunately, poly-alphabetic substitution ciphers are *deterministic*, in the sense that under a fixed key a particular (fraction of) plaintext is always encrypted to the same (fraction of) ciphertext. This fact makes them highly vulnerable to more advanced cryptanalysis such as *know-plaintext attacks* and *chosen-plaintext attacks*. These are attacks where an adversary knows a quantity of ciphertext and its corresponding plaintext, because she was able to find such plaintext or get it encrypted by the target, respectively. In fact, the insecurity of sBWT approach was proved in [38], showing its vulnerabilities to both those two types of attack.

In some respects, preserving confidentiality of genetic data and datasets but allowing their compression through the BWT-MTF pipeline is much more challenging than for natural language messages or free-text data.
First, genetic data and applications which operate on them are very exposed to know-plaintex attacks. For example, the Human Reference Genome (HRG) is used as a standard sequence reference, and most analyses estimate that substitutions in individual bases along a chromosome (i.e. single-nucleotide polymorphisms or SNPs) occur 1 in 1000 base pairs, on average, in the human genome.
Secondly, the genome alphabet composes of only four letters, so that using the sBWT approach results in a easy to mount exhaustive key-search attack.

In order to face these issues, I used a dual-stage encryption process in which the BWT input is subject to a poly-alphabetic substitution with respect to a suitable extension of the genetic alphabet, and the BWT-MTF-RLE0 pipeline is interspersed with a suitable set of instances of a very fast and cryptographically secure pseudo-random generator. This results in a full-text index offering high degrees of data compression and data confidentiality, as I will show in the sequel.

2.1 The EFM-index: data structures and algorithms

The proposed index is a compressed and encrypted full-text index tailored for genomic sequences that allows to perform searches and queries of strings on the IUPAC nucleic acid notation alphabet that I named Σ_{IUPAC}: it contains the five symbols $\{A, C, G, T, U\}$ corresponding to the DNA and RNA bases and a set of 11 additional symbols representing possible ambiguities caused by sequencing machines errors or inaccuracy (for example, the “N”symbol stands for “aNy”).

My index takes from the FM-index its main architecture, since it uses the pipe BWT-MTF-RLE0 to get compression and a similar organization in data blocks and super-blocks to manage auxiliary information. However, it makes use of multiple mechanisms in order to achieve confidentiality, which is not supported in the FM-index. Moreover, I carefully

optimized its design and implementation for the searching and querying of compressed and encrypted genomic sequences. An overall sketch of the index building process is given in Fig. 2.1.

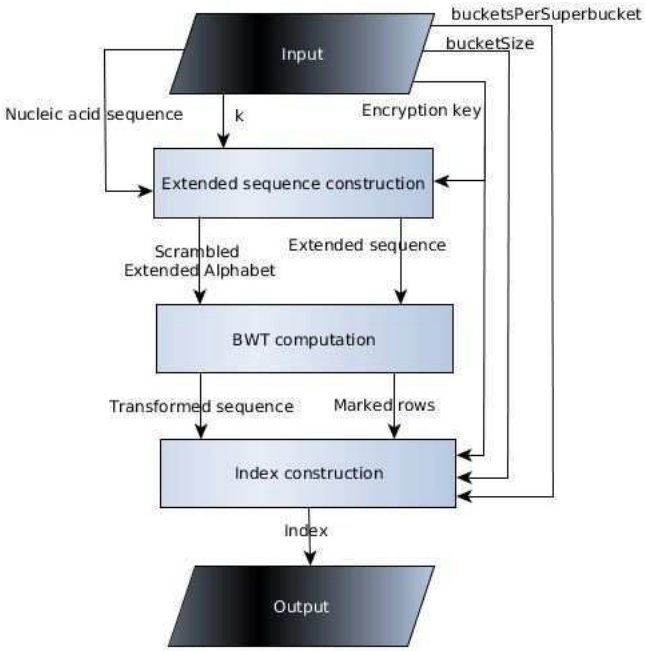


Figure 2.1: An overall view of the EFM-index building process

The index is constructed taking in input the following five main parameters:

- a plaintext T, which consists of an arbitrary length sequence of symbols from $\Sigma \subseteq \Sigma_{IUPAC}$;
- an integer k indicating the *extension* of Σ, that is the k-fold Cartesian product $\Sigma^k = \Sigma \times \ldots \Sigma$;
- the size $bucketSize$ of data blocks composing the index;
- the number $bucketsForSuperbucket$ of blocks for each super-block;
- an enciphering/deciphering key $secretKey$ consisting of a 64 byte array, for a total size of 512 bits.

The index construction composes of three main steps, as shown in Fig. 2.1: *Extended sequence construction*, *BWT computation* and the effective *Index construction*. I give details about each of them in the following subsections.

2.1.1 Extended sequence construction

The purpose of this phase, depicted in Fig. 2.2, is the construction of the input for the BWT-MTF-RLE0 pipeline from T. It is performed by the homonym module shown in

Fig. 2.1, and results in a recoding of T with respect to a permutation of the symbols of the k-extension of Σ.

The first step of such module consists in finding which of the symbols of Σ_{IUPAC} are actually present in T, in order to determine the alphabet $\Sigma \subseteq \Sigma_{IUPAC}$ for which the computations must have actually place. This step is performed for performance reason by module *Retrieve Symbols*. Indeed, in many cases one or more symbols of Σ_{IUPAC} might not be present in the input sequence (e.g. all the symbols indicating uncertainty are not present in the genome of Escherichia Coli), and this results in an exponential decrease in complexity for the index under construction.

The second step concerns the construction of the extended alphabet Σ^k and of its permutation $\widetilde{\Sigma^k}$, through the modules *Build extended alphabet*, *Compute scrambling key* and *Scramble alphabet*. The scrambling key is derived from the encryption key $secretKey$ thanks to a pseudo-random generator, as I am going to detail in Section 2.3.

Lastly, the third step is performed by module *Build Sequence*, which outputs the coding $\tilde{T}_k$ of T with respect to the symbols of $\widetilde{\Sigma^k}$.

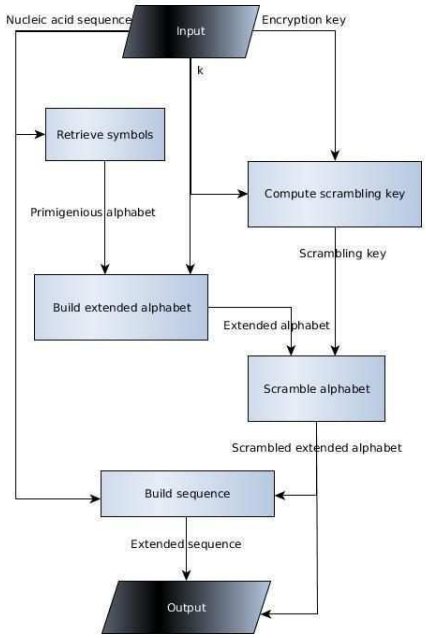


Figure 2.2: Extended sequence construction

2.1.2 BWT computation

The module *BWT computation* of Fig. 2.1 performs the following two main tasks:

- computation of $L = BWT(\tilde{T}_k)$ with respect to the ordering in $\widetilde{\Sigma^k}$;
- marking of the rows of the rotations matrix (see Section 1.1) that must have their position recorded in the index.

In order to minimize the time and the memory footprint needed to calculate the BWT of very long texts as genomic sequences, I used a "block-based" approach similar to that introduced in [21] to build the rotations matrix. The idea is very simple: a suitable set of "splitters" is chosen, so that the ranges of suffixes delimited by them can be ordered separately. Unlike [21], I used for choosing the splitters an algorithm devised by observing the statistical properties exhibited by k-mers in the genome of input. Moreover, I adopted a suitable approach to speed up the ordering in ranges where rotations share a long prefix of equal symbols, as in the case of DNA segments that are coded as $allN$ (i.e. long sequences of N symbols) since they are not well characterized. Overall, the above results in a new strategy for the calculation of the BWT of genomic sequences which allows to obtain a considerable increase in computing speed, and is well suited to be parallelized with a multi-threading approach, resulting in a significant speed-up already on systems having a couple of quad-core CPUs.

2.1.3 Index construction

This last phase (see Fig. 2.1) relates to the effective construction of the index, and it is schematically shown in Fig. 2.3. It is performed by an homonym module, which:

- through module *Split and remap* subdivides $L = BWT(\tilde{T}_k)$ in blocks and super-blocks, performing the remapping of characters;
- applies module *Encode bucket text* to each block. This module (see Fig. 2.3(b)) in turn does the following:
  - performs a Move To Front transform (MTF) followed by a Run Length Encoding of zeros (RLE0);
  - computes a keystream from the secret key $secretKey$ and the bucket number, as described in Section 2.3;
  - encrypts the output of the RLE0 with a stream cipher, using the keystream computed at the previous step;
  - uses a memory-efficient binary encoding for each block. Notice that I do not use the Multiple Tables Huffman (MTH) encoding adopted for the FM-index, because of the large memory footprint of its related decoding tables.
- uses module *Write index* to write the index on disk.

2.2 Pattern search

Like the FM-index, the EFM-index implements an exact pattern search by the backward search algorithm given in [15]. However, I reengineered that algorithm in order to obtain good performance on indexes built with respect to an extended alphabet. Compared to the extension Σ^k, a search for a single pattern $P \in \Sigma$ is indeed equivalent to search for a set of super-patterns. This set consists of super-patterns being associated with each of the k possible displacements ($d = 0, 1, \ldots, k-1$) between P and the symbols of Σ^k. Table 2.1 illustrates this circumstance for $\Sigma = \{\$, A, C, G, N, T\}$, $k = 4$ and $P = "ACGAACTGA"$. Symbol "?" denotes any single character of Σ, with the only constraint that the special

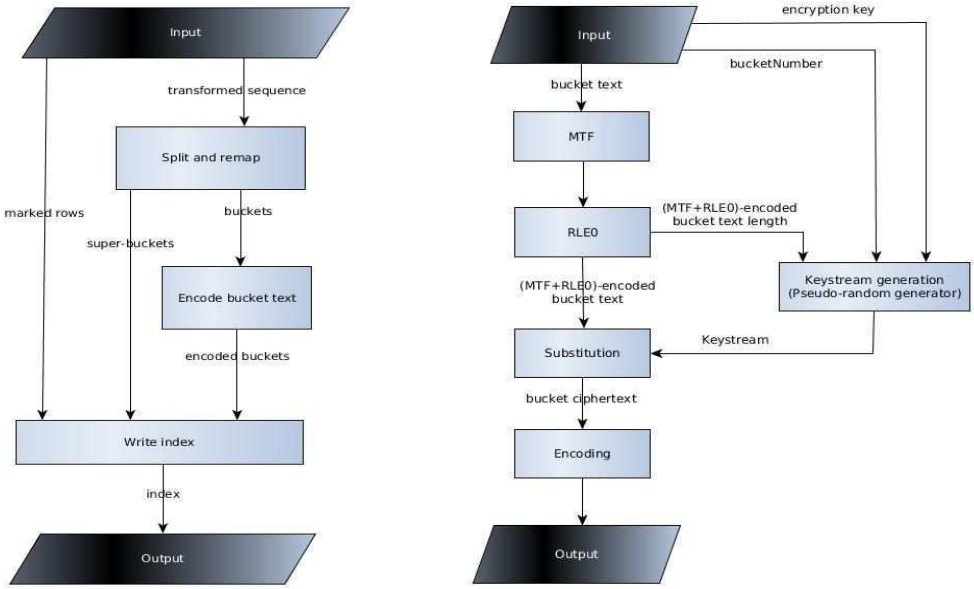


Figure 2.3: Effective index construction (a) and bucket text encoding (b)

symbol "$" cannot occur in a super-pattern. It is easy to see that the set related to each displacement is composed by exactly $(|\Sigma|-1)^{(k-|P| \mod k)}$ elements. Thus, the total number of super-patterns that must be searched in order to look for P is given by

$$k(|\Sigma|-1)^{(k-|P| \mod k)}, \tag{2.1}$$

which can be a significant value for some choices of Σ, k, and $|P|$. For example, in the case illustrated in Table 2.1, the set of super-patterns corresponding to $d = 0$ composes of the 125 strings of 12 characters having the required pattern as prefix. Thus, looking for $P =$ "$ACGAACTGA$" in a naive way would correspond to search for a total number of 500 super-patterns.

Table 2.1: Example of super-patterns with variable symbols [1].

Displacement	Super-patterns		
0	ACGA	ACTG	A???
1	?ACG	AACT	GA??
2	??AC	GAAC	TGA?
3	???A	CGAA	CTGA

[1] Searching for pattern "$ACGAACTGA$" in the alphabet $\Sigma = \{\$, A, C, G, N, T\}$ corresponds to search for the above set of super-patterns in Σ^4. Symbol "?" denotes any single character of Σ.

Performing the backward search algorithm a considerable number of times involves reading a large number of blocks from disk, which in turn can significantly degrade perfor-

Algorithm 3 *CheckLastChar*: a function called by algorithm *SuperPatternSearch* in order to verify if a row i satisfying $\hat{P}$ also satisfies P

```
1: function CHECKLASTCHAR(i,P,m)
2:     pos ← Locate(i);
3:     c ← Extract(pos + m − 1);
4:     if c like P_{m−1} then
5:         return pos;          ▷ The position pos is also that of the entire pattern P
6:     else
7:         return null;                                              ▷ no match
8:     end if
9: end function
```

Algorithm 4 SuperPatternSearch: an optimized algorithm to search for patterns over a k-extension alphabet Σ^k.

```
1: function SS-SEARCH(originalPattern)
2:     positions=[];                          ▷ Positions of the pattern occurrences
3:     superPatterns=computeSuperPatterns(originalPattern);
4:     for P in superPatterns do
5:         m ← length(P);
6:         P̂ = P_0 P_1 P_{m−2};
7:         P̃ ← P_{m−1};
8:         [ŝp, êp] ← backwardSearch(P, m − 1);
9:         for i in [ŝp, êp] do
10:            pos=CheckLastChar(i,P,m);
11:            if pos is not null then
12:                d ← displacement(P);          ▷ Displacement of the super-pattern
13:                add(positions,pos*k+d);          ▷ k is the alphabet extension order
14:            end if
15:        end for
16:    end for
17:    return positions;
18: end function
```

mance; in order to avoid this problem I designed a backward search algorithm optimized for super-patterns having variable super-characters, like those shown in Table 2.1. As it should be clear, variable super-characters can occur just in the first and/or last position of a super-pattern. Actually, variable-super characters in the first position can be managed through one more iteration of the backward search algorithm. Thus, it remains to describe the inner working of the algorithm in the case of a super-pattern with only the last super-character of variable type. Let $P = P_0P_1...P_{m-1} = \hat{P}P_{m-1}$ be a super-pattern with $P_i \in \Sigma^k$, and where P_{m-1} is its unique variable symbol. Searching for $\hat{P}$ requires a single execution of the backward search algorithm, and results in the range of rows with consecutive indexes $[\hat{sp}, \hat{ep}]$ in the array of suffixes [15]. On the other hand, it is easy to show that the rows in $[\hat{sp}, \hat{ep}]$ having P_{m-1} in their position $m-1$ are all and only the suffixes having as prefix the pattern P. Thus, an efficient way to find P consists in checking if the character in position $m-1$ for each of the rows $[\hat{sp}, \hat{ep}]$ is encompassed in the variable symbol P_{m-1}. Such check is performed by function *CheckLastChar* (see Algorithm 3), which uses the standard algorithms Locate and Extract of the FM-index [15] and returns the position of the entire pattern P if such pattern exists, and the null string otherwise .
The above overall search strategy is summed up in Algorithm 4.

2.3 Security

Roughly speaking, a security analysis requires the specification of both a *threat model* and a *breach model*. The first model aims at defining which kind of (harmful) interactions an opponent can have with a system, whilst the second aims to clarify in what circumstances such system should be considered violated. To be meaningful and effective these two models must of course be related and justified with respect to the actual system possibly under attack.

I will suppose in the sequel that the proposed index is implemented in a on-line database service which allows authenticated users to perform actions in function of their roles, as established by a suitable *role-based access control* (RBAC) policy [34]. At the lowest level, users can submit their genomic data in a suitable plaintext format, and get it stored by the service in compressed and encrypted form. At any time later, such users can just query for one or more patterns in their own stored data, or alternatively ask for retrieving their data in plaintext. Higher level users are allowed not only to manage their own stored data as basic user, but also data sets of users for which they got an authorization in function of their role.
From a cryptographic point of view, the data stored by the above on-line service are obtained from strings p of symbols of Σ_{IUPAC} of any length through an enciphering transformation:

$$\mathsf{Enc} : (p, k) \in P \times K \longrightarrow c \in C \ ,$$

where P denotes the plaintext space composed of the strings of symbols of Σ_{ext}, k is the key varying in a sufficiently large key space K, and each ciphertext c is a string of integers (see Section 2.1). The resulting encryption scheme is symmetric, that is it exists a deciphering transformation Dec such that

$$\mathsf{Dec}(\mathsf{Enc}(p, k), k) = p \quad \text{for any} (p, k) \in P \times K \ .$$

The above scheme actually represents an abstraction of the operations performed by the system using the full-text index introduced in this paper, with respect to which I made the following two assumptions:

Property 1 (Service integrity)**.** The flow of computations performed by the system in order to get transformations Enc, Dec and to derive the secret key k cannot be altered in any way. In particular, no read/write or other spurious operations can be interleaved during the computations of k, $\mathsf{Enc}(p, k)$ and $\mathsf{Dec}(c, k))$ performed by the system.

The above property can be obtained through the tamper resistance technologies introduced for software (see e.g. [1, 10]), used in conjunction with the various protection mechanisms available both in software and hardware [36, 11].

Property 2 (Key privacy)**.** The key $k(u)$ of an organism u cannot be inferred from the information owned neither by any user $u' \neq u$, nor by the System Administrator.

This last property follows from the fact that the system associates to each organism u (and thus to each genome) a pseudo-random key $k = k(u) \in K$, so that any two different u_1, u_2 get two different and unrelated keys $k(u_1), k(u_2)$. Moreover, each key $k(u)$ is stored in encrypted form, and its decryption is managed for u by the system, in a way that cannot be circumvented by any user, also with the highest privileges. These features can be easily realized thanks to suitable cryptographic primitives, such as *key derivation functions* and *asymmetric ciphers* [29].

2.3.1 Threat model

Concerning the threat model, I considered an adversary that can possibly be registered to the database service, but that in any case can interact with it in a more powerful way than any other user. This encompasses the case of *insider attacks*, that is attacks performed by entities that have got authorization to interact in some way with the system. On the basis of various studies (e.g. [7, 12]), these attacks are by far the most dangerous and frequent threats, since opponents are already on the inside and they enjoy a certain implicit trust. Besides the fact of being an insider, the adversary is allowed to:

- read as much as ciphertext it wants, where the ciphertext can belong to any of the authorized users on the system;
- have plaintext of its choice encrypted by the service with any key $k \in K$.

In other words, the adversary can have access to the data repository of any user, and can impersonate each of them with respect to the encryption service. However, the adversary has also some limitations that easily follows from service integrity and key privacy (see Properties 1 and 2). More precisely, the adversary cannot:

- query for known patterns and/or ask for the deciphering of any data which is different from that it is entailed to manage as a consequence of its role as authorized user;
- perform *meet-in-the-middle* attacks [29], that is it cannot interleave any operation during the processes of indexing and searching.

2.3.2 Breach model

One main concept in modern cryptography is that of *semantic security*, which represents the notion of *breach* of message confidentiality, and relates to the improvement an adversary can get in its knowledge of a plaintext by observing the related ciphertext plus some other ciphertext obtained with the same key from plaintext of its choice [29].

Translated in my case, the goal of the adversary would be to gain some more information about the genomic data of one or more organisms for which it is not a *trustee*; that is, it did not get the authorization to operate with their genomic data. Let T denote the the set of organisms for which the adversary is a trustee, and let $I = I(u)$ be some auxiliary information the adversary known about a given organism u, besides its ciphertext $c = c(u)$. In the worst case, $I(u)$ can consist of nothing, as when the adversary does not even known the species of u.

More often, however, the adversary can dispose of one or more reference genomes for u, genomic (sub-)sequences of u's relatives, or even of the same u. Because of the capabilities assumed for the adversary (see Section 2.3.1), the above turns out in the *chosen-plaintext* attacks considered in case of semantic security, that is when the adversary can obtain ciphertext from plaintext of its choice, encrypted with the same key of the target ciphertext. Ultimately, in my case the following should be an equivalent problem to that posed by semantic security:

Problem 1. Let u denote an organism whose genome is stored in the system, where $u \notin T$. Given the ciphertext $c = c(u) \in C$ and possibly some auxiliary information $I = I(u)$, find some more information on plaintext p, where $c = \mathsf{Enc}(p, k(u))$.

Of course, the above definition is quite informal and its scope is far from being the starting point of a provable security approach, which is instead often the case for the technical notions of semantic security. After all, the security in the proposed algorithm is in the realm of information security rather than complexity-theoretic security.
On the other hand, Problem 1 specifies some important kinds of cryptographic attacks that my system has to face, allowing to focus the security analysis. These attacks can as usual be divided into the following two main categories, depending on whether they do or not use cryptanalytic techniques.

2.3.3 Key-search attacks

Exhaustive key-search attacks consist in decrypting the target ciphertext c with each trial key through the entire key space, and discarding those keys which do not yield the plaintext p corresponding to c [29]. In natural languages this is possible, also without any preliminary information on p, because of redundancy and grammatical patters contained in plaintexts: the attacker can simply try all possible keys until eventually the decryption of c, possibly, gives rise to a meaningful plaintext (in the language in which the original plaintext is supposed to be).
Although no such redundancy or grammatical rules are known for genetic code, at least at the time being, nevertheless an adversary can mount a kind of attack described in Problem 1 that results in a termination condition for a key search. Given a sub-sequence c' of the target ciphertext $c(u)$, the adversary can indeed make an *hypothesis* $\sigma(c', u)$ about the related genomic sequence for u, and then searching for a key k such that the decryption of c' matches $\sigma(c', u)$. Thus, the goal of the adversary is to solve the following problem:

Problem 2 (Exhaustive key search)**.** Given $c' \subseteq c(u)$ and an hypothesis $\sigma(c', u)$, find $k \in K$ such that $\mathsf{Dec}(c', k) = \sigma(u)$.

An hypothesis $\sigma(c', u)$ can be easily obtained from a reference genomic data (e.g. the HRG, or a know genome from a relative of the individual being under attack), by assuming that it has undergone appropriate mutations.

It can worth to notice here that exhaustive key search are the only key-search attacks concerning my system, since *dictionary* attacks are ruled out by the system key derivation policy described before Section 2.3.1.

My approach thwarts exhaustive key-search attacks for two reasons (see Section 2.1):

1. The enciphering/deciphering key k is 64 byte long, giving rise to a key space of 2^{512} elements. This is more than 10^{77} times that provided for the strongest version of the current standard for encryption, AES-256 [31], which is supposed to be safe for many decades to come;

2. Unlike the method proposed in [25], searching for one or more patterns in p through its scrambled BWT requires the knowledge of k, because my method requires a partial deciphering of the index before performing any search.

2.3.4 Cryptanalytic attacks

The cryptanalytic attacks threatening my index range from ciphertext-only to chosen-plaintext attacks.

In *ciphertext-only* attacks it is assumed that the adversary - besides knowledge of all the details of the encryption function, which is a basic assumption in modern cryptography known as *Kerckhoffs' assumption* [29] - has no additional information other than the ciphertext, so that its cryptanalytic efforts root on flaws in the encryption function and/or correlations between pattern statistics in the ciphertext and its plaintext (*frequency analysis*). In [25] compression is suggested in order to remove redundancy in the data and to produce an homophonic sequence of input for the sBWT. An homophonic sequence is when the frequency distribution of symbols in the sequence is flat, so that an adversary cannot benefit from applying statistical attacks (monogram, digram and other statistics). In other words, statistical attacks on sBWT can be mounted if and only if its input does not consist in an homophonic sequence.

In my approach, as detailed in Section 2.1.1 and depicted in Figure 2.2, I give in input to the BWT a sequence obtained by applying a permutation to the k-extension Σ^k of the alphabet Σ. It is easy to show that such a permutation turns out in a poly-alphabetic substitution for the symbols in the input sequence, and that the multiplicity of substitutions increases with k. Since a poly-alphabetic substitution flattens the statistics in a sequence, then the sequence I pass in input to the BWT is much more homophonic, the greater is the value k. Actually, provided that module Encode bucket text (see Section III-C and Figure 5) works on blocks with $bs \geq 8K$, test results showed that a good degree of flatness in the frequency distribution of symbols is achieved already for $k \geq 4$, and that any residual correlations between pattern statistics in the ciphertext and its plaintext are completely ruled out after the application of the stream cipher to the output of RLE0.

However, the most dangerous threat for my system are *chosen-plaintext* attacks. In such an attack, an adversary selects a plaintext of its choice, submits it to the system and gets back the related ciphertext. Because of the high percentages of similarities in genomes of different individuals, these attacks (alongside with *know-plaintext* attacks, that represent a weaker variant) are actually feasible in my application scenario.

On the other hand, *deterministic* encryption schemes like that proposed in [25] are very vulnerable to this kind of attacks. This is because, with the same encryption key, the same ciphertext corresponds to identical (portions of) plaintexts.

Our approach thwarts such attacks thanks to the second (and last) stage of encryption, since module *Encode bucket text* realizes a *probabilistic* encryption [29]. Indeed, in this stage I encrypt the output of the RLE0 with a XOR-style cipher, using a keystream which is computed from the secret key $secretKey$ and the bucket number (see Section 2.1.3). This, assuming that the keystream is pseudorandom, guarantees that different and unrelated encrypted blocks correspond to the same blocks of plaintext, and that the cipher itself behaves like a pseudorandom function, fully obfuscating frequency and pattern differences between two related plaintexts.

In my implementation I used the Salsa20 stream cipher in its full version Salsa20/20 [6] to get the keystream. Salsa20 is one of the ciphers selected as part of the eSTREAM portfolio of stream ciphers [2], and has been designed for high performance in software implementations. It has a compact source code and uses few resources and inexpensive operations that makes it suitable for implementation on a wide range of architectures. Moreover, it has been designed to prevent leakage of information through *side channel analysis*. As of 2014, there are no published attacks on Salsa20/12 or the full Salsa20/20; the best attack known breaks just 8 of the 12 or 20 rounds.

2.3.5 Security tests

I conducted a set of security tests in order to corroborate through experimental evidence our security analysis: the goal of these tests was to verify that the implemented cipher behaves as a pseudorandom function, allowing protection also against chosen-plaintext attacks.

Figures 2.4, 2.5 and 2.6 show the results of two kinds of test.

The first kind measures the differences in the occurrence frequency of symbols between two ciphertexts: symbols are ordered as increasing integer values along the x-axis, and each y-value is the difference between the number of times a symbol appears in the two ciphertexts.

The second kind of tests have produced some scatter plots that measure the variation in the position of each symbol when passing from one ciphertext to the other: symbols are split in ranges, in function of the position they have in one ciphertext (x-axis) and in the other (y-axis). Symbols having the same position appear on the diagonal of a square, whilst the most a square is filled, the greater is the number of symbols that have changed their position in the transition from one cryptogram to the other.

In any case, the two ciphertexts were obtained with the same key from a reference genome of an organism and its modification is due to a simple mutation.

Figure 2.4 gives experimental evidence of the fact that ciphertexts obtained by applying only the sBWT do not obfuscate the changes in plaintexts, whilst our cipher does. Indeed, a simple perturbation in the plaintext (specifically, a single substitution between A and T bases in the 10-th position of the E. Coli reference genome) turns out in both *diffusion* (the perturbation affects many symbols of the ciphertexts) and *avalanche effect* (the intensity of the perturbation is amplified in cryptograms).

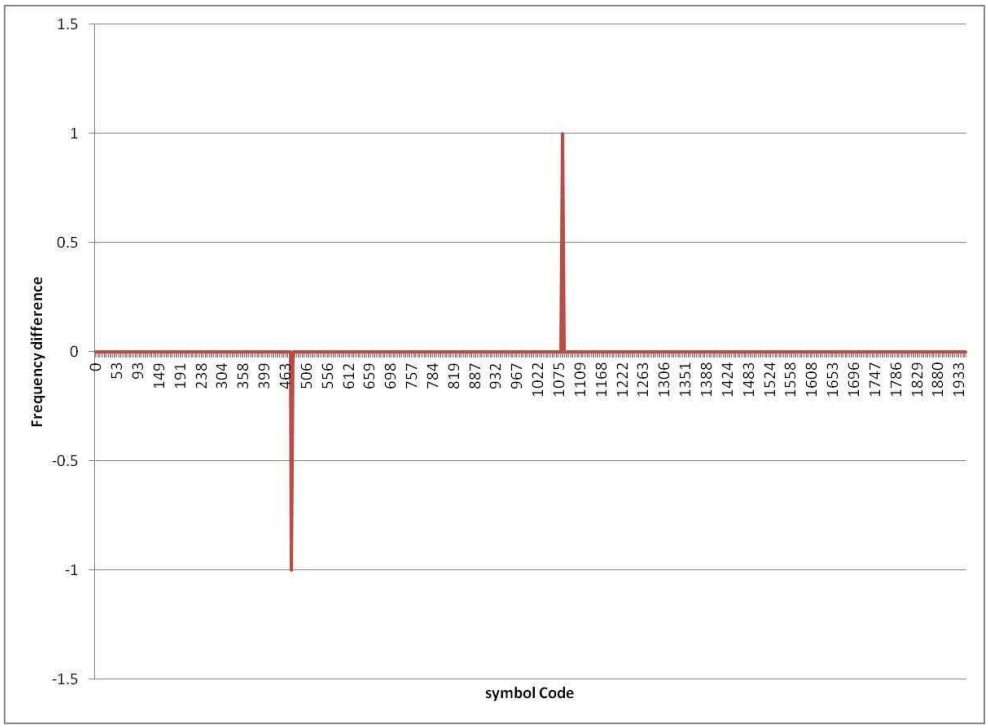
These results are confirmed by Figure 2.5, which shows two scatter plots related to the same simple perturbation in the plaintext considered in the previous case. However, Figure 2.4 shows the average case of tests performed for different values of the encryption key, whilst Figure 2.5 reports the best and worst cases. Notice the relevance of scattering also in the worst case, where a lot of symbols between positions 700000 and 900000 change location.

Finally, Figure 2.6 illustrates what happens as consequence of the multiple insertion identified in [18] for E. Coli-K12 MG1655. This last example shows that the cipher realized through our full-text index is such that bigger perturbations in plaintext turns out in more diffusion and avalanche effect.

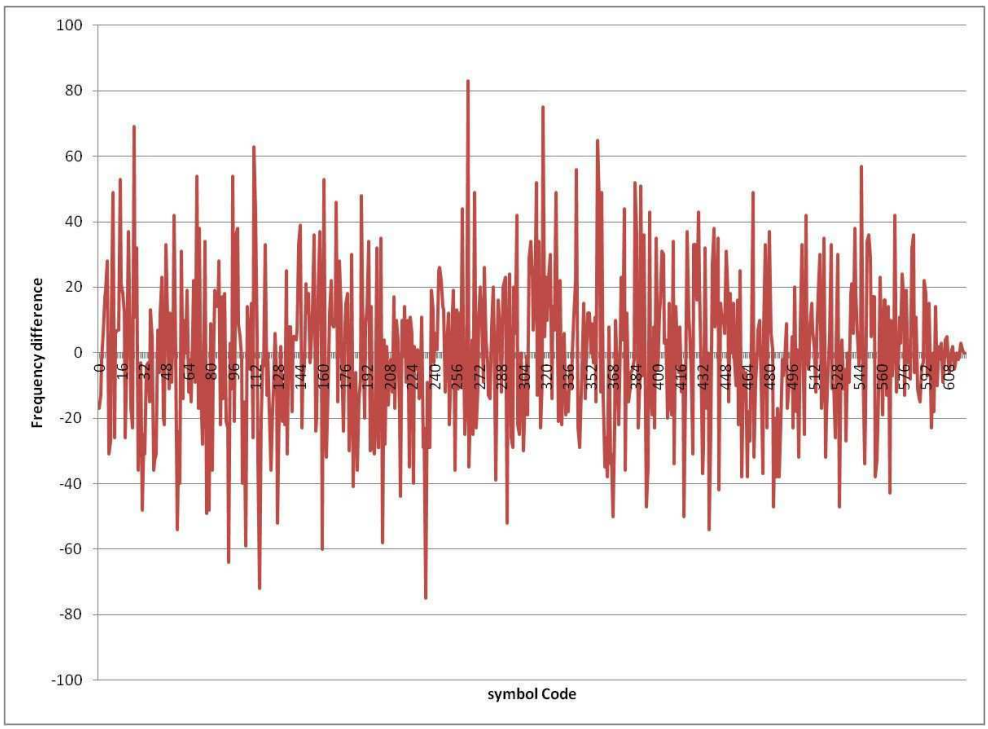
2.4 Performance results

In order to measure the efficiency of the approach and to compare its performance with the FM-index, I developed a prototype of the EFM-index using the Java language; this section illustrates some of the tests I performed through that prototype.

The tests were performed on a machine with an Intel i7 4500U microprocessor having clock frequency of 1.80 GHz and 4MB of L3 cache , main memory of 8 GB, and an SSD hard drive of 512 GB. The software stack consisted of Ubuntu 13.10 and Java HotSpot^{TM}64-Bit Server version 1.7.0.51_b13. It is well known that the result of compiling a Java program, unlike what happens with other languages such as C++ gives not rise to code directly executable by the microprocessor of a physical machine, but to the bytecode, i.e. a machine language for a virtual platform called Java Virtual Machine (JVM) . In order to optimize the

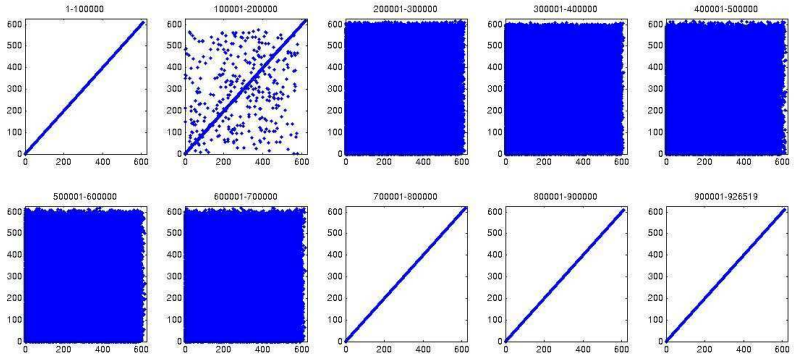


(a)

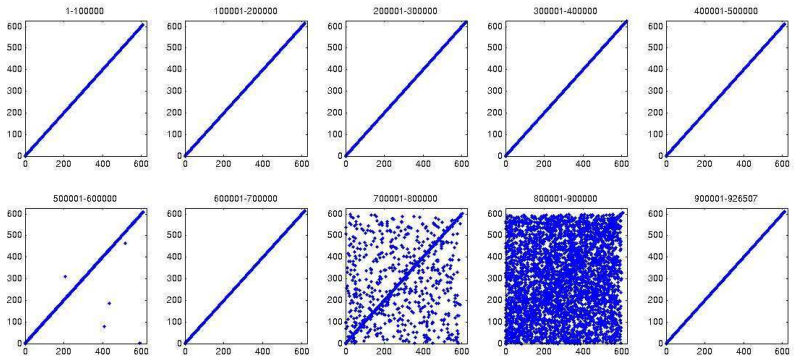


(b)

Figure 2.4: Differences in frequencies of symbols in case of ciphertexts obtained applying only the sBWT (a) versus our approach (b)

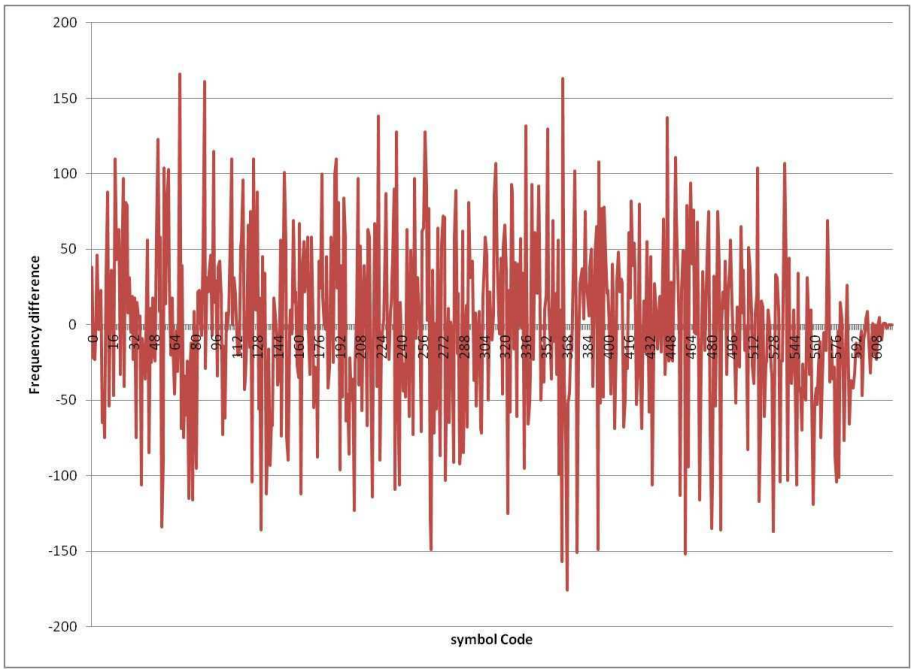


(a)

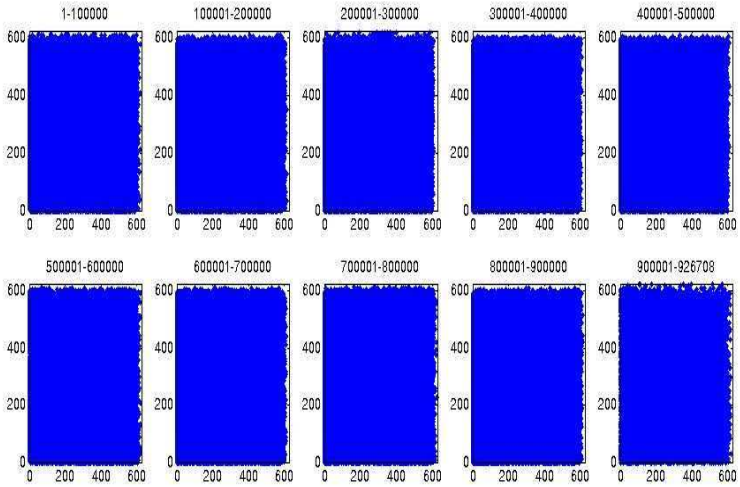


(b)

Figure 2.5: Scatter plots of ciphertext symbols in the best case (a) and in the worst case (b), as resulting from tests with different encryption keys



(a)



(b)

Figure 2.6: Plots related to the insertion identified in [18] for E. Coli-K12 MG1655. Plot (a) and (b) show frequency differences of ciphertext symbols and changes in their locations w.r.t. the reference sequence NC_000913, respectively

performance of an application, the JVM uses the so-called *Just-in-Time* compilation. Just-in-Time compilers are essentially very fast traditional compilers that translate "on the fly" bytecode into native code. The new versions of the JVM (so-called HotSpot) make use of an adaptive optimization technique of Just-in-Time compilation, which is based on the following observation: the vast majority of programs run most of the time a very small portion of their source code. Therefore, rather than compiling just-in-time each method before its execution, the HotSpot VM runs first the bytecode to determine the performance of critical parts ("hot spots"), and then compiles them into native code[13]. Therefore, the performance of a program written in Java are those of an interpreted language before the above operation, and become comparable to those of native code only after it. In order to take into account that behaviour and to get more reliable results, the figures report data obtained after a first batch of tests, enough to allow the "warming up" of the JVM with the compilation of the hot spots.

The tests concerned reference sequences from different human chromosomes, extracted from the assembly GRCH37 downloaded from the FTP site of the *1000 Genomes Project*. For each test sequence, I built compressed and encrypted indexes corresponding to different values of k (degree of extension of the alphabet Σ), bs (size of blocks) and percentage MRP of marked rows (i.e. relative number of lines marked for searching purposes). Then, I conducted two types of tests aimed at detecting the performance of our prototype:

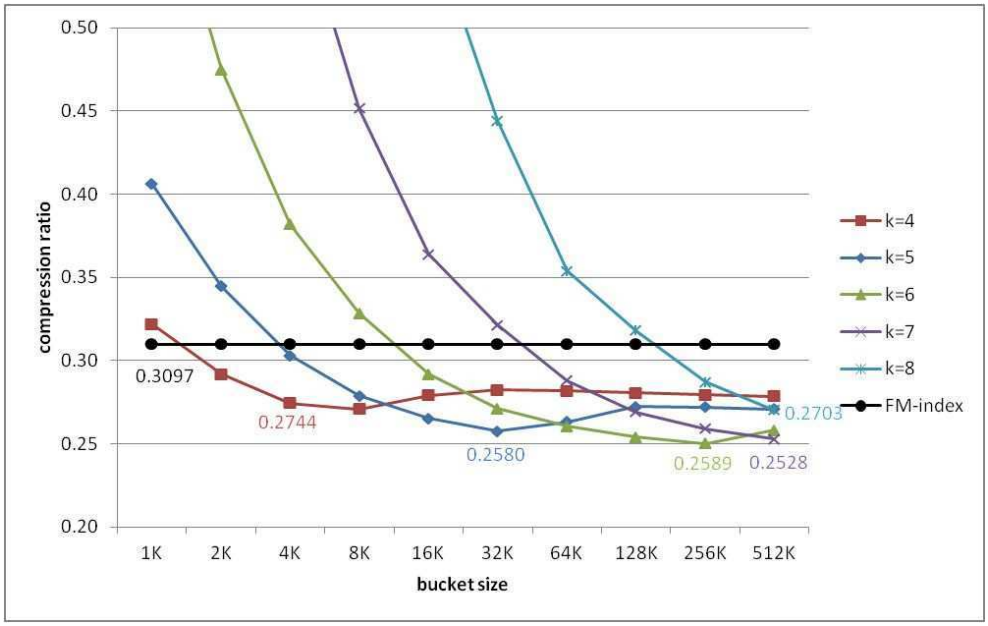
- search of patterns of predetermined lengths (through count and locate);
- Electronic polymerase chain reaction (e-PCR) operations. An e-PCR consists in the extraction of the fragment of the nucleotide sequence between two terminations, denominated left and right primer.

For brevity, I report only the results for chromosome 20, one of the smallest, and for chromosome 1, the largest one. In order to compare the performance of my prototype with a tool of reference, I ran the same tests using the implementation of the FM-index version 1 downloaded from [16].

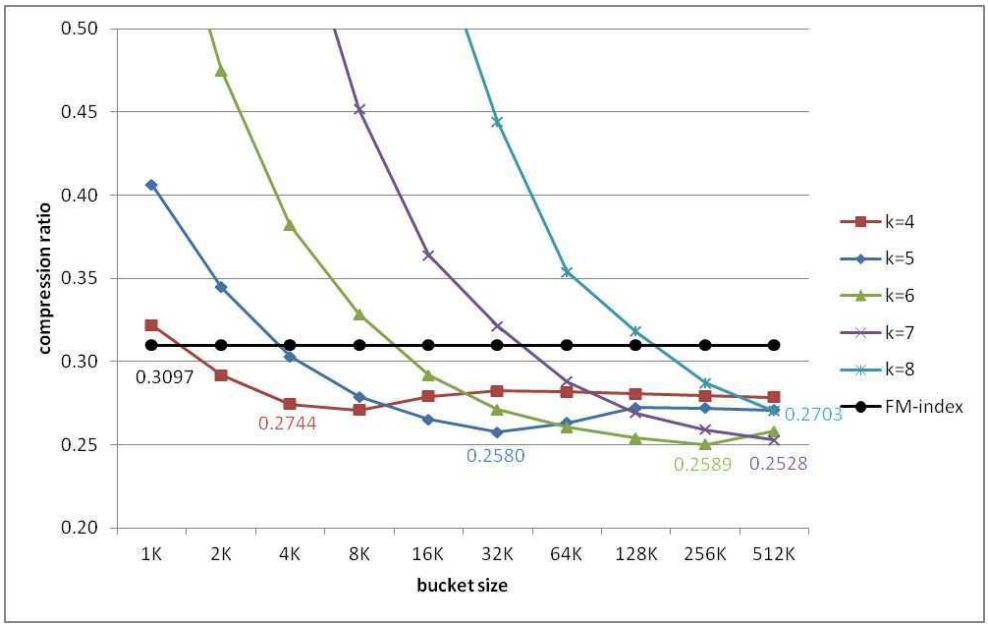
2.4.1 Compression ratios

The graphs of this section show compression ratios (percent values) versus block sizes bs for a fixed percentage MRP of marked rows and different degrees k of extension for alphabet Σ. Block sizes bs are expressed as kibi (K) of characters in Σ^k, since this corresponds to the logical segmentation of our index as result of the splitting phase (see Section 2.1.3). Figure 2.7, which concerns human chromosome 20, clearly shows that the best compression ratio is achieved by increasing bs as k increases. Precisely: up to $bs = 8K$ it is achieved for $k = 4$, and from $bs = 16K$ to $bs = 32K$ it results by choosing $k = 5$. From $bs = 64K$ to $bs = 256K$ the best choice is $k = 6$, whilst $k = 7$ is the best choice for $bs = 512K$.

It should be noted that the use of too large block sizes causes the production of an index with a very low number of blocks, which is virtually equivalent to do no indexing at all. This degrades performance in searching, as I will show hereafter. The horizontal line corresponds to the compression ratio for a type Fat 1 FM-index, which can perform both count and locate operations [15]. These results show very clearly that my prototype allows for a very good compression efficiency. Moreover, compression ratios are better than for the FM-index (which does not make use of encryption) for values of bs and k that, as I will



(a)



(b)

Figure 2.7: Compression ratios achieved for the human chromosome 20 with (a) $MRP = 2\%$ and (b) $MRP = 5\%$

discuss in Section 2.3, offer appropriate levels of security. Similar results hold for the human chromosome 1 (see Figure 2.8), and all the other tested chromosomes (human or not).

As I previously said, the use of large values for the size of blocks results in a poor indexing, and thus in low search performance. This is clearly shown by graph 2.8(b), where the number of blocks composing our index for chromosome 1 is plotted versus different values of k and bs. For example, for $bs = 512K$ the index has just 60 blocks in front of the 15212 blocks corresponding to a block size of $4K$.

2.4.2 Pattern search performance

In order to evaluate the search performance of my prototype, I run test sequences for patterns of predetermined lengths l. For each l, I randomly chose 100 patterns from a given chromosome. Then, for each of the above patterns I performed a search (through a Count and a Locate operation) for a given set of values of the parameters k, bs and MRP. The obtained values are the average searching-time values computed over those set of patterns and each triple (k, bs, MRP).

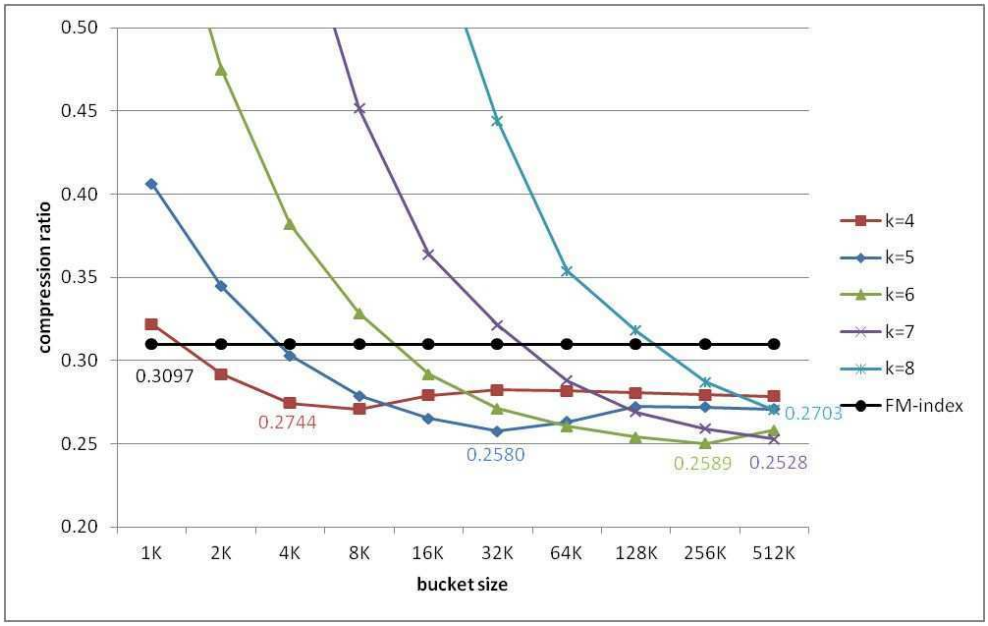
For reason of space, I report only the results for patterns composed of 50 and 500 nucleotides. I chose the length $l = 50$ since it corresponds to short patterns that rarely have multiple occurrences in a given chromosome, whilst the value $l = 500$ allowed me to test the prototype for large sequences (of course, also in this case there was a single occurrence of the pattern in the given chromosome). I decided to search for patterns occurring just one time in order to get more reliable timing figures: in fact, in this way the average search time matches with the average search time for occurrence, and this is appropriate because the search time for a pattern depends on the number of its occurrences.

The obtained results show that the searching time increases with the block size bs for a fixed extension degree k, and with k for a fixed bs. Moreover, as shown in Figure 2.9(a), the searching time for patterns of small sizes decreases as the percentage of marked lines MRP increases. Such behaviour does not occur in case of larger patterns (see Figure 2.10(b)), for which performance becomes slightly worse by increasing the MRP, and I think that it is due to the way in which my prototype handles the marked rows: they are stored in the block headers, and searching for large patterns requires a larger number of blocks, thus resulting in more I/O operations.

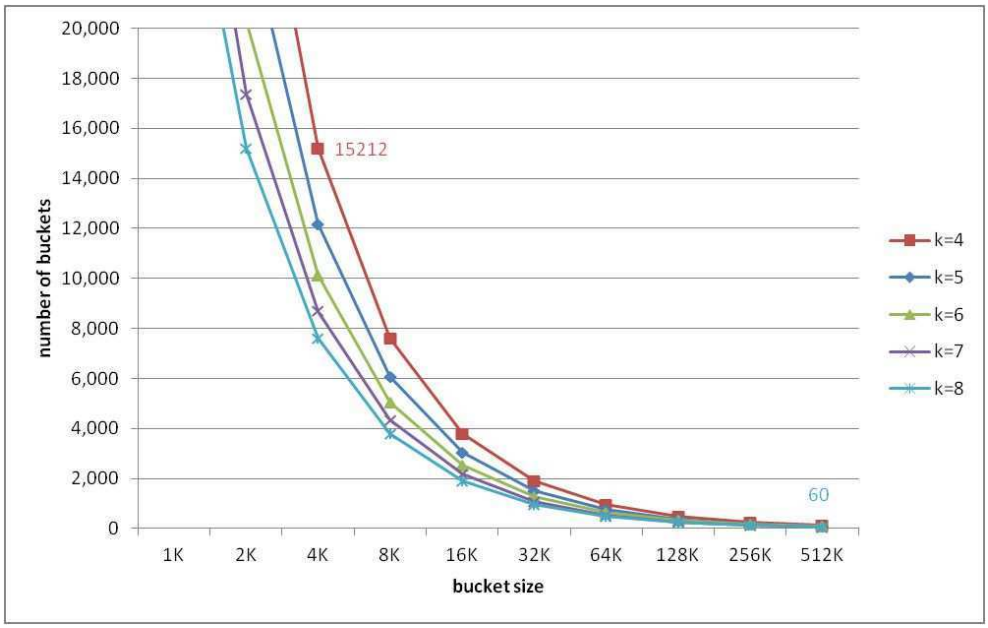
As before, in order to compare the performance of our prototype with a tool of reference, each figure reports a horizontal line corresponding to values obtained by performing the same operation with the FM-index. In general, the FM-index achieves better performance: searching times were better than for the FM-index only in some cases, for values of k and bs too small to ensure good compression ratios. However, the searching times achieved with our prototype are of the same order or just one order of magnitude larger than those achieved with the FM-index, and in the worst cases they are few tens of milliseconds (ms). In particular, for $k = 4$ and $bs = 4K$, our index outperforms the Fat FM-index in compression ratio, offering a good level of confidentiality (see Section 2.3) at the price of an increment of only 8 ms in searching time.

2.4.3 e-PCR performance

As I wrote above, an e-PCR consists in the extraction from a genomic sequence of the fragment between the initial and final terminations, called, respectively, the left and right primers. The e-PCR is the equivalent in silico of a polymerase chain reaction in vitro, and it is realized through the following: (a) a Count operation followed by a Locate operation,

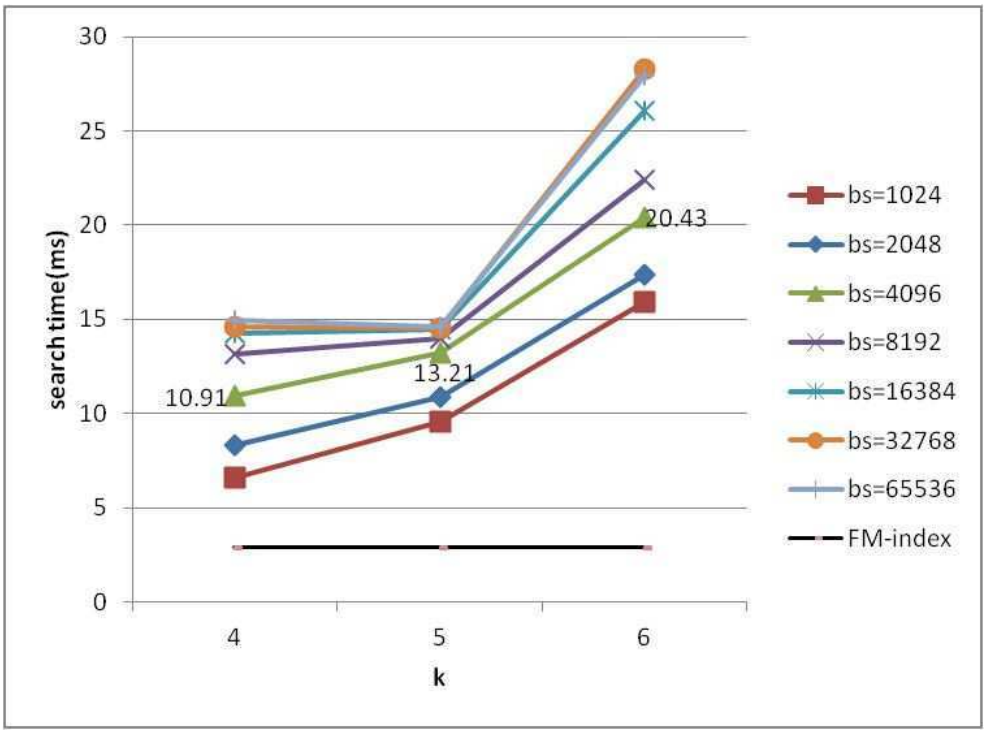


(a)

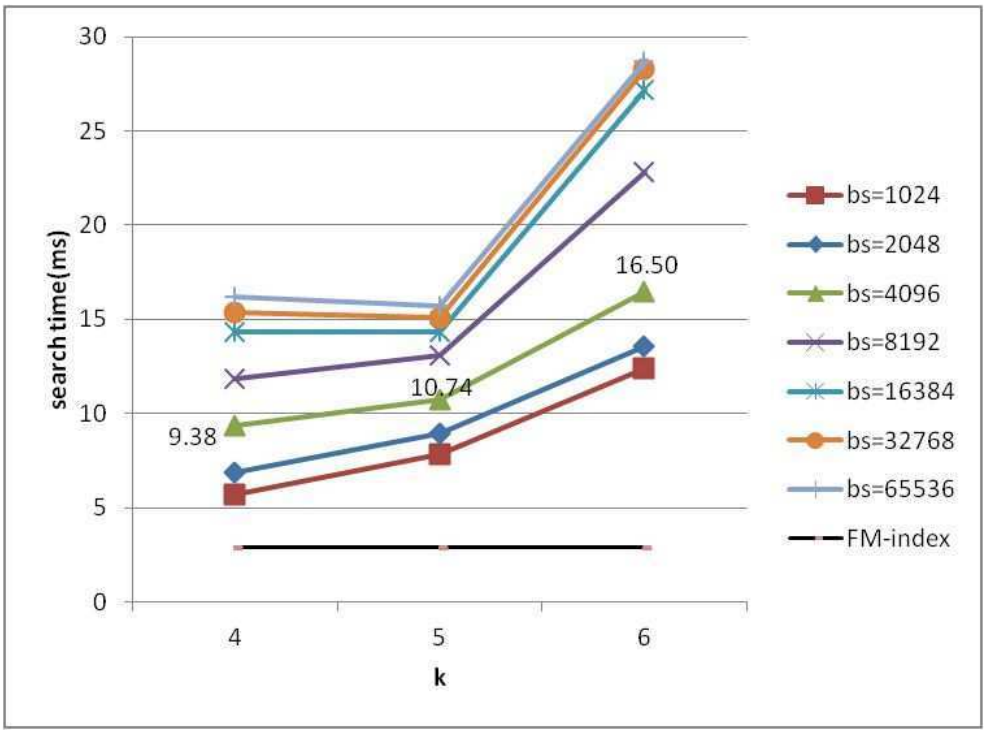


(b)

Figure 2.8: (a) Compression ratios achieved for chromosome 1 with $MRP = 2\%$, and (b) number of blocks composing the related index

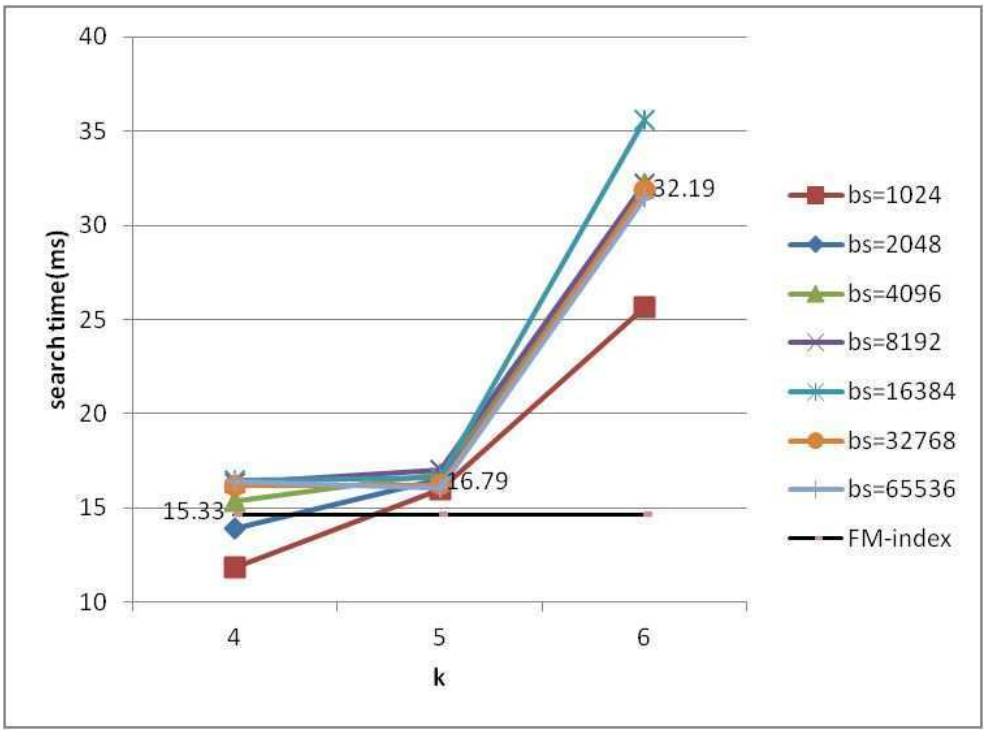


(a)

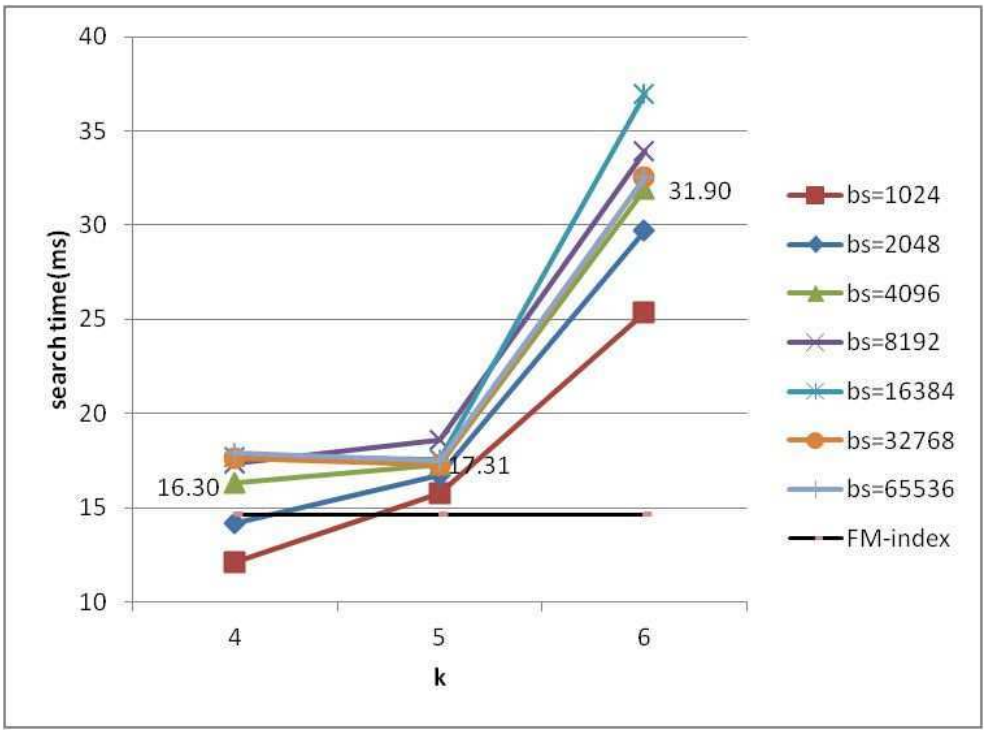


(b)

Figure 2.9: Average times for searching for a (single) pattern of length $l = 50$ in human chromosome 20. Time values are plotted in different colours for $k = 4, 5, 6$ and relates to: (a) $MRP = 2\%$, and (b) $MRP = 5\%$

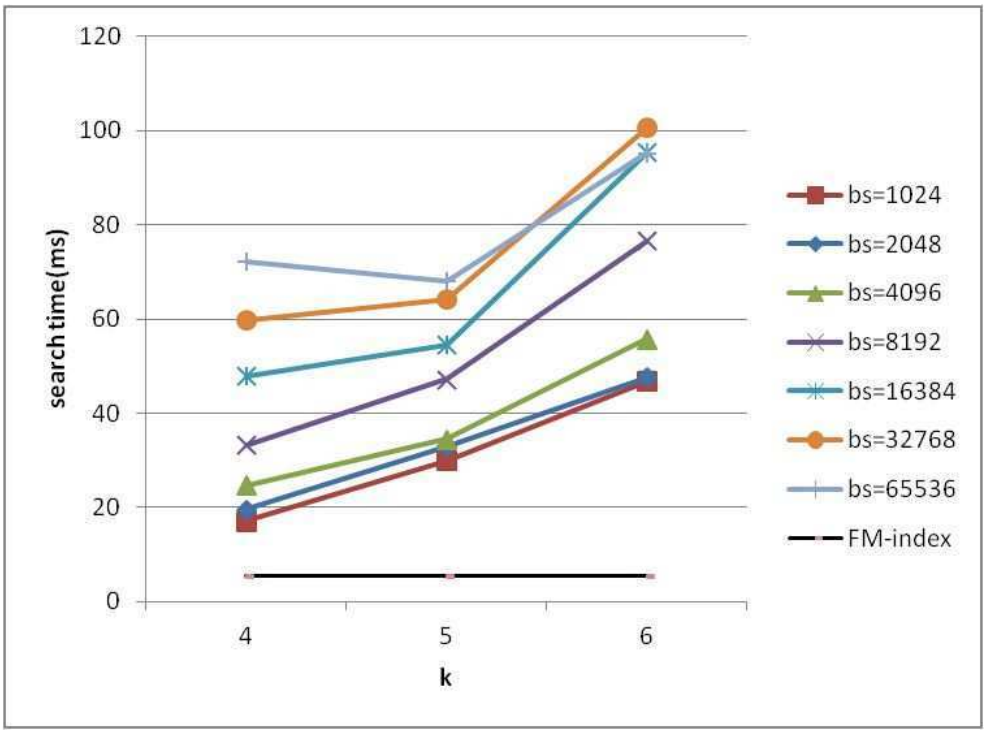


(a)

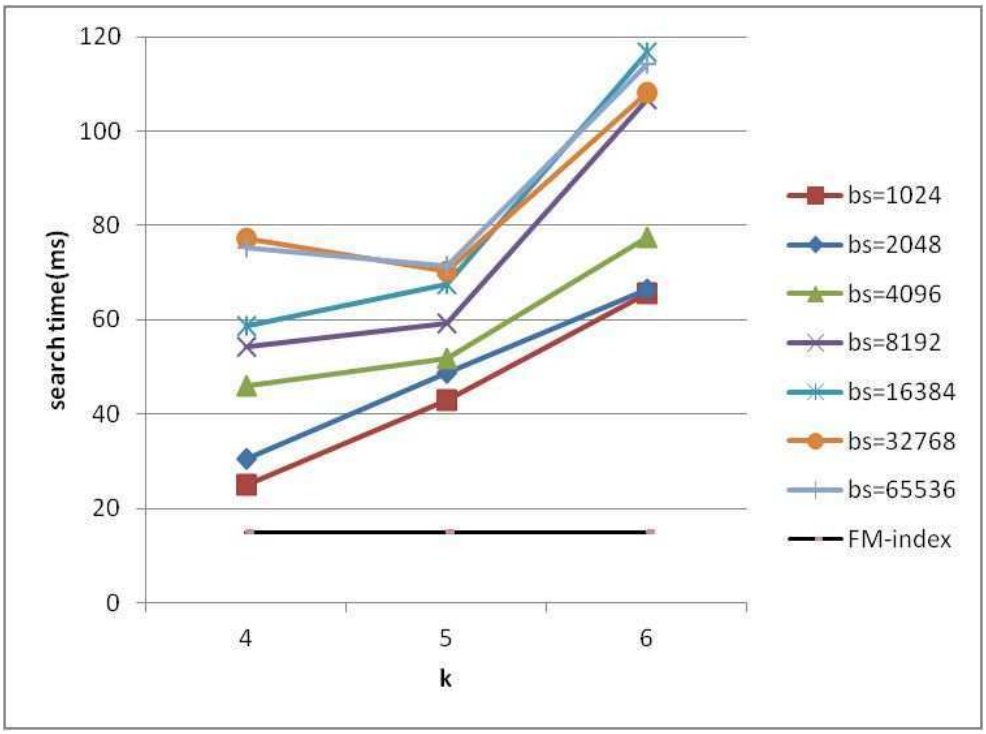


(b)

Figure 2.10: Average times for searching for a (single) pattern of length $l = 500$ in human chromosome 20. Time values are plotted in different colours for $k = 4, 5, 6$ and relates to: (a) $MRP = 2\%$, and (b) $MRP = 5\%$



(a)

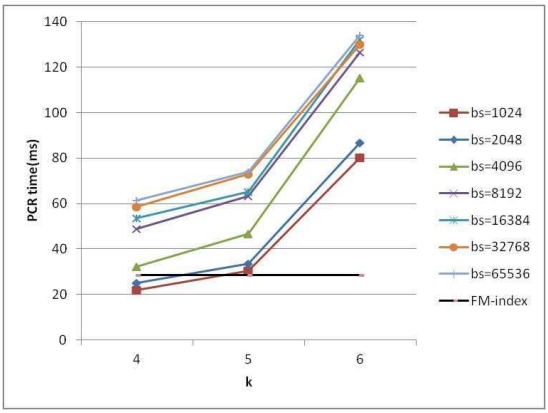


(b)

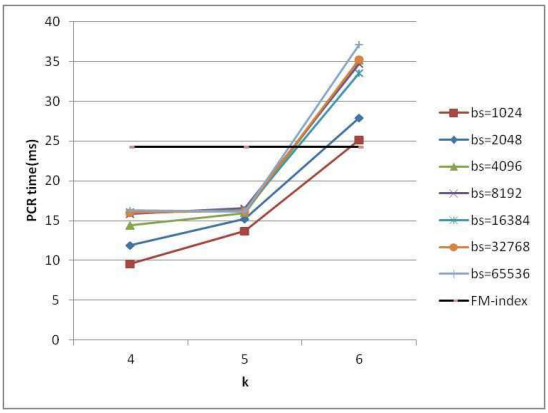
Figure 2.11: Average times for searching for a (single) pattern of length (a) $l = 50$ and (b) $l = 500$ in human chromosome 1. Time values are plotted in different colours for $k = 4, 5, 6$ and relate in each case to $MRP = 2\%$

performed twice in order to find the positions of both the left and the right primers, and (b) the determination of the fragment of the sequence which is comprised between the two positions above.

I conducted the tests to evaluate the performance of my prototype in doing e-PCRs as follows. We randomly chose 100 genetic *markers* from a database of the 1000 Genome Project [17], determining for each marker its name and its left and right primers. Then I performed the e-PCR for each of the above markers with respect to different values of the parameters k, bs and MRP. The figures shown are the average e-PCR time values computed over that set of patterns for each triple (k, bs, MRP). As shown by the graphs given in Figure 2.12, I obtained results comparable and in some cases better than those obtained with the FM-index.



(a)



(b)

Figure 2.12: Average times for performing an e-PCR on (a) human chromosome 1 and (b) human chromosome 20. Time values are plotted in different colours for $k = 4, 5, 6$ and relate in each case to $MRP = 2\%$

2.5 Collections of genomic data

In this section I show how the EFM-index can be used to store collections of genomic data, obtaining good compression ratios and search performance.

2.5.1 Method

Given a collection $C = \{S_1, S_2, \cdots, S_n\}$ of sequences, strings on the alphabet Σ, a compressed and indexed representation of the whole collection can be obtained[37] building a unique self-index on the concatenated string:

$$S_c = S_1 \# S_2 \# \cdots \# S_n,$$

where $\#$ is a character not belonging to Σ.

So, I extended the EFM-index implementation to handle a collection of sequences $C = \{S_1, S_2, \cdots, S_n\}$ on the alphabet Σ, so that it behaves as follows:

- it represents the single sequences in the k-mers extended alphabet Σ_k;
- it concatenates in a single string $S_c \in \Sigma_k$ the obtained representations, separating them through a $\#_k$ super-character (this super-character is obtained repeating k times a character $\# \notin \Sigma$);
- it builds an index on S_c;
- it stores the sequence identifiers in the index header.

I chose not to explicitly store the $\#_k$ separators positions, because they can be retrieved by a single very fast locate operation. Moreover, I slightly modified the locate operation to transform the global positions returned by the previously described EFM-index locate operation in sequence-relative positions, on the basis of the aforementioned separators positions.

2.5.2 Experimental results

In order to measure the efficiency of the above described method, I converted in C++ the prototype of the EFM-index originally written in Java and I extended it to handle collections of sequences; this section illustrates some tests that I performed through my C++ prototype.

To evaluate the EFM-index performance I built first of all the consensus sequences related to chromosomes 20 and 11 of 50 individuals of 1000 Genomes Project, on the basis of the BAM files downloaded from that project FTP site; then I built the EFM-indexes of the two obtained collections. In order to ensure a good compromise between security and performance, I chose to build the indexes with the following combination of parameters: $k = 5$, $bucketSize = 8192$ and $markedRowsPercentage = 2$.

Moreover, in order to compare the performance of my prototype with a state of the art software, I used the Sdsl C++ library, which implements some *succinct data structures*[20] usable to construct self-indexes like Compressed Suffix Arrays (CSA) and wavelet tree FM-indexes. I had to extend the wavelet tree FM-index supplied by that library so that it was able to manage collections of items and to report sequence-relative locations: I used the same approach of EFM-index, but with a separator consisting in a single $\#$ character.

It was impossible to perform comparisons with the FM-index version 1, because it can't handle texts larger than 2GB.

I performed the tests a on virtual machine hosted by a RHEV (Red Hat Enterprise Virtualization) 3.4 system with 196 GB of RAM memory and 4 Intel(R) Xeon(R) CPU E5-2697 v2 @ 2.70GHz 6-core processors.

Figure 2.13 compares the compression ratios obtained with EFM-index with those exhibited by the Sdsl library on three sequence collections of different size:

1. 20_FULL: the chromosome 20 sequences of the aforementioned *1000 Genomes Project* 50 individuals;
2. 20_1MB: the first 1 million bases of those sequences;
3. 20_5MB: the first 5 million bases of those sequences.

The compression ratios exhibited by EFM-index are better than those obtained with Sdsl, although both indexes are BWT-based: this is due to the alphabet extension used in my index, which helps the Move To Front transformation to achieve universal coding performance[5], i.e. those of a compressor having a data compression ratio which differs at most for a constant factor from that of the optimal prefix code.

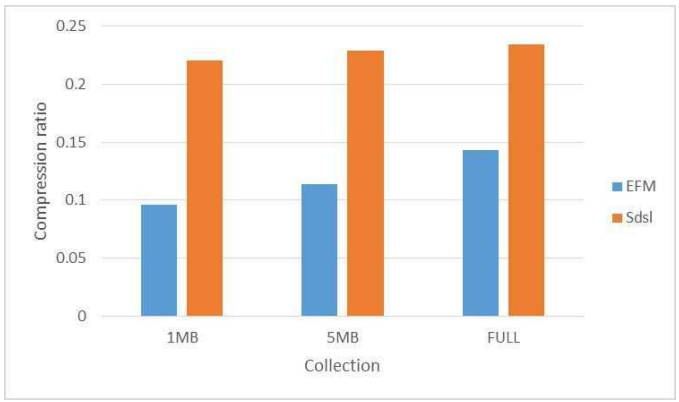


Figure 2.13: Compression ratio

Table 2.2 shows that the construction times of the EFM and Sdsl indexes are very close each others, despite the fact that EFM-index implements data encryption.

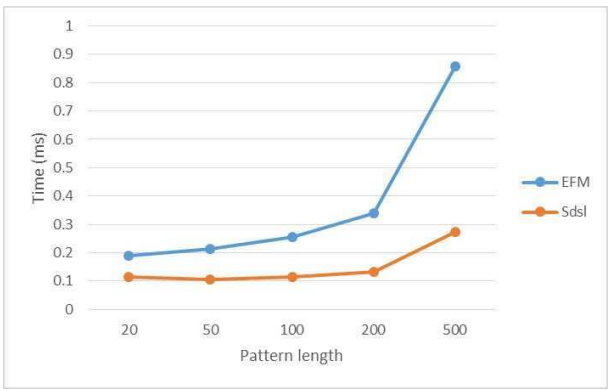
1MB		**5MB**		**FULL**	
EFM	Sdsl	EFM	Sdsl	EFM	Sdsl
22.94	20.08	106.9	132.1	2222	2061

Table 2.2: Construction times (s)

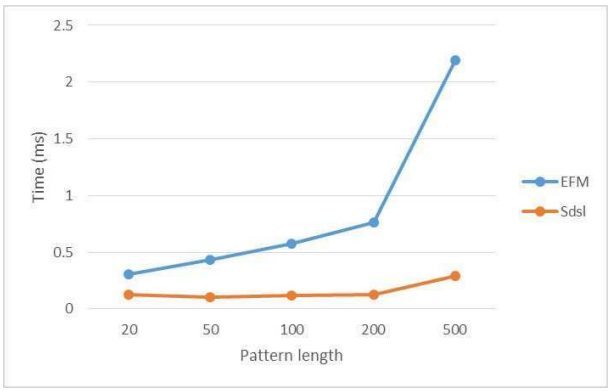
The time taken to perform a pattern search is proportional to the number of occurrences found. So, in order to evaluate the performance in search operations, I chose 500 patterns for several pattern lengths (20,50,100,200,500) and, for each pattern length, I computed the mean search time per occurrence, i.e. the mean of times taken by my prototype to report each pattern occurrence.

Figures 2.14 and 2.15 show, first of all, that those times grow with pattern length both for my index and the Sdsl one: this behaviour depends on the greater number of backward steps in BWT that are needed to search for a larger pattern. The indexes based on Sdsl library exhibit better performance, but in any case the EFM-index times per occurrence are very close to the Sdsl ones and they do not exceed 10ms; furthermore, it is important to consider that:

1. again, Sdsl implements no kind of encryption;
2. Sdsl needs to load the entire index in memory before starting a search operation, whilst my index loads in memory the minimal number of needed blocks during search operations: this is an EFM-index strength, but penalizes it in the described comparison.



(a)



(b)

Figure 2.14: Mean search (count+locate) times per occurrence on the collections (a) 20_1MB and (b) 20_5MB, for several pattern lengths

In conclusion, EFM-index exhibits good compression ratios, short construction times and optimal search performance also on collections of genomic sequences, outperforming in compression the Sdsl library despite of the encryption implementation.

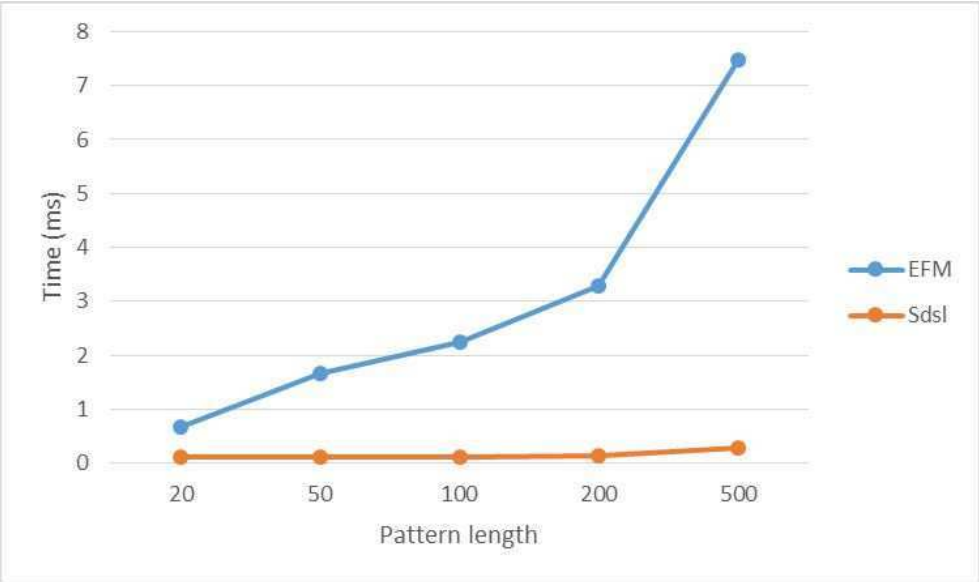


Figure 2.15: Mean search (count+locate) times per occurrence on the collection 20_FULL, for several pattern lengths

Chapter 3

ER-index: an index model designed to handle collections of similar genomic sequences

Introduction

The EFM-index presented in chapter 2 has two major issues:

- it is not suitable to handle collections of highly similar genomic sequences, as it cannot fully exploit the inter-sequence redundancy, defined as the redundancy that exists between the several sequences of a collection;
- its encryption model does not permit to store the genomic sequences of different individuals with distinct encryption keys within the same index.

The first issue strongly affects the compression ratio, while the second forces to create one index for each set of individual sequences whose access must be separately authorized; consequently, it is necessary to search for patterns in several indexes, potentially slowing down search performance.

In order to overcome the above problems I had to orient towards other models of indexes capable of storing a collection of sequences:

- exploiting the inter-sequence redundancy;
- encrypting the sequence of different individuals with distinct encryption keys;
- offering good search performance.

I focused my attention on indexes based on the Lempel Ziv parsing, and especially on the original LZ77 variant. Most of the self-indexes inspired to the Lempel Ziv parsing used the LZ78 variant, because the LZ78 factorization of a text has some interesting properties which allow[37] to design efficient pattern search algorithms; on the other hand, LZ78 does not permit to gain the most advantage from the high similarity of genomic sequences.

The first self-index based on LZ77 was presented in [22]: it offered good compression ratio and search performance, but its internal data structures were not designed to explicitly

handle the several items of a collection and so in my opinion that index is not suitable to be a valid starting point for my aims.

That index also does not exploit a fundamental property of my application domain: I need to store human genomic sequences and I have a reference sequence for it.

Both the LZ77 and LZ78 algorithms compress a text T building a dictionary of phrases occurring in the yet scanned part of T and then encoding the remaining part of it in term of that phrases; however, for human genomic sequences there is no need to create such a dictionary, in that it is already available once and for all: it is the reference sequence.

The first attempt to "*referentially compress*" a collection of individual genomes, just encoding the differences with respect to a reference sequence, was made by the authors of [8]: that work, as [26] and [27], aimed to build data structures suitable to efficiently compress the collection, while allowing fast random access to parts of it. Pattern search still remained an open question.

The problem of efficiently searching patterns in a such index was addressed and resolved by a recent work[40], but once again some of the used data structures cannot be used to achieve one of my goals, i.e. to encrypt the sequences of different individuals with distinct encryption key.

Once described the state of the art, hereafter I introduce my contribution: the first encrypted self-index based on referential Lempel-Ziv compression, designed to be the core of a multi-user database engine. This type of engine must allow pattern search in a set of such indexes, permitting a logged user to retrieve results only from the sequences for which he/she has been explicitly authorized.

3.1 Definitions and background

3.1.1 Relative Lempel-Ziv factorization

Let a text S be a string of symbols over a finite alphabet A.

All the lossless compression algorithms known in literature are aimed to exploit the repetitions in a text S in order to to obtain more compact representations of it. The Lempel-Ziv *dictionary-based* family is not an exception to this rule, as its basic idea is to replace repetitions with references to previous occurrences of them.

Both *LZ77* and *LZ78*, the most used family members, consist in rules for parsing a text S into a sequence of factors, which are substrings occurring in it before the current scanning position. Their main differences are the the dictionary definition rules: while *LZ78* builds explicitly the dictionary, *LZ77* is a so called "sliding window compression algorithm", as a factor can only be a reference to a substring found till w positions earlier in the text, that is within a *sliding window* of width w. In both cases the dictionary consists of substrings of the same text S.

Let us suppose now that S is a genomic sequence of an individual belonging to a given species, for which a reference sequence R is known: S is very similar to R, presenting only a few number of mutations, deletions and insertions, often in a percentage not greater than 1%.

Let us suppose that we want to express S as a sequence of phrases. Rather than search for these phrases within a dictionary of previously occurred substrings of S, it would be better to search for them in R; in fact, a given portion of S is more similar to the corresponding portion of the reference sequence than to a previously seen substring of S. This is the basic idea of the so-called Relative Lempel-Ziv factorization.

Definition 3.1. Relative Lempel-Ziv factorization

Let S and R be an individual and a reference sequence, respectively. The Relative Lempel-Ziv factorization of S with respect to R, denoted as $LZ(S|R)$, is a sequence of fn factors

$$z_0 \cdots z_{fn-1}$$

Each factor can be seen as the concatenation of a *referential part*, that is a substring of R, and a mismatch character; so the generic j^{th} factor (for $j = 0, \ldots, fn-1$) is representable with a triple $\langle p_j, l_j, mc_j \rangle$, where:

- p_j is the position of its referential part in R;
- l_j is the length of its referential part;
- mc_j is the mismatch character.

For example, let

$$R = ACTAACCGTACATGA$$

be the reference sequence and

$$S = ACTACACCCTACATGQCTAA$$

be the individual sequence to factorize. Both the sequences are strings over the IUPAC extended alphabet, which I will call from now on Σ_{DNA}. The first one generally contains only N (aNy) characters in addition to those representing the bases A, C, G, T, whilst the second could be, for example, the consensus sequence for a diploid individual and could contain all Σ_{DNA} characters.

The Relative Lempel-Ziv factorization of S with respect to R is:

$$LZ(S|R) = \boxed{ACTA|C}\boxed{ACC|C}\boxed{TACATG|Q}\boxed{CTA|A} =$$
$$= \langle 1, 4, C \rangle \langle 5, 3, C \rangle \langle 9, 6, Q \rangle \langle 3, 3, A \rangle$$

To understand how this representation can be used to compress the string S, just think that in an actual scenario the referential parts consist of hundreds or even thousands of bases.

3.1.2 B+ trees

B Trees[42] and their B+ variant are the most used index structures in Relational Database Management Systems. Substantially, a B Tree is a N-ary search tree providing efficient algorithms aimed to mantain a sorted set of numbers, or keys and to do fast search operations on it; moreover, like every N-ary search tree, a B Tree consists of nodes organized as a hierarchy starting from a root node and descending from each node to its children until reaching the last level of nodes, called *leafs*.

Every node contains a certain number n of keys, to each of which is associated a value; furthermore, each node contains a set of $n+1$ pointers to its children.

The above described data structure supports efficient updates and exact match queries, which find the value/s associated to a given key; it also permits to do efficiently an operation known in literature as *range query*, which finds all the values associated to keys falling within a range $[l, r]$. In RDBMS B Tree based indexes, by example, node keys correspond to the values of a table column C and node values uniquely identify a row of that table T: so the index permits fast access to the rows of T satisfying the condition $l \leq C \leq r$.

More in detail, defined the order m of a B Tree as the minimum number of keys a non-leaf node may hold, a B Tree of order m has the following properties:

- all leaf nodes are in the same level, i.e. the tree is *balanced*;
- every node contains at most $n \leq 2m$ keys (and associated values);
- every node, except the root node, contains at least m keys (and associated values);
- every node stores values in increasing key order;
- a non-leaf node containing n keys contains also $n + 1$ pointers to node children; the i^{th} key act as separation value between the subtrees corresponding to the i^{th} and the $(i + 1)^{th}$ pointer;
- if the root node is not a leaf node, it has at least 2 children.

In B+ Trees, the B Trees variant I took as a model for the ER-index auxiliary data structures, only leaf nodes store values, while root and internal nodes only store search keys; therefore in such a tree it is possibile to distinguish two kinds of nodes:

- *index nodes* (root and internal nodes), containing only search keys and pointers to children nodes;
- *leaf nodes*, containing only keys and values.

Tipically B Trees nodes are stored on secondary storage as fixed size disk pages, whose size is a multiple of the hosting file system page size; the pages are retrieved from a cache when needed, modified and transfered again to the cache if it is necessary to do an update on their content.

In order to increase pages capacity, i.e. the number of keys and pointers stored in each page, several compression schemes can been applied[23]. They are all based on the assumption that the keys stored in a node page are very close each other. An example is the *Invariable Coding* method that I used to compress my auxiliary data structures. It is very simple, in that:

1. it stores the node's first key, which is the smallest one;
2. then it stores all the differences between each key and the first one, on a number of bits sufficient to express the difference between the last and the first key.

3.1.3 A bit of cryptography

In order to fully understand the *ER-index* data structures and algorithms, some notions about *cryptography* are also needed.

Basically, let $p \in P$ be a text to encrypt, also said plaintext, and $c \in C$ the encryption result, also said ciphertext. There are two great categories of encryption methods: symmetric and asymmetric.

In *symmetric ciphers*, also called *secret-key ciphers*, the same secret key $k \in K$ is used both to encrypt a plaintext and to decrypt a ciphertext and it is necessary to define two separate functions:

- an enciphering function $E : (p, k) \in P \times K \rightarrow c \in C$;
- a deciphering function $D : (c, k) \in C \times K \rightarrow p \in P$.

The first one transforms a plaintext in a ciphertext and the second one does the reverse transformation, using the same secret key. Symmetric ciphers can be in turn divided into two categories:

- *block ciphers* split the plaintext in blocks, encrypting each block at a time with the same key, so that the encryption of each plaintext bit depends on other plaintext bits contained in that block;
- *stream ciphers* encrypt bits individually, first generating a keystream from the secret key and subsequently adding (modulo 2) the i^{th} plaintext bit to the i^{th} keystream bit.

The addition modulo 2 is a very simple and fast operation, as it is equivalent to an XOR, and so stream ciphers are generally faster than block ciphers; moreover, some stream ciphers are hardware-optimized, that is designed to work even more efficiently on certain hardware architectures.

In particular, Salsa20[6] is one of the ciphers selected as part of the eSTREAM[2] portfolio of stream ciphers [2], and has been designed for high performance in software implementations on Intel platforms; it produces a keystream long 2^{70} bytes starting from a 32 bytes (256 bits) key and a 8 bytes (64 bits) *nonce* (unique message number); it subsequently encrypts a plaintext, seen as sequence of b bytes, simply by XOR-ing the plaintext with the first b bytes of the stream and discarding the rest of the stream.

The aforementioned *nonce* is an arbitrary number used to ensure that several plaintexts or, in our scenario, several plaintext portions are encrypted with different keystreams.

Asymmetric ciphers, also called *public-key ciphers*, use instead a couple of keys consisting of a private key k_{pvt} and a public key k_{pub}, so that a message encrypted with any of them can be deciphered only using the other one. To achieve confidentiality of messages between several individuals, a couple (k_{pvt}, k_{pub}) has to be generated for each of them: the public key must be available for everyone, while the private one must remain known only by the owner.

If an individual A wants to send B a confidential message M, A will encrypt M with the public key of B, so that only B can decrypt it using his private key.

3.2 The ER-index

The Encrypted Referential Index (ER-index) is an encrypted full-text index designed to handle a collection of highly similar genomic sequences. It consists substantially in two major components:

- a set of relative Lempel-Ziv factorizations, one for each sequence of the collection;
- a set of auxiliary data structures, inspired to *B+ Trees*.

Both the factorizations and the auxiliary data structures are designed to permit efficient pattern searching, while allowing the user to search only on that sequences to which he/she was granted access.

In particular, to apply encryption mantaining good search and compression performance, I split each factorization in a series of fixed-length blocks of factors, so that each of them contains the same number of factors; then I processed each block, producing a compact representation of it, whose size depends on the compressibility of the information

describing its factors; finally, I encrypted the obtained variable size blocks independently from each other using the *Salsa20* cipher.

To design the auxiliary data structures I was inspired instead to *B+ Trees* indexes, the most used index data structures in the field of relational databases; this choice derived from three considerations:

- they support range queries;
- they are by design stored in blocks, the so-called tree nodes, that I could compress and encrypt independently from each other;
- it could be quite simple to bring the ER-index inside a relational database, to integrate genomic information with personal and clinical data.

The encryption model design for tree nodes required however a greater effort. The major issue I had to address was the coexistence within the same node of information regarding different individuals, each of which had to be enciphered using a different encryption key. The result was an index structure over all the individual factorizations, designed to retrieve through a unique range query the factors falling in a range, limited to the set of individuals to which the user was granted access.

Another design choice covered the data structure to handle the reference sequences. My factorization and search algorithms, as other methods known in literature to approach the same problem, needed a set of data structures enabling to find all the factors ending with certain pattern substrings, as well as the factors starting with other pattern substrings. Differently from the authors of [40], who used Compressed Suffix Trees, I chose to store each reference sequence R as a couple of FM-indexes, the first build on R and the second on its reverse R_{rev}; this allowed me to address the above needs obtaining good search performance while saving disk space.

Below I describe the ER-index in detail.

Let $\{S_1, \ldots, S_l\}$ be a collection of sequences corresponding to l different individuals and R a reference sequence. Let $f_i = LZ(S_i|R)$ be the Relative LZ-factorization of the individual sequence S_i with respect to R; let $BL_{i,1}, \ldots, BL_{i,bn(i)}$ be the sequence of bs-length blocks factors get from $LZ(S_i|R)$, where $bn(i)$ denotes the obtained number of blocks and each block contains exactly bs factors.

Let moreover $S20 = S20(plaintext, key, nonce)$ denote the enciphering function of the Salsa20 cipher, as previously described, and k_i be the secret encryption key used to encode the genome of the i^{th} individual.

ER-index encrypts each of the aforementioned blocks with Salsa20, using the block number as nonce, so that the encrypted representation of the i^{th} factorization, denoted as $E(f_i, k_i)$, could be build as a concatenation of encrypted blocks, as follows:

$$E(f_i, k_i) = S20(BL_{i,1}, k_i, 1) \cdots S20(BL_{i,bn(i)}, k_i, bn(i))$$

Moreover, in order to speed up search operations, I needed a data structure enabling fast access to factors of all the individual factorizations by a search key related to the referential parts position in the reference genome.

I designed for this purpose the *Encrypted B+ tree* (abbreviated as EB+ Tree). An ordinary B+ tree simply would store for each key k_j a list of all the values

$$v_{j,0}, \ldots, v_{j,h_j}$$

associated to that key: this would be my case if I had only a single factorization and the values corresponded to factor indexes, each of which uniquely identifies a factor in that factorization.

$$\boxed{\cdots \| kj | v_{j0}, v_{j1}, \cdots, v_{jh_j} \| \cdots}$$

Figure 3.1: Structure of an ordinary B+ Tree node
h_j is the number of values associated to the key k_j and $v_{j0}, v_{j1}, \cdots, v_{jh_j}$ is the list of those values.

My aim instead was to build a B+ Tree indexing a collection of individual genomic sequence factorizations; so, each value (factor index) had to be associated with the individual identifier that uniquely selects the right factorization; moreover, the value had to be ciphered with the encryption key associated to that individual, as well as the entire factorization to which it refers.

$$\boxed{\cdots \| kj | \langle i_{j0}, v_{j0} \rangle, \langle i_{j1}, v_{j1} \rangle, \cdots, \langle i_{jh_j}, v_{jh_j} \rangle \| \cdots}$$

Figure 3.2: Structure of an EB+ Tree node
h_j is the number of values associated to the key k_j and $v_{j0}, v_{j1}, \cdots, v_{jh_j}$, while $i_{j0}, i_{j1}, \cdots, i_{jh_j}$ are the related individual identifiers.

In EB+ trees I applied the *Invariable Coding* compression method both to keys and values, but in a different way than [42]. The authors of that work applied compression to arrange more values into fixed size node pages; instead, I used it in order to obtain smaller variable length nodes and to maximize in this way the overall index size.

After the compression phase, I encrypted the values (factor indexes) contained in each leaf with *Salsa20*, virtually considering the list of values related to each individual as a unique bytestream. The encryption method works for each leaf as follows:

- it initializes an encryption context for each individual, computing a *Salsa20* keystream on the basis of that individual's encryption key;
- it ciphers the leaf data not related to a specific individual with a *Salsa20* keystream obtained using a system encryption key, corresponding to a system encryption context;
- every time it meets a factor index related to an individual, it switchs to that individual's encryption context, encrypts the value and switches again to the system encryption context.

Definition 3.2. Encrypted Referential index
An **E**ncrypted **R**eferential index (ER-index) build over a collection of individual sequences $\{S_1, \ldots, S_l\}$ with respect to a reference sequence R and a set of individual keys $\{k_1, \ldots, k_l\}$ is a self-index consisting of:

- all the encrypted relative Lempel Ziv factorizations of the individual sequences towards a reference sequence: $\{E(f_1, k_1), \ldots, E(f_l, k_l)\}$;
- a set of three EB+ Trees build over the factors of all the factorizations, whose search keys are respectively:

1. a suffix array index corresponding to a R_{rev} suffix prefixing the reverse of the factor referential part;
2. a suffix array index corresponding to a R suffix prefixing the factor referential part;
3. the position of the referential part in the reference R.

and whose values are couples $\langle i, v \rangle$, where i identify an individual and v a factor of the related genomic sequence; v is ciphered with the encryption key associated to the individual i.

3.2.1 Factorization algorithm

My factorization algorithm slightly differs from that proposed in [26] and [40], as the ER-index uses a couple of FM-indexes to represent the reference sequence R and its reverse R_{rev}. The j^{th} factor is always representable as a triple $\langle sai\_rev\_start_j, l_j, mc_j \rangle$ of numbers, but in my index they have a different semantic:

- $sai\_rev\_start_j$ is the R_{rev} suffix array index from which to start the backward scan of R_{rev} in order to obtain the factor;
- l_j is the length of the entire factor, comprehensive of the mismatch character;
- mc_j is the mismatch character.

In order to speed-up pattern search, the algorithm retrieves also three auxiliary data, storing them as search keys in as many encrypted B+ Trees:

- sai_rev_j, a suffix array index corresponding to a R_{rev} suffix prefixing the reverse of the factor referential part;
- sai_j, a suffix array index corresponding to a R suffix prefixing the factor referential part;
- tp_j, the position of the referential part in the reference R.

The algorithm 5 uses four data structures related to the reference sequence:

- the FM-index of the reference R, denoted as FM;
- the FM-index of the reversed reference R_{rev}, denoted as FM_{rev};
- a correspondence table between the suffix array of R_{rev} and the suffix array of R, denoted with $R2F$, which permits to move from a suffix of R_{rev} to the R suffix starting from the same character;
- the reverse correspondence table, denoted with $F2R$, which permits to move from a suffix of R to the R_{rev} suffix starting from the same character.

In short, the algorithm aims to find a series of factors whose concatenation gives as result the sequence S. From an overall point of view, it scans S from left to right and at each step it tries to factorize the suffix S_i, searching the maximum length referential factor starting from i: at this purpose it scans the BWT of the reverse reference R_{rev} through FM_{rev}, starting from $S[i]$ and proceeding backward on R_{rev} until a mismatch is found (further details are given within the pseudo-code).

The backward search gives as result the R_{rev} suffix array range containing the suffixes prefixing the reverse of S_p: the algorithm choose the first among them, as they are all logically equivalent for its purposes.

The next step consists in retrieving the auxiliary information related to the just found factor.

I do not report the $getTextPosition$ and $backwardStep$ functions implementation details in the pseudo-code, as they match exactly the canonical FM-index implementations.

Algorithm 5 Factorization algorithm

```
1: function FACTORIZE(S, FM_rev, FM, R2F, F2R)
2:     j ← 0                                              ▷ Current factor index
3:     l_max ← 0;                                         ▷ Maximum factor length
4:     len ← length(S);
5:     i ← 0;
6:     while i < len do
7:         ▷ Retrieve the next factor
8:         numberOfN ← 0;          ▷ Number of N characters found in the next factor referential part
9:         nrc ← S[i];             ▷ Current character, not yet remapped in the FM index compact alphabet
10:        if nrc = N then
11:            numberOfN ← numberOfN + 1;
12:        end if
13:        l ← 1                                          ▷ Current length of the next factor referential part
14:        if i < len − 1 AND isInReference(FM_rev, nrc) then
15:            lastNrc ← nrc;
16:            ▷ Start a backward search on the reverse reference index
17:            c ← remap(FM_rev, nrc)];                   ▷ Remap current character
18:            sp ← C(FM_rev, c);
19:            ep ← C(FM_rev, c + 1) − 1;
20:            backwardStepSuccessful ← true;
21:            ▷ The backward search stops when
22:            ▷ - the referential part includes the last but one character of S or
23:            ▷ - the next character is not in the reference sequence or
24:            ▷ - the last backward step was not successful or
25:            ▷ - the next character is N and the last was different from N or
26:            ▷ - the next is different from N and the last was N
27:            while i + l < len − 1 AND
                   isInReference(FM_rev, nrc ← S[i + l]) AND
                   backwardStepSuccessful AND
                   (lastNrc ≠ N AND nrc ≠ N OR lastNrc = N AND nrc = N) do
28:                if nrc = N then
29:                    numberOfN ← numberOfN + 1;
30:                end if
31:                c ← remap(FM_rev, nrc)];
32:                trySp ← C(FM_rev, c) + Occ(FM_rev, EOF_shift(FM_rev, sp − 1), c);
33:                tryEp ← C(FM_rev, c) + Occ(FM_rev, EOF_shift(FM_rev, ep), c) − 1;
34:                if trySp <= tryEp then
35:                    sp ← trySp;
36:                    ep ← tryEp;
37:                    l ← l + 1;
38:                    backwardStepSuccessful ← true;
39:                else
40:                    backwardStepSuccessful ← false;
41:                end if
42:                lastNrc ← nrc;
43:            end while
```

Algorithm 5 Factorization algorithm (continued)

```
44:            sai_rev_pref ← sp;
45:            mc ← S[i + l]
46:            ▷ Find sai_rev_start, sai_pref and tp, as follows:
47:            ▷ 1) Find the suffix array index sai of R corresponding to the suffix array index sai_rev_pref of
    R_rev
48:            sai = R2F(sai_rev_pref);
49:            ▷ 2) Do l − 1 backward steps on the straight index, in order to find sai_pref
50:            for i ← 1 To l − 1 do
51:                sai ← backwardStep(FM, sai);
52:            end for
53:            sai_pref ← sai;
54:            ▷ 3) Find the position tp of R corresponding to sai_pref using the FM index marked rows
55:            tp = getTextPosition(FM, sai_pref)
56:            ▷ 4) Do another backward step on the straight index, in order to find sai_rev_start
57:            sai_rev_start ← backwardStep(FM, sai_pref);
58:            ▷ 5) Store the retrieved factor in the factors array
59:            factors[j] ← ⟨sai_rev_start, l, mc⟩;
60:        end if
61:        i ← i + 1
62:    end while
63: end function
```

Return to the factorization example presented in the last section, where:

- $S =$ `CTACACCCTACATGQCTAA` is the sequence to be factorized;
- $R =$ `ACTAACCGTACATGA` is a reference string;
- $R_{rev} =$ `AGTACATGCCAATCA` is the reverse reference string.

Remember that FM-index is a compact and high speed pattern search data structure build over the BWT of a text T, and BWT is computed sorting the rotations matrix of T; therefore, in order to fully understand the details of the factorization algorithm in a simple example, it is needed to visualize the rotations matrices, the BWT and the suffix arrays of R and R_{rev}, shown in tables 3.2 and 3.1.

A # character, preceding each Σ_{DNA} character in lexicographical order, has been appended to both the strings before the BWT computation. The # character does not belong to the strings and so the FM-index does not store the BWT position occupied by #, but it stores in its header the virtual position of that character in order to signal the end of text and to skip it during the backward search operation; note too that the rotation starting with # does not contribute to the suffix array, as # does not really belong to the strings.

Moreover, obviously the suffixes of R and R_{rev} are related, as for 0-based strings the suffix starting from the i^{th} place of R corresponds to the suffix of R_{rev} starting from position $n - i - 1$. So I computed once and for all a correspondence table between the two suffix arrays, in order to speed-up the factorization operations, as I will show afterwards. The correspondence table $R2F$, shown in table 3.3, permits to move from a suffix of R_{rev} to the R suffix starting from the same character.

Suppose the algorithm has just started and it has to find the first factor: the maximal referential factor is `CTA-C`, whose referential part `CTA` corresponds to the R_{rev} suffix array range `[4,4]`. This implies that $sai\_rev\_pref = 4$, and from 3.1 it is simple to see that the R_{rev} suffix corresponding to the index 4 prefixes `ATC`, which is the reverse of the factor's referential part. The algorithm proceeds:

1. moving to the corresponding suffix of R: this happens simply computing `sai=R2F(4)=1`, as can be evicted from 3.3;

2. doing `l-1=2` backward steps on $BWT(R)$, in order to compute the `sai_pref=9` index of a R suffix prefixing the factor's referential part;

3. doing another backward step on BWT in order to reach the `sai_start=4` suffix array index, and finally computing `sai_rev_start=F2R(sai_start)=F2R(4)=0`: it is simple to see from table 3.1 that, starting from the index `sai_rev_start=0` and doing two backward steps, it is possible to obtain all the referential part characters, from the last one to the first one;

4. computing the text position corresponding to the R suffix array index `sai_pref=9`,

Summarizing, the algorithm has retrieved until now the first factor $\langle$`0,4,C`$\rangle$ and the auxiliary data `sai_rev_pref=4`, `sai_pref=9` and `tp=1` to use as search keys within the three aforementioned B+ trees.

BWT index	SA index	SA	rotation	BWT
0			#AGTACATGCCAATCA	A
1	0	14	A#AGTACATGCCAATC	C
2	1	10	AATCA#AGTACATGCC	C
3	2	3	ACATGCCAATCA#AGT	T
	3	0	AGTACATGCCAATCA#	
4	4	11	ATCA#AGTACATGCCA	A
5	5	5	ATGCCAATCA#AGTAC	C
6	6	13	CA#AGTACATGCCAAT	T
7	7	9	CAATCA#AGTACATGC	C
8	8	4	CATGCCAATCA#AGTA	A
9	9	8	CCAATCA#AGTACATG	G
10	10	7	GCCAATCA#AGTACAT	T
11	11	1	GTACATGCCAATCA#A	A
12	12	2	TACATGCCAATCA#AG	G
13	13	12	TCA#AGTACATGCCAA	A
14	14	6	TGCCAATCA#AGTACA	A

Table 3.1: BWT and suffix array of R_{rev}

BWT index	SA index	SA	rotation	BWT
0			#ACTAACCGTACATGA	A
1	0	14	A#ACTAACCGTACATG	G
2	1	3	AACCGTACATGA#ACT	T
3	2	9	ACATGA#ACTAACCGT	T
4	3	4	ACCGTACATGA#ACTA	A
	4	0	ACTAACCGTACATGA#	
5	5	11	ATGA#ACTAACCGTAC	C
6	6	10	CATGA#ACTAACCGTA	A
7	7	5	CCGTACATGA#ACTAA	A
8	8	6	CGTACATGA#ACTAAC	C
9	9	1	CTAACCGTACATGA#A	A
10	10	13	GA#ACTAACCGTACAT	T
11	11	7	GTACATGA#ACTAACC	C
12	12	2	TAACCGTACATGA#AC	C
13	13	8	TACATGA#ACTAACCG	G
14	14	12	TGA#ACTAACCGTACA	A

Table 3.2: BWT and suffix array of R

$SA_{R_{rev}}$ index	SA_R index
0	4
1	3
2	5
3	0
4	1
5	2
6	9
7	7
8	6
9	8
10	11
11	10
12	14
13	12
14	13

Table 3.3: Suffix arrays correspondence table

3.2.2 Pattern Search

ER-index supports exact pattern matching through the algorithm 6.

Before describing the algorithm details, it is appropriate to make some considerations. A pattern search operation on a relative Lempel-Ziv factorization can retrieve two types of occurrences:

- *internal occurrences*, which are completely contained in a factor's referential part;
- *external occurrences*, also known in literature as *overlapping occurrences*, which have at least a character outside of a factor's referential part: these occurrences can span two or more factors or stop on a factor's mismatch character.

A solution to find external occurrences on a $LZ78$ factorization has been proposed in [30] and it is based on *trie* data structures, belonging to the tree search data structures category and also known as *prefix trees*.

A *trie* built on a collection of strings stores all the collection items so that the children of a given node share a common prefix; therefore such a data structure allows searching quickly for all the collection items prefixed by an assigned string.

The proposed method uses two tries: given a text T, the first trie, named $LZTrie$, stores the collection of the $LZ78$ factors of T, whilst the second one, named $RevTrie$, stores the reversed factors. It splits the searched pattern P in all possible ways and, for each split point, it searches for the reverse left side prefix in $RevTrie$ and the right side prefix in $LZTrie$, obtaining respectively two set of factors:

- the first set contains all the factors ending with the pattern's left side;
- the second one contains all the factors starting with the pattern's right side.

Ultimately, the proposed algorithm joins the two sets in order to obtain couples of consecutive factors. The first element of each couple ends with the pattern's left side, while the second element starts with the right side.

This approach could also be used for relative Lempel-Ziv factorizations, but it has a big issue: trie is a very expensive data structure, because it requires too much disk space, and so it is infeasible to build a couple of tries for each individual factorization. In order to overcome this issue, my external occurrences search algorithm keeps the main idea to split the pattern in all possible ways, but uses very different and less expensive data structures:

- the same FM_{rev} and FM indexes of the above presented factorization algorithm, used respectively to search for the maximal prefix of the reversed left side in the reverse reference string R_{rev} and for the maximal right side prefix in the reference string R;
- a couple of $EB+$ trees:
  1. the first one allows to retrieve all the individuals factors that end with the maximal prefix of the reversed left side;
  2. the second one allows to retrieve all the individuals factors that start with the maximal right side suffix.

As regards the internal occurrences, the approach proposed in [30] is not applicable to relative Lempel Ziv factorization, because it is based on a specific $LZ78$ property: each factor can be seen as the concatenation of a previous factor and an additional character.

Therefore, I designed an original algorithm to retrieve the internal occurrences using once again the FM index built on the R reference sequence, together with a third $EB+$ tree, named $posTree$, whose search keys are the starting positions of factors referential parts in R: it allows to retrieve the factors whose referential part starts in a given positions range of the reference sequence.

My algorithm uses also the auxiliary information l_{max}, defined as the maximum length of all factors contained in the ER-index: it is determined during the factorization process and it is stored into the index header.

I started from a simple consideration: an internal occurrence of a pattern P is completely contained into a factor's referential part and so it corresponds exactly to a P occurrence in the reference sequence. Therefore, the first step could consist in retrieving all the pattern's occurrences in the reference sequence, but two not trivial problems remained to be addressed:

- is a reference sequence occurrence really an individual sequence occurrence, that is it exists an individual sequence factor really containing that reference occurrence?
- if yes, how it is possible to retrieve the position of that occurrence in the individual sequence?

I explain now the way I dealt with these problems. Suppose that the suffix array interval $[sp, ep]$ is the result of the pattern search on the reference sequence: the position tp of each interval's element in the reference sequence can be retrieved using the related reference index marked rows (likewise the ordinary FM-index *locate* operation).

Let f be a factor, l the length of its referential part and tpf its starting position in the reference sequence; let moreover m be the pattern length. A reference occurrence located in tp is also an individual sequence occurrence if and only if:

$$\begin{cases} tpf \leq tp \\ tpf + l - 1 \geq tp + m - 1 \end{cases}$$

The first condition ensures that the factor referential part does not start after the first character of the reference occurrence; the second one ensures that the factor referential part does not end before the ending of the reference occurrence. The above conditions can be written as:

$$\begin{cases} tpf \leq tp \\ tpf \geq tp + m - l \end{cases}$$

and finally as:

$$tp + m - l \leq tpf \leq tp \; (1)$$

This means that tpf must fall in a range and then the factors could be retrieved by a range query on $posTree$ if both the range bounds were fixed values; the lower bound however is not a fixed value, as it depends from the specific factor's referential part length l.

In order to overcome this issue I considered that

$$l_{max} \geq l \Rightarrow tp + m - l_{max} \leq tp + m - l$$

So tpf satisfies also the following condition:

$$tp + m - l_{max} \leq tpf \leq tp \; (2)$$

However a range query based on condition (2) also returns factors not satisfying the original condition (1): they can be filtered out keeping only factors complying to the additional condition $tpf \geq tp + m - l$.

The algorithm described above is implemented by the *LocateInternalOccurrences* function, reported into the whole Pattern search algorithm 6 pseudo-code.

Algorithm 6 Pattern search algorithm

```
1: function LOCATE(pattern)
2:     occurrences ← LOCATEEXTERNALOCCURRENCES(pattern) ⋃ LOCATEINTERNALOCCURRENCES(
                                                                              pattern);
3:     SORT(occurrences);                          ▷ sorts each individual occurrence by position
4:     REMOVEDUPLICATES(occurrences);                                  ▷ remove any duplicates
5:     FINDTEXTPOSITIONS(occurrences);          ▷ find each occurrence text position from its factorId and
   FactorOffset
6:     return occurrences;
7: end function

8: function LOCATEEXTERNALOCCS(pattern)
9:     occurrences = []
10:    pl ← LENGTH(pattern);
11:    for sp ← 0 to pl − 1 do
12:        splitPointCharacter ← pattern[sp];
13:        if splitPoint > pl/2 then
14:            ▷ The left side part is longest than the right side part: so the number of factors expected to end
15:            ▷ with the left side part is less than the number of factors expected to start with the right side
16:            ls ← substr(pattern, 0, splitPoint);   ▷ the 2nd argument is the substring starting point, the 3rd
   one is its length
17:            [lsFactors, lsLongestSuffixLength] ← FINDLEFTSIDEFACTORS(ls);
18:            for each distinct individualId in lsFactors do
19:                ▷ Retrieves the factorization from an associative array containing all the individual factorizations
20:                factorization ← factorizations[individualId];
21:                for each factorIndex in GETINDIVIDUALRETRIEVEDFACTORS(lsFactors, individualId) do
22:                    factor ← factorization[factorIndex];
23:                    ▷ Excludes the case in which the left side crosses the starting point of the current factor:
24:                    ▷ an occurrence of this type will be found for a preceding split point
25:                    if factor.length − 1 ≥ length(ls) then
26:                        if factor.letter = splitPointCharacter then
27:                            occurrence.factorIndex ← factorIndex;
28:                            occurrence.factorOffset ← factor.length − 1 − length(ls);
29:                            occurrence.endingFactorIndex ← factorIndex;
30:                            occurrence.endingFactorOffset ← factor.length − 1;
31:                            lsVerifiedLength ← lsLongestSuffixLength;        ▷ Left side verified length
32:                            rsVerifiedLength ← 0;                            ▷ Right side verified length
33:                            if VERIFYPATTERNREMAININGPART(factorization, pattern, sp, lsvl, rsvl,
                                                                              occurrence) then
34:                                ADDOCCURRENCE(occurrence);
35:                            end if
36:                        end if
37:                    end if
38:                end for
39:            end for
40:        else
41:            ▷ The right side part is equal-length or longest than the left side one: so the number of factors
   expected to start
42:            ▷ with the right side part is less than the number of factors expected to end with the left side
43:            rs ← SUBSTR(pattern, splitPoint + 1, pl − splitPoint − 1);
44:            [rsFactors, rsLongestPrefixLength] ← FINDRIGHTSIDEFACTORS(rs);
```

Algorithm 6 Pattern search algorithm (continued)

```
45:             for each distinct individualId in rsFactors do
46:                 ▷ Retrieves the factorization from an associative array containing all the individual factorizations
47:                 factorization ← factorizations[individualId];
48:                 for each factorIndex in GetIndividualRetrievedFactors(rsFactors, individualId) do
49:                     factor ← factorization[factorIndex];
50:                     if rslpLength < factor.length − 1 then
51:                         rsvl ← rslpLength;                                  ▷ Right side verified length
52:                     else
53:                         rsvl ← factor.length − 1;
54:                     end if
55:                     if factorIndex > 0 then
56:                         lsFactor ← factorization[factorIndex − 1];
57:                         if lsFactor.letter = splitPointCharacter then
58:                             occurrence.factorIndex ← factorIndex − 1;
59:                             occurrence.factorOffset ← lsFactor.length − 1;
60:                             occurrence.endingFactorIndex ← factorIndex;
61:                             occurrence.endingFactorOffset ← rsVerifiedLength − 1;
62:                             lsvl ← 0;                                       ▷ Left side verified length
63:                             if VerifyPatternRemainingPart(factorization, pattern, sp, lsvl, rsvl,
                                                                                occurrence) then
64:                                 AddOccurrence(occurrence);
65:                             end if
66:                         end if
67:                     end if
68:                 end for
69:             end for
70:         end if
71:     end for
72:     return occurrences;
73: end function

74: function LocateInternalOccs(pattern)
75:     occurrences = []
76:     ▷ An internal occurrence occurs certainly in the reference sequence
77:     if searchPatternInReferenceIndex(FM, pattern, sp, ep) then
78:         for i ← sp to ep do
79:             m ← length(pattern);
80:             tp ← getPositionInReference(FM, i);
81:             ▷ lmax is the maximum factor length in the index and it is stored into the index header
82:             factors ← getFactorsInRange(posTree, tp + m − lmax, tp);
83:             for each distinct individualId in rsFactors do
84:                 ▷ Retrieves the factorization from an associative array containing all the individual factorizations
85:                 factorization ← factorizations[individualId];
86:                 for each factorIndex in GetIndividualRetrievedFactors(rsFactors, individualId) do
87:                     factor ← factorization[factorIndex];
88:                     tpf ← factor.referentialPartPositionInReference;
89:                     l ← factor.length − 1;                  ▷ length of the factor's refential part
90:                     if tpf >= tp+m-l then
91:                         occurrence.factorIndex ← factorIndex;
92:                         occurrence.factorOffset ← tp − tpf;
93:                         occurrence.endingFactorIndex ← factorIndex;
94:                         occurrence.endingFactorOffset ← occurrence.factorOffset + m − 1;
95:                         lsvl ← 0;                               ▷ Left side verified length
96:                         AddOccurrence(occurrence);
97:                     end if
98:                 end for
99:             end for
100:        end for
101:    end if
102:    return occurrences;
103: end function
```

Algorithm 6 Pattern search algorithm (continued)

```
104: ▷ Verifies if the partial occurrence given as parameter is really a whole pattern occurrence
105: ▷ If successful it returns true and adjusts coherently the occurrence information, otherwise it returns false
106: function VERIFYPATTERNREMAININGPART(factorization, pattern, splitPoint, lsVerifiedLength,
                                         rsVerifiedLength, occurrence)
107:     ▷ I omit the implementation details for reasons of space: the algorithm simply tries to extend the verified part
108:     ▷ of the pattern both on the left and on the right side it compares the yet not verified pattern characters
109:     ▷ with the several factors characters that lie to the left and to the right respect to the pattern verified part
110:     ▷ In order to obtain good performance, the aforementioned extension is made without extracting the full text
111:     ▷ of the involved factors, but scanning that text one character at a time through using once again
112:     ▷ the reverse reference index.
113: end function

114: ▷ Returns a couple whose first element is a list of factors ending with the left side longest suffix
115: ▷ and whose second element is the left side longest suffix length
116: function FINDLEFTSIDEFACTORS(ls)
117:     ▷ Find the left side longest suffix (the longest left side suffix that occurs in the reference string)
118:     l ← FINDLEFTSIDELONGESTSUFFIX(ls);
119:     lsls ← SUBSTR(ls, ls.length − l, l);
120:     if SEARCHPATTERNREVERSEINREFERENCEINDEX(FM_rev, lsls, sp, ep) then     ▷ sp and ep are output parameters
121:         ▷ Find the factors whose suffixArrayPosition is included into the [sp,ep] interval
122:         return [GETFACTORSINRANGE(reverseTree, sp, ep), l];
123:     else
124:         return [[],0];
125:     end if
126: end function

127: ▷ Returns a couple whose first element is a list of factors starting with the right side longest prefix
128: ▷ and whose second element is the right side longest prefix length
129: function FINDRIGHTSIDEFACTORS(rs)
130:     ▷ Find the right side longest prefix (the longest right side prefix that occurs in the reference string)
131:     l ← FINDRIGHTSIDELONGESTPREFIX(ls);
132:     rslp ← SUBSTR(rs, 0, l);
133:     if SEARCHPATTERNINREFERENCEINDEX(FM, rslp, sp, ep) then
134:         ▷ Retrieve the factors whose suffixArrayPosition is included into the [sp,ep] interval
135:         return [GETFACTORSINRANGE(forwardTree, sp, ep), l];
136:     else
137:         return [[],0];
138:     end if
139: end function

140: function FINDRIGHTSIDELONGESTPREFIX(rs)
141:     ▷ I omit the implementation details of this function: it simply scans backward the reverse index,
142:     ▷ starting from the first character of the right side and going on until a mismatch is found.
143:     ▷ Substantially the algorithm is the same used during sequence factorization
144:     ▷ The function returns the right side longest prefix length
145: end function

146: function FINDLEFTSIDELONGESTSUFFIX(ls)
147:     ▷ I omit the implementation details of this function: it simply scans backward the straight index,
148:     ▷ starting from the last character of the left side and going on until a mismatch is found.
149:     ▷ The function returns the left side longest suffix length
150: end function
```

Algorithm 6 Pattern search algorithm (continued)

```
151: function SEARCHPATTERNINREFERENCEINDEX(FM_index, pattern, sp, ep)
152:     ▷ I omit the implementation details of this function: it is a canonical backward search on the given FM-index
153:     ▷ It returns true if the search is successful, false otherwise.
154:     ▷ It returns also the [sp,ep] suffix array range corresponding to the pattern
155: end function

156: function SEARCHPATTERNREVERSEINREFERENCEINDEX(FM_index, pattern, sp, ep)
157:     ▷ I omit the implementation details of this function: it is simply a backward search on the given FM-index, but
158:     ▷ it starts from the first patterns character, goes forward to its second characters, and so on
159:     ▷ It returns true if the search is successful, false otherwise.
160:     ▷ It returns also the [sp,ep] suffix array range corresponding to the pattern
161: end function
```

3.3 Encrypted Referential Database

I detail in this section how several ER-indexes can be joined to form an Encrypted Referential database, i.e. a unique encrypted multi-user database storing genomic information about a set of individuals. I give first of all a formal definition of such a database and on a second stage I explain some implementation details of my ER-database propotype.

Definition 3.3. Encrypted Referential Database Let $= \{i_1, \ldots, i_l\}$ be a set of l individuals and $S = \{s_{j,i_k} \mid j \in \{1, \ldots, 22, X, Y\}, k \in \{1..l\}\}$ be a set of sequences corresponding to their chromosomes. Let moreover $R = \{R_j \mid j \in \{1, \ldots, 22, X, Y\}\}$ be a set of reference sequences for human chromosomes. An **Encrypted Referential Database** (abbreviated **ER-database**) is a tuple

$$D = \{I, K, U, ER, P, RS\}$$

where:

- $I = \{i_1, \ldots, i_l\}$ is a set of individuals;
- $K = \{k_1, \ldots, k_l\}$ is a set of symmetric encryption keys associated biunivocally to those individuals;
- $U = \{u_1, \ldots, u_r\}$ is a set of users, each of which is allowed to access only to sequences related to a subset I' of the individuals;
- ER is a set consisting of an ER-index for each chromosome;
- P is a relation between U and I, which indentifies the individuals to whose genomic information each user is authorized to access;
- R is the above described set of reference sequences.

I implemented the database D so that it could simply be hosted by a file system directory, named the *database root*. The root directory contains a *catalog.xml* file which contains the database catalog: it identifies all the individuals, the users and the reference sequences involved with the database. Moreover, the root directory is articulated in different subdirectories:

- `references`: it contains the reference sequences;

- `indexes`: it contains the ER-indexes;
- `security`: it contains the *key portfolio*, informally the "bunch of keys", of each database user.

The *key portfolio* contains only the symmetric (*Salsa20*) encryption keys related to the individuals whose genomic information the user has been granted access; the portfolio is handled with asymmetric encryption techniques and it is enciphered with the specific user's public key so that only that user could decipher it using his/her private key.

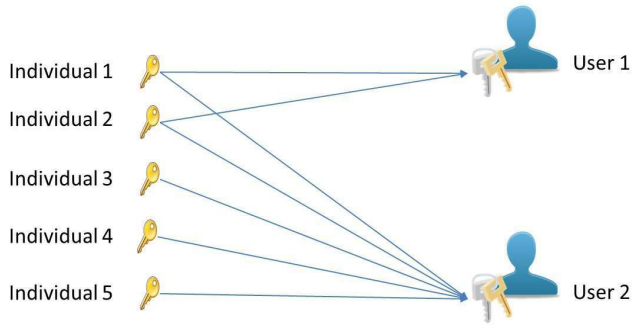


Figure 3.3: ER-database: a simple scenario
User 1 can access only the individuals 1 and 2 genomes

3.4 Experimental results

In order to evaluate the ER-Index performance, I designed and built the prototype of a small database management system: it can handle a test ER-database hosted by a file system directory, as described in the previous section. The prototype was implemented in C++11 language.

Moreover, in order to compare the performance of my prototype with a state of the art software, I used the Sdsl C++ library: it implements some succint data structures usable to construct self-indexes like Compressed Suffix Arrays (CSA) and wavelet tree FM-indexes, but I had to extend the wavelet tree FM-index supplied by that library so that it was able to manage collections of items and to report sequence-relative locations.

3.4.1 Experimental setup

In order to obtain a set of individual sequences on which to assess the prototype performance, I built first of all the diploid consensus sequences related to chromosomes 20 and 11 of 50 individual genomes sequenced whithin the 1000 Genomes Project: to do this I started from the respective BAM files and I used the `samtools mpileup` command, combined with the `bcftools` and `vcfutils` utilities.
On a second stage I did the encryption system set-up, and precisely I:

- randomly generated a 256 bit *SALSA20* encryption key for each of the 50 individuals, using the `openssl rand` command;

- randomly generated a couple of RSA keys for each of the 10 users of the test database, using the `openssl genrsa` and `openssl rsa` commands;
- generated the *key portfolio* of each user, adding to it some individual encryption keys and ciphering it with the user's public key.

3.4.2 Results

I carried out a series of ER-index construction and pattern search tests on 6 different collections:

1. 20_FULL: the chromosome 20 sequences of the aforementioned *1000 Genomes Project* 50 individuals;
2. 20_1MB: the first 1 million bases of those sequences;
3. 20_5MB: the first 5 million bases of those sequences;
4. 11_FULL: the chromosome 11 sequences of the aforementioned *1000 Genomes Project* 50 individuals;
5. 11_1MB: the first 1 million bases of those sequences;
6. 11_5MB: the first 5 million bases of those sequences;

All tests were conducted on a small-size server equipped with an Intel(R) Xeon(R) CPU E5-2697 v2 @ 2.70GHz processor and 180GB of RAM memory.

Figure 3.4 reports ER-index compression ratios, compared with those obtained using the Sdsl library wavelet-tree FM-index: in all cases my index exhibits compression ratios which are an order of magnitude smaller than those offered by Sdsl.

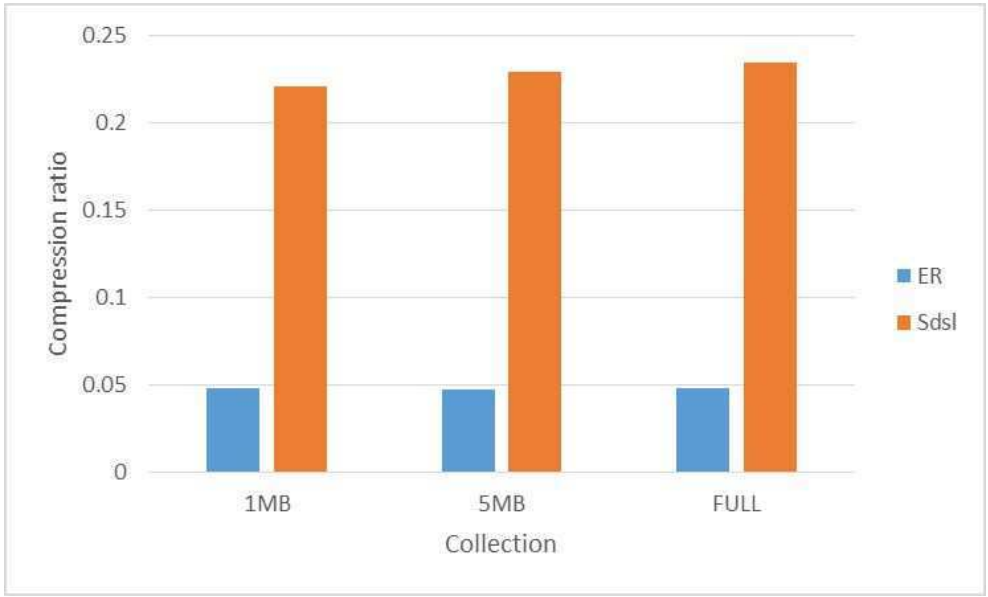
Table 3.4 shows that ER-index construction times are significantly lower than the Sdsl indexes ones, despite the fact that ER-index implements data encryption: this noticeable perfomance has been obtained parallelizing the factorization process so as to exploit the multi-core hyper-threading architecture of modern processors. Note that the test machine has only 24 cores: the speed-up could be greater on a higher-end machine, equipped with more processor cores. The only case in which the Sdsl index exhibits a construction time lower than my index is the 11_1MB collection, but it is not too significant, in that it relates to a very small collection (its total size is only 50 Mbases).

	20_1MB	**20_5MB**	**20_FULL**	**11_1MB**	**11_5MB**	**11_FULL**
ER	11.38	33.74	455.7	23.18	42.79	1005
Sdsl	20.08	132.1	2061	19.33	154.9	5693

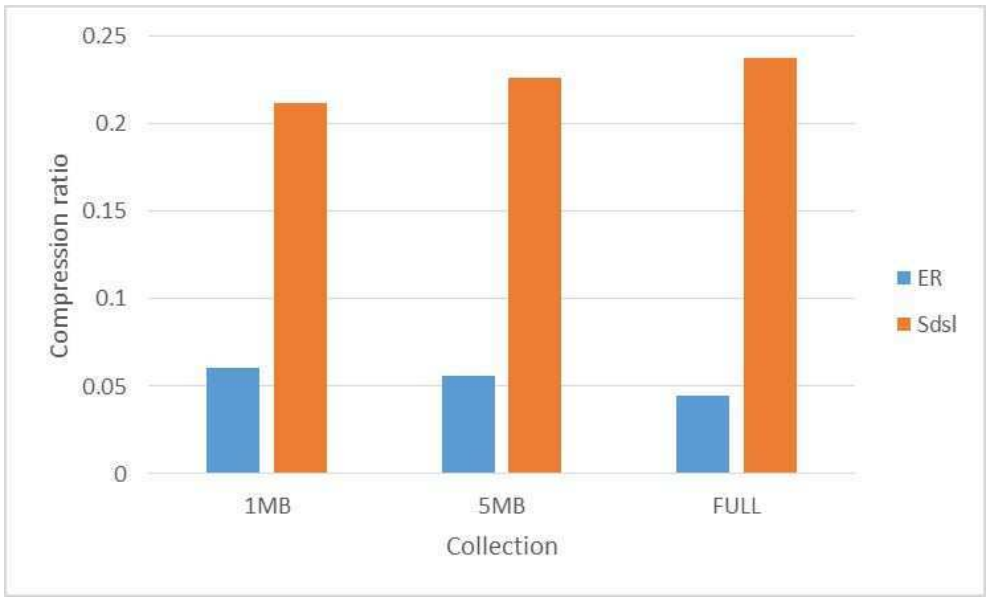
Table 3.4: Construction times (s)

Watching the results shown up to this point, I can affirm that **ER-index outperforms Sdsl FM-index** both in compression ratio and in costruction times: it can store genomic sequence collections in a compressed and encrypted format using less than 5% of the original disk space and its construction time is compatible with the storage of large amounts of data.

I show now some results regarding pattern search performance. For each of the aforementioned collections, I carried out multiple search tests as follows:



(a)



(b)

Figure 3.4: Compression ratios (a) on chromosome 20 collections and (b) on chromosome 11 collections

1. I built the ER-index;
2. for each pattern length $pl \in \{20, 50, 100, 200, 500\}$, I randomly selected a set of 500 patterns from the collection sequences;
3. for each pattern length:
   (a) I opened the index, loading in memory only the index header;
   (b) I searched for any of the 500 patterns, computing for each of them the search time and the search time per occurrence;
   (c) I computed means and medians of the 500 search times and search times per occurrences;
   (d) I closed the index.

Figure 3.5 shows pattern search performance on two quite big sequence collections, 20_FULL and 11_FULL, having respectively a total size of about 2.97 GiB and 6.4 GiB. For both collections pattern search time decreases passing from length 20 to length 50: this occurs because search time is proportional to the number of occurrences and patterns of length 20 have many more occurrences than those of length 50. The length 50 forward, mean time grows with pattern length: it is due to the algorithm adopted for external occurrences and in particular to the greater number of split points checked. However, this behaviour could be noticeably improved using a multi-threaded approach to parallelize the several split points operations, which are naturally independent from each others: this multi-threading strategy has not yet been implemented in my software prototype.

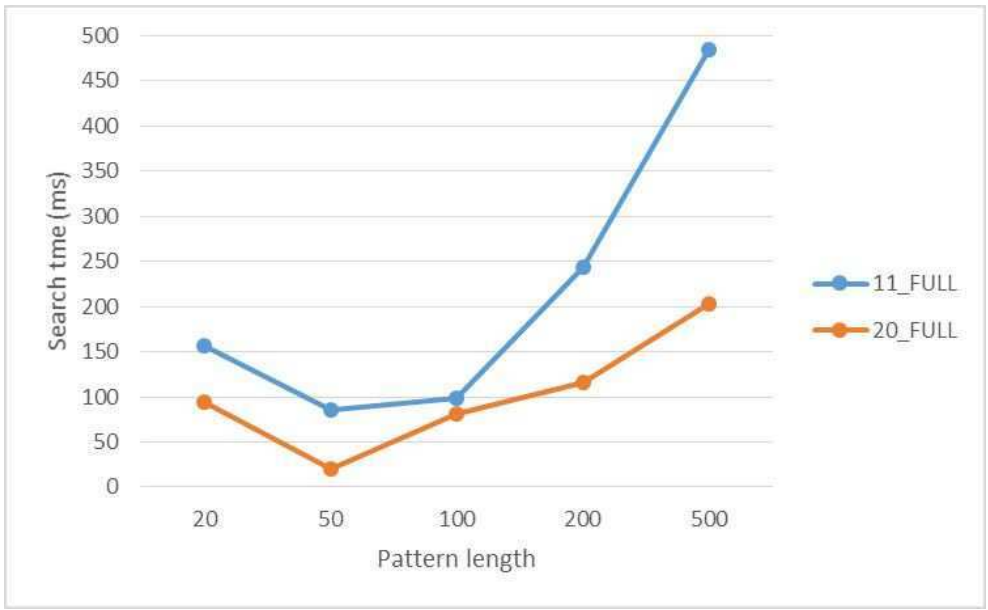
Figure 3.5(b) shows the medians of pattern search times: they are significantly lower than mean times showed in figure 3.5(a), cause of a 10-15% of outliers. These outliers belong to 2 categories:

- the first one consists of really harder to search patterns, for two reasons:
  1. they have a number occurrence much greater than other patterns of the same length;
  2. they span on many short factors and so the left or right side related to some split points are very short strings: this can slow down the external occurrences search algorithm;
- the second category simply consists of the first patterns elaborated after the index file opening, when only a small amount of factorization and EB+ indexes blocks have been loaded in memory.

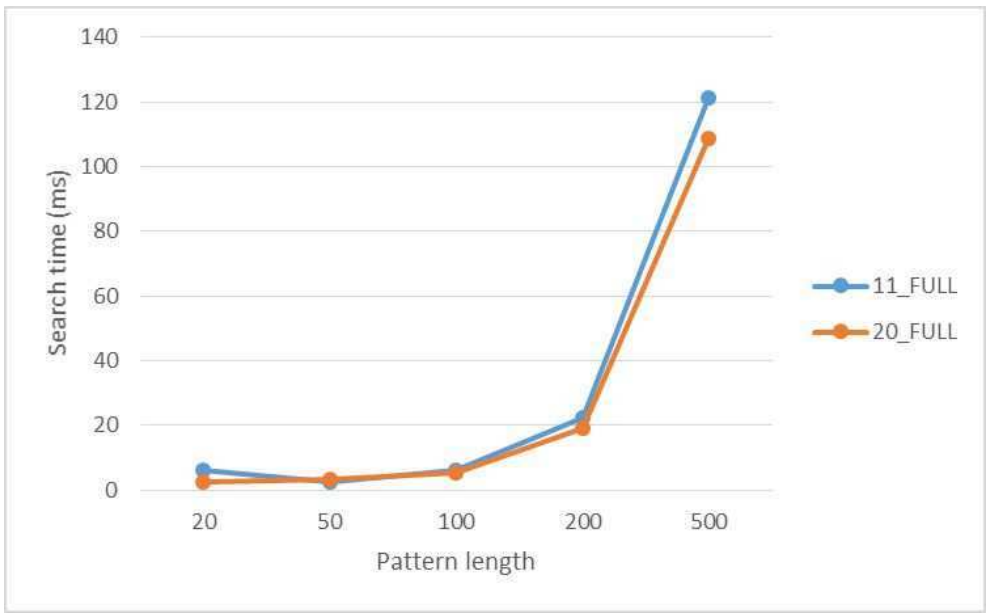
Since pattern search times are proportional to the number of found occurrences, it is appropriate to look at mean and median of search times per occurrence, reported in figure 3.6.

Figure 3.6(a) shows that mean search times per occurrence grow with pattern length, starting from a few milliseconds for length 20 to a maximum of 190.622 ms for length 500 on 11_FULL; moreover mean search times does not exceed some tenth of a second per occurrence on an encrypted and compressed 6.4 GiB collection, and it is really a very good result.

Figure 3.6(b) shows instead that medium case performance is much better than worst case: the median computation indeed exclude the above mentioned outliers and, apart from them, times per occurrence start from some hundredth of a millisecond for small patterns to



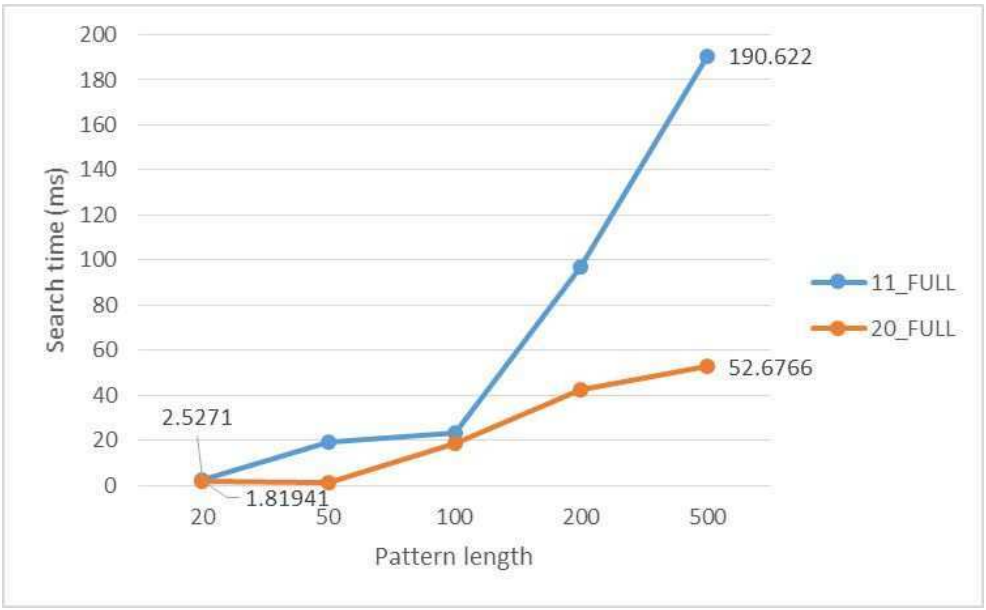
(a)



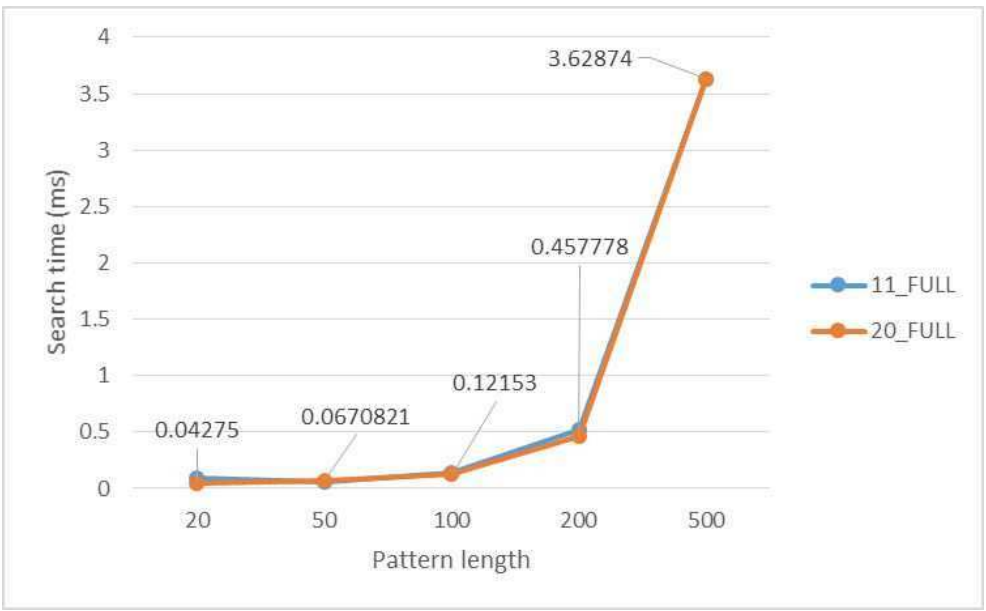
(b)

Figure 3.5: (a) Mean and (b) median of pattern search times on 20_FULL and 11_FULL collections

some millisecond for length 500 patterns. Moreover the curves for the 2.97 GiB and 6.4 GiB collections perfectly overlap, and this attests the scalability of my approach: the ER-index can scale to bigger sequence collections without a significant loss in performance.



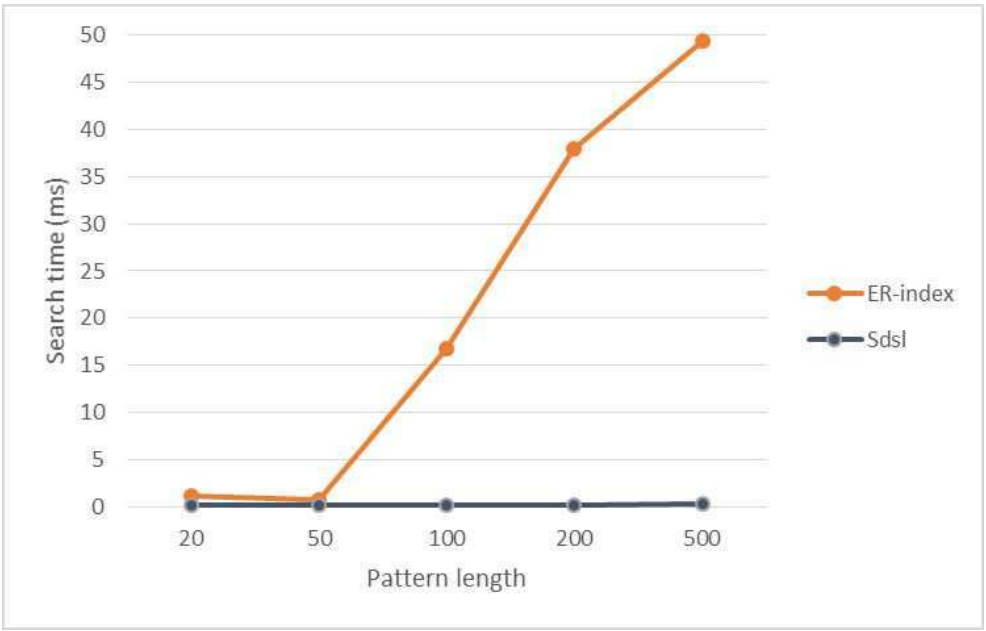
(a)



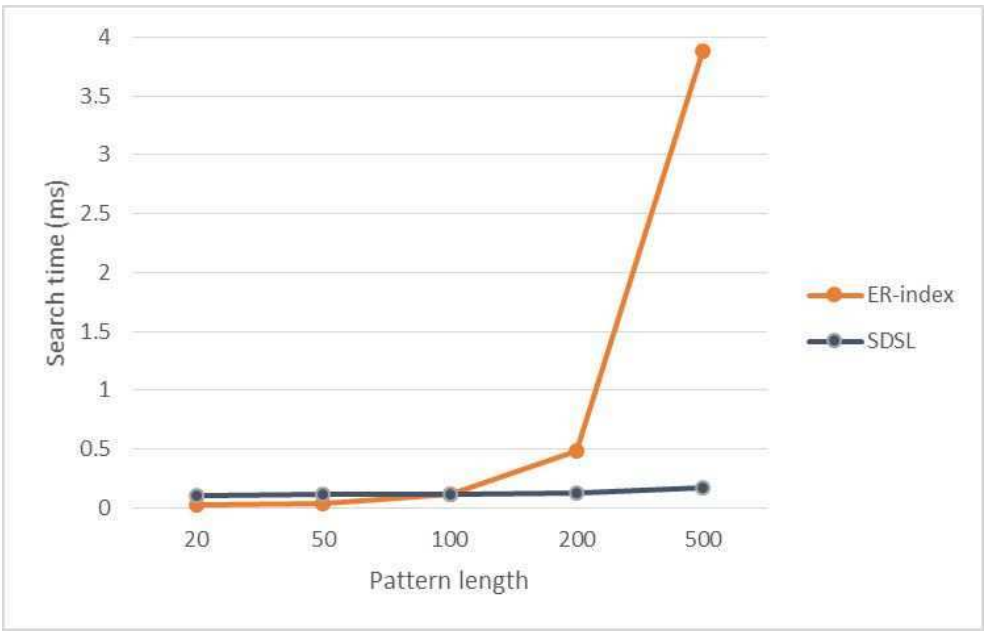
(b)

Figure 3.6: (a) Mean and (b) median of search times per occurrence on 20_FULL and 11_FULL collections

At this point I compare the ER-index search performance with those exhibited by the Sdsl library wavelet tree FM-index.



(a)



(b)

Figure 3.7: (a) Mean and (b) median of ER-index and Sdsl library search times per occurrence on the 20_FULL collection

The Sdsl library index has a big issue: it requires to be entirely loaded in RAM before searching, while my index loads in RAM the only needed blocks *during search operations*. Therefore, in order to properly compare the two index models performance, I run an additional series of tests loading my indexes entirely in memory before starting pattern search operations, although the ER-index has been designed to meet different needs.

Figure 3.7(a) shows that mean times per occurrences are very similar for small patterns, but Sdsl exhibits better performance on large patterns; 3.7(b) shows even that for small patterns, apart from the above mentioned outliers, ER-index performs better, while Sdsl outperforms ER-index on larger patterns. However remember that:

1. Sdsl does not implement encryption;

2. Sdsl has an order of magnitude greater compression ratio respect to ER-index.

Finally, I can conclude that ER-index can store ciphered collections of genomic sequences in less than 1/20 of their original space, indexing them so to have great pattern search performance, which in some cases are better than state of the art third-parties tools.

Morever, the pattern search times could be noticeably slow down for large patterns with the help of multi-threading techniques, as search algorithms are well suited to be parallelized.

The implementation of such a multi-threading search strategy and the introduction of inexact search operations are now under investigation and will be object of future developments.

Conclusions

This thesis gives a new contribution in the area of genomic sequences storage and management: two encrypted and compressed self-index models designed to be the core nucleus of big privacy-preserving genomic databases storage engines.

The first index model, named EFM-index, well suites to collections of lowly similar data that require to be encrypted as a whole, using a single encryption key for each collection: it exhibits optimal compression ratios and pattern search times both on a single sequence and on sequence collections.

The second index model, named ER-index, was designed instead to handle collections of highly similar sequences, like collections of human genomes: it exploits inter-sequence redundancy to obtain better compression ratios and allows to store sequences related to different individuals using different encryption keys within the same index. The ER-index exhibits optimal exact pattern search performance, scales very well with respect to data amount and so it naturally candidates to be used in the context of big multi-user genomic databases.

Acknowledgements

I would like to express my sincere gratitude to my wife Maria and to my daughters Chiara and Manuela for the endless comprehension, love and support they have given to me during my Phd studies.

I am also immensely grateful to my tutors, Prof. Roberto Tagliaferri and Dr. Giovanni Schmid, for their precious guidance and their valuable support on Bioinformatics and Information Security themes.

I would finally thank my Phd course coordinator, Prof. Antonella Leone and her staff, for their precious guidance and support.

Bibliography

[1] David Aucsmith. Tamper resistant software: An implementation. In *Information Hiding*, pages 317–333. Springer, 1996.

[2] S Babbage, C DeCanniere, A Cantenaut, C Cid, H Gilbert, T Johansson, M Parker, B Preneel, V Rijmen, and M Robshaw. The estream portfolio (rev.1), 2008. Available at: `http://www.ecrypt.eu.org/stream/finallist.html`.

[3] Jon L. Bentley and Robert Sedgewick. Fast algorithms for sorting and searching strings. In *Proceedings of the Eighth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '97, pages 360–369, Philadelphia, PA, USA, 1997. Society for Industrial and Applied Mathematics.

[4] Jon Louis Bentley, Daniel D Sleator, Robert E Tarjan, and Victor K Wei. A locally adaptive data compression scheme. *Communications of the ACM*, 29(4):320–330, 1986.

[5] Jon Louis Bentley, Daniel D. Sleator, Robert E. Tarjan, and Victor K. Wei. A locally adaptive data compression scheme. *Commun. ACM*, 29(4):320–330, April 1986.

[6] Daniel J Bernstein. Salsa20 specification, 2005. Available at: `http://www.ecrypt.eu.org/stream/salsa20pf.html`.

[7] Richard C Brackney and Robert H Anderson. Understanding the insider threat. proceedings of a march 2004 workshop. Technical report, DTIC Document, 2004.

[8] Marty C. Brandon, Douglas C. Wallace, and Pierre Baldi. Data structures and compression algorithms for genomic sequence data. *Bioinformatics*, 25(14):1731–1738, July 2009.

[9] Michael Burrows and David J Wheeler. A block-sorting lossless data compression algorithm. *Digital SRC Research Report*, 1994.

[10] Hoi Chang and Mikhail J Atallah. Protecting software code by guards. In *Security and privacy in digital rights management*, pages 160–175. Springer, 2002.

[11] Xiaoxin Chen, Tal Garfinkel, E. Christopher Lewis, Pratap Subrahmanyam, Carl A. Waldspurger, Dan Boneh, Jeffrey Dwoskin, and Dan R.K. Ports. Overshadow: A virtualization-based approach to retrofitting protection in commodity operating systems. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XIII, pages 2–13, New York, NY, USA, 2008. ACM.

[12] Carl Colwill. Human factors in information security: The insider threat–who can you trust these days? *Information security technical report*, 14(4):186–196, 2009.

[13] Oracle Corporation. The java hotspot performance engine architecture. Available at: `http://www.oracle.com/technetwork/java/whitepaper-135217.html`.

[14] McVean et Al. An integrated map of genetic variation from 1,092 human genomes. *Nature*, 491:56–65, 2012.

[15] Paolo Ferragina and Giovanni Manzini. Opportunistic data structures with applications. In *Foundations of Computer Science, 2000. Proceedings. 41st Annual Symposium on*, pages 390–398. IEEE, 2000.

[16] Paolo Ferragina and Gonzalo Navarro. Pizza&chili corpus. Available at: `http://pizzachili.di.unipi.it/`.

[17] 1000 genomes project. Database of human genome markers. Identified through the connection link: `jdbc:mysql://mysql-db.1000genomes.org:4272/homo_sapiens_core_73_37`.

[18] Christiane Honisch, Anu Raghunathan, Charles R Cantor, Bernhard Ø Palsson, and Dirk van den Boom. High-throughput mutation detection underlying adaptive evolution of escherichia coli-k12. *Genome research*, 14(12):2495–2502, 2004.

[19] PKWARE Inc. .zip file format specification, version: 6.3.3, September 2012. available at `www.pkware.com/documents/casestudies/APPNOTE.TXT`.

[20] Guy Joseph Jacobson. *Succinct Static Data Structures*. PhD thesis, Pittsburgh, PA, USA, 1988. AAI8918056.

[21] Juha Kärkkäinen. Fast bwt in small space by blockwise suffix sorting. *Theoretical Computer Science*, 387(3):249–257, 2007.

[22] Sebastian Kreft and Gonzalo Navarro. Self-indexing based on lz77. In *Proceedings of the 22Nd Annual Conference on Combinatorial Pattern Matching*, CPM'11, pages 41–54, Berlin, Heidelberg, 2011. Springer-Verlag.

[23] Filip Krizka, Michal Krátkỳ, and Radim Baca. Benchmarking a b-tree compression method. In *Proceedings of the Conference on Theory and Practice on Information Technologies*, 2009.

[24] M Oguzhan Kulekci. A method to ensure the confidentiality of the compressed data. In *Data Compr, Communications and Processing (CCP), 2011 First International Conference on*, pages 203–209. IEEE, 2011.

[25] M Oguzhan Kulekci. On scrambling the burrows–wheeler transform to provide privacy in lossless compression. *Computers & Security*, 31(1):26–32, 2012.

[26] Shanika Kuruppu, Simon J. Puglisi, and Justin Zobel. Relative lempel-ziv compression of genomes for large-scale storage and retrieval. In *Proceedings of the 17th International Conference on String Processing and Information Retrieval*, SPIRE'10, pages 201–206, Berlin, Heidelberg, 2010. Springer-Verlag.

[27] Shanika Kuruppu, Simon J. Puglisi, and Justin Zobel. Optimized relative lempel-ziv compression of genomes. In *Proceedings of the Thirty-Fourth Australasian Computer Science Conference - Volume 113*, ACSC '11, pages 91–98, Darlinghurst, Australia, Australia, 2011. Australian Computer Society, Inc.

[28] Ben Langmead and Steven L Salzberg. Fast gapped-read alignment with bowtie 2. *Nature methods*, 9(4):357–359, 2012.

[29] Alfred J Menezes, Paul C Van Oorschot, and Scott A Vanstone. *Handbook of applied cryptography*. CRC press, 2010.

[30] Gonzalo Navarro. Indexing text using the ziv-lempel trie. In *Proceedings of the 9th International Symposium on String Processing and Information Retrieval*, SPIRE 2002, pages 325–336, London, UK, UK, 2002. Springer-Verlag.

[31] US Department of Commerce/NIST. Federal information processing standards publication 197: Advanced encryption standard (aes), November 2001. Available at `http://http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf`.

[32] Igor Pavlov. 7-zip, January 2013. Available at: `www.pkware.com/documents/casestudies/APPNOTE.TXT`.

[33] Boris Yakovlevich Ryabko. Data compression by means of a "book stack". *Problemy Peredachi Informatsii*, 16(4):16–21, 1980.

[34] Ravi S Sandhu. Role-based access control. *Advances in computers*, 46:237–286, 1998.

[35] Julian Seward. bzip2 and libbzip2, 1996. available at `www.bzip.org`.

[36] Abraham Silberschatz, Peter B Galvin, and Greg Gagne. *Operating system concepts*, volume 8. Wiley, 2013.

[37] Jouni Sirén, Niko Välimäki, Veli Mäkinen, and Gonzalo Navarro. Run-length compressed indexes are superior for highly repetitive sequence collections. In *Proceedings of the 15th International Symposium on String Processing and Information Retrieval*, SPIRE '08, pages 164–175, Berlin, Heidelberg, 2009. Springer-Verlag.

[38] Martin Stanek. Attacking scrambled burrows-wheeler transform. *IACR Cryptology ePrint Archive*, 2012:149, 2012.

[39] Roberto Togneri and JS Christopher. *Fundamentals of information theory and coding design*. CRC Press, 2003.

[40] Sebastian Wandelt, Johannes Starlinger, Marc Bux, and Ulf Leser. RCSI: Scalable similarity search in thousand(s) of genomes. *Proc. VLDB Endow.*, 6(13):1534–1545, August 2013.

[41] Inc. WinZip Computing. Aes encryption information: Encryption specification ae-1 and ae-2, version 1.04, January 2009. Available at: `www.winzip.com/aes_info.htm`.

[42] Donghui Zhang. *Handbook Of Data Structures And Applications (Chapman & Hall/Crc Computer and Information Science Series.)*, chapter "B Trees". Chapman & Hall/CRC, 2004.