University of Pisa and Scuola Superiore Sant'Anna

Master Thesis

Compressing Dictionaries of Strings

Master Degree in Computer Science and Networking

Author
Lorenzo Landolfi

Supervisor
Rossano Venturini

a.y. 2013/2014

Contents

Chapter 1

Introduction

In the time we are living right now we have seen an unbelievably fast growth of the computer science related technologies. Such a growth has driven one of the most important revolution of mankind, currently and constantly changing the life of all of us. Indeed we are living and *experiencing* such a revolution on our skin. Think about for instance how it changed the way we buy things, we relate to people and have access to information. The e-commerce gives us the possibility to visit the largest store on Earth without the need to physically go in there; the Social Networks changed the way we relate with other people, allowing us to communicate and know persons from all over the world; Search Engines radically changed the way we access any information, allowing all of us to know almost everything cheaply and democratically. In general, we can say that nowadays computers play an important role in everything we do and maybe in everything we are.

Connecting all the computers through the Internet offered the possibility to connect also all the people of the world to each other, speeding the process of creating a self-aware Mankind, in which each man, thanks to the incredibly various sources of information, is capable of enlarging the horizons of his mind.

The enormous growth of computational technologies actually coincided with an enormous growth of the data available (mostly) on the Internet. Such data is *extremely heterogeneous*, we go from human readable, completely unstructured data such as *text* to semi-structured, still human readable data such as HTML[1] or XML[2], to completely structured and only machine readable data such as the data used in relational databases. The study of the algorithms and data structures needed to store and handle this huge quantity of diversified data in an efficient fashion has grown up together with the growth of data. The reason is that hardware improvements do not suffice to manage big data. An inefficient al-

[1] HyperText Markup Language.
[2] eXtensible Markup Language

gorithm will probably remain unusable whatever the speed of memory or processor, forever.

One of the most studied problem in the field of algorithm engineering for big data is the *prefix search* problem, which asks for preprocessing a set of variable length data[3] in such a way that we can retrieve all the elements that have the queried pattern P as prefix. This problem, easy to enunciate, is the backbone of many other more complex problems and nowadays it is experiencing a revamped interest because of the discovered usefulness of Search Engines auto-completion facility or because of the intense use of the IP-lookup facility in Internet based applications.
Indeed, the auto-completion facility provided by all the modern Search-Engines is exactly a prefix search query executed on the fly over a *dictionary* composed by the most relevant queries every issued by the users. Since the customers of any Search-Engine is in theory everyone on Earth we can imagine how big such a dictionary is and how much efficiently the prefix search facility must be provided in order to get the response in few milliseconds.
Another typical application of prefix search is Bioinformatics, where a common problem is to find the occurrences of a given pattern over the DNA chain. Indeed we can reduce this kind of problem to prefix searching the pattern over the set of all the possible suffixes of the DNA subset we want to analyse. The set of those suffixes is actually a dictionary, so we are prefix searching over a dictionary. A popular software that solves the aforementioned problem is BLAST [22], which stores the positions where all the different substrings of length L occur in the sequence database, allowing to index them.
Just to give an idea of how much this problem has been studied, we propose in the next lines a brief history of the research done around it.
The first (theoretically) efficient solution to prefix search dates back to 1960, when Fredkin in [18] proposed a new data structure, called (compacted) Trie capable of solve prefix search. From that point on, the trie structure became the most known solution for this problem. In particular, it became very famous in the algorithmic community during the 80s-90s because of one of its variant, known as the *Suffix Tree*, which dominated the scene for a long time.
The Oxford English Dictionary initiative [17] and the subsequent advent of the Web forced the design of efficient tries managing large datasets. However, scientists noticed soon that storing a trie in little space allowing performant prefix search queries was not easy at all. In particular, tries suffered of a poor amount of memory transfers [9]. The article that allowed to make a decisive step towards the solution of the problems posed in the paper just mentioned was [13], in which Ferragina and Grossi designed the String B-Tree data structure. In few words, String B-Trees are analogous to Tries and Suffix Trees, but they have some *redundant bits* in their representation that permit to overtake the bounds stated in [9].
After the devisal of the String B-Tree, the research moved further on, improving such data

[3] Data representable with various amount of bits.

structure in many of its aspects, for instance, space occupancy. One of the last step in the design of efficient solutions for prefix search has been made again by Ferragina *et al.* in [14]. In such a paper it is described a data structure which is close to the optimal storage complexity and supporting efficiently prefix search queries. The interesting feature of this data structure is that its space occupancy actually depends on a modifiable parameter, which allows the user to trade between space occupancy and query time.

This work The aim of this work is to develop a data structure capable of storing a set of strings in a compressed way providing the facility to access and search by prefix any string in the set. The notion of *string* will be formally exposed in Section 2.3, but it is enough to think a string as a *stream of characters* or a *variable length data*. We will prove that the data structure devised in our work will be able to search *prefixes* of the stored strings in a very efficient way, hence giving a performant solution to one of the most discussed problem of our age.
In the discussion of our data structure, particular emphasis will be given to both space and time efficiency and a tradeoff between these two will be constantly searched. To understand how much string based data structures are important, think about modern search engines and social networks; they must store and process continuously immense streams of data which are mainly strings, while the output of such processed data must be available in few milliseconds not to try the patience of the user.
Space efficiency is one of the main concern in this kind of problem. In order to satisfy real-time latency bounds, the largest possible amount of data must be stored in the highest levels of the memory hierarchy. Moreover, data compression allows to save money because it reduces the amount of physical memory needed to store abstract data. This fact is particularly important since storage is the main source of expenditure in modern systems. Indeed compressed data can be read faster by the CPU because any memory access brings to the CPU cache a *fixed* amount of bits, decided by the hardware.
Therefore, the more the data is physically compressed, the more the information accessible to the CPU. If important chunks of data are in the closest levels of memory, the probability of accessing a data which is in a farer memory level decreases, hence reducing the memory transfer cost, which is for nowadays algorithms on big data, the most relevant cost in terms of time.
A very important feature of our data structure is that the primitives that allow to prefix search and access any string in the dictionary are performed directly on the compressed data structure. What we mean is that we do not need the decompress the whole file to answer any query carried out on the dictionary, we just need to decompress a portion of the file, or a *block* of it.
In fact, we can say that our data structure is constituted by such blocks, and the smaller are the them, the poorer is the compression ratio. Anyway, if the blocks are large, the cost of decompressing one of such block can be not acceptable. For this reason, a tradeoff between the size of the blocks and their decompression time is constantly studied in this

work. We will see that we will adopt a solution capable of decompressing blocks that are guaranteed to be of *optimal* size.
Finally, we can say that we managed to devise a *parametric* and *compressed* data structure, solving the prefix search problem in asymptotically optimal time and space.
The following paragraph illustrates how we decided to organise the work done in this Thesis.

Thesis overview This Thesis is divided into seven chapters.

- *Chapter 2* Contains a review of the arguments that are needed in order to understand the topics covered in this work together with the notation that will be used.
- *Chapter 3* Illustrates the methods we devised and implemented in order to compress a set of strings and in order to provide access to any string or prefix belonging to the set.
- *Chapter 4* Shows the experiments that have been performed in order to prove the efficiency and the correctness of the methods presented in chapter 3.
- *Chapter 5* Illustrates the methods we devised in order to lookup any prefix or any string belonging to the set compressed and indexed with the same algorithms described in chapter 3.
- *Chapter 6* Shows the experiments performed in order to prove the efficiency and the correctness of the solutions provided in chapter 5. In chapter 6 will be presented also the comparison between our data structure and other state-of-the-art solution that have been proved to be particularly efficient.
- *Chapter 7* Sums up what has been done in this Thesis and indicates some future directions to improve our designed data structure.

Chapter 2

Background and Tools

In this chapter we propose a quick review of the arguments and tools that are needed in order to fully understand the topics covered in this work together with the notation that will be used. The exposition of every topic in this chapter will be supported by examples and reference to the literature for further reading.

2.1 Basic Notation

We denote the cardinality of a set S as $|S|$. Ranges of integers from 0 to a certain natural number n: $\{0, 1, ..., n-1\}$ will be denoted with $[n]$ or $[0, n)$ in case of ambiguity. All the logarithms will be in base 2, unless otherwise specified: $\log(x) = \log_2(x)$.

2.2 Asymptotic notation

Assume that $f(x)$ and $g(x)$ are functions from $\mathbb{N}$ to the set of all the positive real numbers $\mathbb{R}^+$.

1. $g(x) \in O(f(x)) : 0 \leq \lim_{x\to\infty} \frac{g(x)}{f(x)} < \infty$
2. $g(x) \in \Omega(f(x)) : 0 < \lim_{x\to\infty} \frac{g(x)}{f(x)} \leq \infty$
3. $g(x) \in \Theta(f(x)) : 0 < \lim_{x\to\infty} \frac{g(x)}{f(x)} < \infty$
4. $g(x) \in o(f(x)) : \lim_{x\to\infty} \frac{g(x)}{f(x)} = 0$
5. $g(x) \in \omega(f(x)) : \lim_{x\to\infty} \frac{g(x)}{f(x)} = \infty$

According to a deep rooted tradition of computer science literature, in this work we will drop the set notation and substitute the "$\in$" symbol with the "$=$" symbol. So for example writing $f(x) = O(g(x))$ has the same meaning of writing $f(x) \in O(g(x))$.
The notation illustrated in this section is of crucial importance because it allows to compare the behaviour of very complex functions to the behaviour of other very well known ones when their input grows large.
For example if $f(x) = \frac{1}{2}x^2 + 2x - 1$, we may say that $f(x) = O(x^2)$, or $f(x) = \Theta(x^2)$[1] because $\lim_{x\to\infty} \frac{\frac{1}{2}x^2+2x-1}{x^2} = \frac{1}{2}$.
Another widespread tradition allows to use asymptotic notation inside formulas. For instance we may meet an equation like $4x^2 + 10x - 3 = 4x^2 + O(x)$. Obviously also in this case $O(x)$ does not represent a set (how can we add a set to an expression?), but a generic function $g(x) \in O(x)$, which in this case is $10x - 3$. When functions are represented in such a way, they are conventionally called *anonymous functions*.
In the following we also give the intuitive meaning of the formal definitions given in this section.

1. $g(x) \in O(f(x))$: g asymptotically grows *not faster* than f
2. $g(x) \in \Omega(f(x))$: g asymptotically grows *not slower* than f
3. $g(x) \in \Theta(f(x))$: g has asymptotically the *same growth* of f
4. $g(x) \in o(f(x))$: g asymptotically grows *slower* than f
5. $g(x) \in \omega(f(x))$: g asymptotically grows *faster* than f

2.3 Sequences

Definition 1. *We define a sequence of length n simply as an ordered collection of elements* $s = \langle s_0, s_1, ..., s_{n-1}\rangle$ *belonging to an alphabet* Σ *of cardinality* σ*.* s_i *or* $s[i]$ *is the i-th element of s, and* $|s| = n$*.*

We denote the set of all the sequences of length n drawn from Σ with Σ^n. The set of empty sequence Σ^0 will be also called ϵ, while the set of sequences of any possible length is called $\Sigma^* = \bigcup_{i=1}^{\infty} \Sigma^i$. The elements of the alphabet are also called *symbols*.
We will also exploit a notation very well known in computer science and engineering world: $s[i, j]$ to denote a contiguous range of elements of a sequence s, i.e.,

$$s[i, j] = \langle s_i, s_{i+1}, ..., s_j\rangle \ .$$

[1] Note that $\Theta(g(x)) = O(g(x)) \cap \Omega(g(x))$.

It is important to note that $s = s[0, |s| - 1]$[2] and $|s[i, j]| = j - i + 1$.
A binary sequence, i.e., a sequence drawn from alphabet $\Sigma = \{\mathtt{0}, \mathtt{1}\}$ is called *bitvector*. The symbols of such an alphabet are also called *bits*, while a sequence made by *characters* is called a *string*.
For simplicity, strings will be denoted as plain sequences of characters, without any separator. For instance in the following way.

$$s = \mathtt{abaco}$$

In this case $|s| = 5$, $s_0 = \mathtt{a}$, $s_4 = \mathtt{o}$.
Moreover, for any string s, its *prefix* of length $l \leq |s|$ is exactly $s[0, l - 1]$. A generic string p is a prefix of s if for $i = 0, ..., |p|$ we have that $s_i = p_i$.
Recursively, we denote a *sequence of strings* S as a sequence $S \in \Sigma^{**}$ drawn from an alphabet of strings Σ^*, which in turn are drawn from an alphabet Σ.
A *sequence of strings* can be represented as follows.

$$S = \langle \mathtt{abaco}, \mathtt{zio}, \mathtt{nonno}, \mathtt{abate}, \mathtt{abaco} \rangle$$

In this case $|S| = 5$, $S_0 = \mathtt{abaco}$, $S_2 = \mathtt{nonno}$.
It is important to mention the fact that any sequence s drawn from an alphabet Σ can be represented as a binary sequence. This is possible if we represent any symbol of Σ as a binary sequence of length $\lceil \log |\Sigma| \rceil$.
The *binary representation* of s, denoted $B(s)$, is therefore the concatenation of the binary encodings of the symbols constituting s. Hence, the *length* or cardinality of $B(s)$ can be expressed by the following.

$$|B(s)| = |s| \lceil \log |\Sigma| \rceil \text{ bits.}$$

2.4 Basic Operations on sequences

In this section we mention some of the most common operations used on sequences. Suppose that a generic sequence b is drawn from an alphabet Σ and k is any of the symbols of Σ. Then we define the following

- $\mathsf{Rank}_k(b, i)$ returns the number of occurrences of k in $b[0, i - 1]$
- $\mathsf{Select}_k(b, i)$ returns the position of the i-th occurrence of k
- $\mathsf{Access}(b, i)$ returns the i-th element of b
- $\mathsf{Predecessor}_k(b, i)$ returns the position of the *rightmost* occurrence of k preceding or equal to i

[2] We follow the convention accepted by all the major programming languages according to which the index of the first element of any collection is 0.

If the sequence on which the operation are applied is clear from the context, then from now on any of the operations above can be written without the first argument, e.g., $\mathsf{Select}_k(b, i) = \mathsf{Select}_k(i)$.
So, if we take the following bitvector b

$$b = 101101001110011$$

$$\mathsf{Access}(2) = 1,\ \mathsf{Rank}_1(0) = 0,\ \mathsf{Rank}_1(6) = 4,\ \mathsf{Select}_1(0) = 0,\ \mathsf{Select}_1(3) = 5,\ \mathsf{Predecessor}_1(7) = 5$$

The following properties are noteworthy.

1. $\mathsf{Rank}_k(\mathsf{Select}_k(i)) = i$
2. $\mathsf{Select}_k(\mathsf{Rank}_k(i)) = \mathsf{Predecessor}_k(i)$

Now we mention other important operations which are defined only for sequences of sequences. Taking a generic alphabet Σ and $\mathcal{S} \in \Sigma^{**}$, we define the following.

- $\mathsf{lcp}_{\mathcal{S}}(i, j)$ returns the *longest common prefix* among the sequences in range $[i, j]$
- $\mathsf{Retrieval}_{\mathcal{S}}(i, l)$ returns the first l symbols of the sequence returned by $\mathsf{Access}(\mathcal{S}, i)$, or, equivalently $\mathsf{Retrieval}_{\mathcal{S}}(i, l) = \mathsf{Access}(\mathcal{S}, i)[0, l-1]$

2.5 Theoretical lower bounds

In this section we report the space occupancy theoretical lower bounds concerning the data structures we are going to deal with in this work.

Sequences. When we have to deal with sequences, if we do not dispose of any additional information except the alphabet Σ and the sequence length n, then, as a consequence of the pigeonhole principle, for each encoding algorithm, there will be at least one sequence encoded with at least $n\lceil \log |\Sigma| \rceil$ bits.
This quantity is the space occupied by the binary representation of any sequence and it can be viewed as a *worst case* lower bound.
If we can have access to some additional information, i.e., the data set we are going to compress, we may obtain more satisfying bunds which are called *data dependent* bounds.
A very theoretical way to achieve a perfect lower bound is via the study of the *Kolmogorov complexity* [27], which is defined as the shortest encoding of a Turing machine that outputs a given sequence. This complexity, despite its clear definition, is actually impossible to compute starting from a specific sequence and it's therefore useless in the devising of lower bounds.

A paper by one of the fathers of *Information theory*, Claude Elwood Shannon, [36] introduced the notion of *information content* of a discrete random variable X which can have values belonging to a finite set Σ. We call the distribution of such a random variable $(p_c)_{c\in\Sigma}$, so that $p_c = \mathbb{P}\{X = c\}$. He then defined the *entropy* of X as

$$H(X) = -\sum_{c\in\Sigma} p_c \log(p_c)$$

If we take a generic sequence s and we assume *empirical frequencies as probabilities*, i.e., $p_c = n_c/n$, where n_c is the number of occurrences of c in s. We can define what is called *0-th order empirical entropy* of a sequence s of length n as

$$H_0(s) = -\sum_{c\in\Sigma} \frac{n_c}{n} \log \frac{n_c}{n} = \sum_{c\in\Sigma} \frac{n_c}{n} \log \frac{n}{n_c} \text{ bits.} \tag{2.1}$$

Now, suppose that s is a binary sequence of length n which has n_1 bits set to 1. Because of the empirical frequencies as probabilities assumption, we have that $p_1 = \frac{n_1}{n} = p$ and $p_0 = 1 - \frac{n_1}{n} = 1 - p$. We can rewrite the 0-th order empirical entropy of such a sequence in the following way

$$H_0(s) = -p\log p - (1-p)\log(1-p) = p\log\frac{1}{p} + (1-p)\log\frac{1}{(1-p)}$$

If we assume that the elements of s are independent and identically distributed random values, $H_0(s)$ can be an estimate of $H(X)$, and $p_c = \frac{n_c}{n} \forall c \in \Sigma$ are called *empirical probabilities*.
Concerning our main aim, which is define a practical lower bound in space occupancy valid for any sequence, the following important theorem has been shown in [10]

Theorem 1. *The minimum number of bits needed to encode a generic sequence s of length n with an encoder that maps every symbol of the alphabet Σ into a binary sequence is equal to $nH_0(s)$.*

The encoder described in Theorem 1 is *position independent*. As a first step it defines the binary encoding for each symbol of the sequence s and then it directly substitutes any of the occurrence of this symbol in s with its binary representation.
In general, we can also define the *k-th order empirical entropy* of a generic sequence $H_k(s)$ as follows

$$H_k(s) = \sum_{u\in\Sigma^k} \frac{|s_u|}{n} H_0(s_u) \tag{2.2}$$

where $u \in \Sigma^k$ is a sequence composed of k alphabet symbols and s_u is the sub-sequence of s that follows any occurrence of u.
The impressively important result reported in the following Theorem was given by Manzini in [28].

Theorem 2. *$nH_k(s)$ is the lowest number of bits needed to encode any sequence s of length n using an encoder that decides the encoding of a symbol basing its decision on the k symbols that precede it (also called its k-context).*

Subsets. If we call U a set of cardinality n, usually referred as *universe* and X a subset of this universe, whose cardinality is $m \leq n$, then the minimum number of bits needed to represent X is given by the following.

$$\mathcal{B}(m,n) \;=\; \left\lceil \log \binom{n}{m} \right\rceil \tag{2.3}$$

Notice that $\mathcal{B}(m,n)$ is nothing more than the logarithm of the number of all the possible subsets of m elements out of a universe U of cardinality n.
The following interesting properties also hold:

1. $\mathcal{B}(m,n) \leq nH(\frac{m}{n}) + O(1)$
2. $\mathcal{B}(m,n) \leq m \log \frac{m}{n} + O(m)$

The first of the properties above is intuitively linked to the fact that any subset S of size m of a universe of size n can be represented as a bitvector of length n with m bits set to one. This bitvector is called the *characteristic function* of S.

2.6 Models of computation

Models of computation provide a way to estimate the computational complexity of any algorithm through the definition of a set of *elementary operations*. The computational complexity, representing an estimation of the completion time of an algorithm is therefore given by the number of such elementary operations performed by an algorithm during its execution.

Word-RAM model. The most used model of computation emulating the behaviour and the set of operations available on modern machines is the *word-RAM* model. In this model, the unit of memory is called *word*, and its size is w bits. Any word can be accessed in constant time. The basic operations are the typical logic and arithmetic ones, including also the bitwise shifts.

If we denote the size of any problem (or input) with n, then the *word*-RAM model also assumes that the size of a word is bounded by the following equation.

$$w = \Omega(\log n)$$

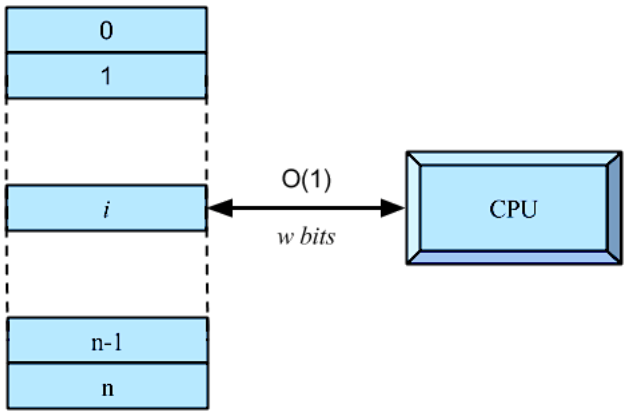


Figure 2.1: RAM model of computation

So that the word size is, for large enough n, at least $k \log n$ for some constant k. Indeed if this were not true, then we could not fit any address of the input in a constant number word.
A stronger condition is the *transdichotomous assumption* [19],which states the following.

$$w = \Theta(\log n)$$

i.e., the length of the word is upper and lower bounded by a multiple of the logarithm of the input size. The reason for its name is due to the fact that this assumption ties the characteristic of an abstract CPU (the word size) to the size of the problem.

External-memory model. Even though the *word*-RAM model is widely used in computer science, it does not take into account one of the most important feature of modern computers; that is, it assumes that the access time to any memory location is constant.
This is particularly false nowadays because modern machines memory are hierarchical and the access to a datum that is located for example in main memory could be thousand of times slower than the access to one that is in primary cache. A model that takes into account the organisation of modern memories is the *external-memory model* [38]. It considers the machine as composed by two levels of memory: one "fast" memory of bounded size M (e.g. the main memory) and another one of unbounded size (e.g., the hard disk), which is considered "slow". The I/O complexity of an algorithm is computed taking in consideration the number of transfers from the external memory to the main one. Each one of these transfers moves exactly B words across the memories.
Actually, reality is much more complex. Modern machines have many memories hierarchically organised. Typically there are 2-3 level of cache memories, the RAM memory (or

main memory), the disk and possibly the "Network" of memories, made by the union of all the machines that are connected through a network infrastructure to the one executing the algorithm.
The faster the memory, the smaller it is. The smallest memory is layer 1 cache (L1 cache), whose size is in KiloBytes as order of magnitude. Moving from an inner layer to an outer layer, the size is increased by one order of magnitude as well as the access time.
Memory layers do not differ only in size and access time, also the amount of words fetched per reference increases while moving away from the CPU, as it is shown in Figure 2.2.

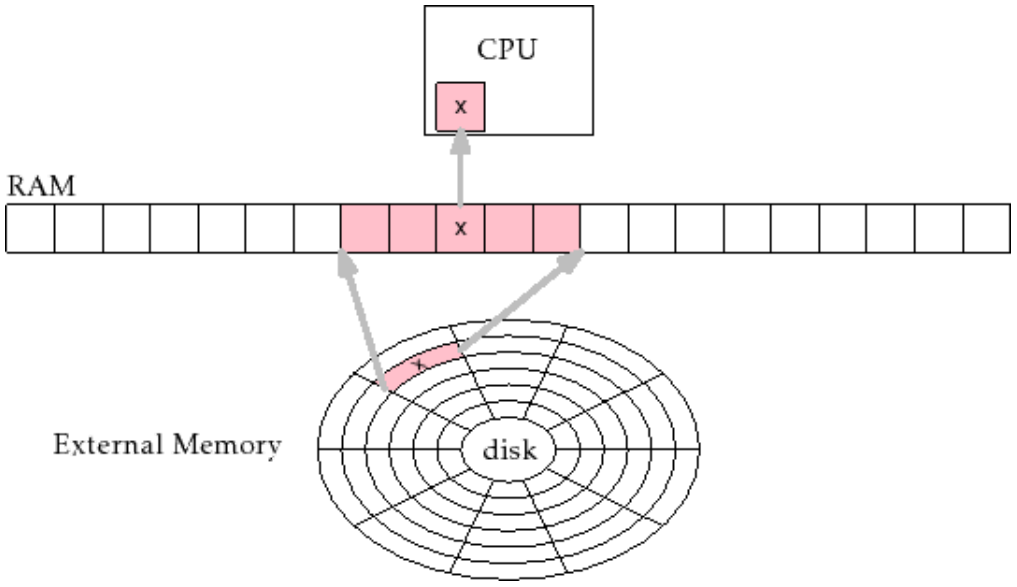


Figure 2.2: External-memory model of computation. Image borrowed from [29]

Cache-oblivious model. Another very well known model is the *cache-oblivious model*, introduced in [20]. This model is very similar to the external-memory one but it assumes that the size of the main memory M and the "page size" B for external memory transfers are *unknown* to the analysed algorithm. This assumption prevents the algorithm designer to "tune" the algorithm according to B. Actually, designing an efficient algorithm for this kind of model is particularly valuable because it is performant whatever are the values of B or M. Good cache-oblivious algorithms or data structures are extremely useful in reality, since modern machines have many levels of memory with different sizes.

2.7 Succinct representation of sequences

Succinct data structures have been introduced by Jacobson in [25] as data structures that occupy a space equal to their theoretical lower bounds plus a negligible number of bits. This additional space is actually used to let the succinct data structure support in efficient time i.e., in a time comparable to the one needed to perform the same operation on non succinct data structures, the operations described in Section 2.4.
In particular, Jacobson proposed in his paper a succinct data structure which could encode a bitvector of size n with $n + O(\frac{n \log \log n}{\log n})$ bits, supporting Access and Rank operations in *constant* time.
The data structure proposed by Jacobson was later be improved by Clark in [7], allowing to state the following theorem.

Theorem 3. *There exists a data structure $\mathfrak{D}$ that can encode a bitvector b of size n with*

$$|\mathfrak{D}| = n + O(\tfrac{n \log(\log n)}{\log n}) \text{ bits}$$

supporting Access, Rank *and* Select *operations in $O(1)$ time.*

The field of applications of succinct data structures is vast and the benefit is significant expecially for large data sets that can be kept in main memory when encoded in succinct format.

Fully indexable dictionaries. Raman et al. in [35] further improved the results by Jacobson and Clark introducing the term *fully indexable dictionary* (FID) defining a data structure that was able to support the properties asserted in the following theorem.

Theorem 4. *There exists a data structure $\mathfrak{F}$ able to encode a set of m elements drawn from a universe U of size n such that*

$$|\mathfrak{F}| = \mathcal{B}(m, n) + O(\tfrac{(n \log \log n)}{\log n}) \text{ bits}$$

and capable of supporting Access, Rank *and* Select *operations in $O(1)$ time.*

The improvement with respect to the data structure introduced by Clark is given by the fact that we can encode a bitvector of length n with m ones with the bound reported in the Theorem above. In fact the following equation shows that $\mathcal{B}(m, n) < n$.

$$\mathcal{B}(m, n) \;=\; \left\lceil \log \tbinom{n}{m} \right\rceil < \left\lceil \log \sum_{k=0}^{n} \tbinom{n}{k} \right\rceil = \lceil \log 2^n \rceil = n$$

Elias-Fano representation. The *Elias-Fano representation of monotone sequences* was introduced in [11] and [12]. It is a widely exploited and elegant data structure satisfying the properties stated in the following theorem.

Theorem 5. *Given a non decreasing sequence of non negative integers* $s = \langle a_1, a_2, ..., a_m \rangle$ *drawn from the universe* $[0, n-1]$ *of cardinality* n*, there exists an encoding scheme able to represent* s *with* $2m + m\lceil \log \frac{n}{m} \rceil + o(m)$ *bits and supporting* Access *operation in* $O(1)$ *time. Such a representation of* s *is called Elias-Fano representation of* s*.*

Proof. Let's call $l = \lfloor \log(n/m) \rfloor$. First of all, each integer a_i is transformed in its binary representation $B(a_i)$, composed by $\lceil \log n \rceil$ bits. Then, the binary encoding of each integer is split in two parts: the first part is formed by the most significant $\lceil \log n \rceil - l$ bits, called *higher* bits, while the second one is formed by the least significant l *lower* bits.
Now, the higher and the lower bits are treated separately. The higher ones are represented as a bitvector H of length $m + \frac{n}{2^l}$ able to support the Select operation (hence requiring additional $o(m)$ bits according to Theorem 3). If h_i is the value of the higher bits of the integer a_i, the position $h_i + i$ of H is set to one. All the other positions of H are set equal to zero. In order to define h_i unambiguously, it is better to say that

$$h_i = \sum_{j=l}^{\lceil \log n \rceil - l - 1} 2^{j-l} B(a_i)[j]$$

Another bitvector L of length ml is formed concatenating the lower bits of the integers.
To access the i-th integer we have to retrieve the higher and the lower bits of it and concatenating them. Therefore, we start accessing the lower bits of a_i which is easy because they are exactly L[l(i-1),li-1], then we retrieve h_i noting that $h_i = Select_1(\mathrm{H}, i) - i$. □

It is very important to notice that the Elias-Fano representation is particularly useful and efficient to represent a generic *sparse bitvector*[3] b by encoding the sequence of the positions of the ones. In this way accessing to the i-th integer of such a sequence can be interpreted as $\mathsf{Select}_1(b, i)$

2.8 String dictionaries

If we call $\mathcal{S} \in \Sigma^*$ a set of string (Σ is an alphabet), a *String dictionary* is a data structure that stores this set and supports the following operations.

1. Lookup(s) returns an integer for each string s such that $s \in \mathcal{S}$, or NULL otherwise.
2. Access(i) returns the string s such that Lookup(s)=i.

It is possible to say that Lookup and Access primitives form a one-to-one correspondence between $\mathcal{S}$ and $[|\mathcal{S}|]$.

[3] A sparse bitvector is a bitvector having much more 0s than 1s

2.9 Range Minimum queries

A *Range Minimum Query* (shortly RMQ) is an operation defined on a sequence A whose elements belong to a *totally ordered* universe U.[4]
$\mathsf{RMQ}_A(i,j)$, with $i \leq j$ retrieves the position of the minimum element of $A[i,j]$ according to the ordering relation of the universe U (in case of ties the leftmost position is returned), for example.

$$A=[8,7,3,20,2,17,5,21,11,12]$$

$$\mathsf{RMQ}_A(2,6)=4;\ \mathsf{RMQ}_A(6,9)=6;$$

A trivial implementation of $\mathsf{RMQ}_A(i,j)$ would be to scan the entire interval $A[i,j]$, thus requiring $O((j-1)) = O(|A|)$ elementary operation.
The range minimum query problem on sequences is reducible to the *Least Common Ancestor* (LCA) problem on trees and LCA is reducible to RMQ. In fact, LCA can be reduced to RMQ transforming the nodes of the tree in a sequence of integers, while RMQ can be reduced to LCA transforming the sequence in a tree, called *Cartesian Tree*, as proved by Bender and Colton in [1].
Fischer and Neun devised in [16] a data structure able to support RMQ in constant time on any sequence of length n with $2n + o(n)$ bits, hence with a negligible overhead.

2.10 Integers encodings

In this section we face the problem of representing a sequence of positive integers in a binary output alphabet {0,1} using the least possible number of bits. This problem has to be tackled in many situation, for example search engines inverted indexes would require too much space if integers were stored not compressed.
The simplest encoding of a positive integer is its binary representation. Suppose that we must encode a sequence A of integers whose maximum value is M, then we need $\lceil \log M \rceil$ bits for each number, which can imply a large number of non meaningful bits if the sequence is composed mostly by small numbers.
In general, according to the Shannon's theory [36], the following equation leads to the probability distribution $P(x)$ such that the binary representation of x has optimal length L_x.

$$P(x) = 2^{-L_x} \tag{2.4}$$

Therefore, since the formula above holds for any x, if we use a fixed length binary encoder we need that the numbers are uniformly distributed in $\{1, ..., M\}$; something that happens very rarely in reality. As a consequence, it is much more fruitful to encode the numbers

[4] A totally ordered universe is simply a set in which the relation $\leq$ is defined.

with a different amounts of bits.
Huffman encoding [24], provides an algorithm capable of encode any sequence of integers in an optimal way. Unfortunately this method requires the explicit storage of a data structure of size $O(M \log M)$ that must be accessed in order to decode the integers, hence leading to performance degradation both in time and space.
In the following subsection we will show some of the most common integers encoders that do not require any additional data structure.

2.10.1 Bit-aligned integers encoders

Bit-aligned encoders are integers encoders such that $\forall x \in \mathbb{N}, |E_b(x)| = y$ bits. Where $E_b(x) : \mathbb{N} \to \{\texttt{0}, \texttt{1}\}^*$ is a generic Bit-aligned encoder of an integer x.
Bit-aligned encoders produce representations which are just a constant factor far from the optimal one. For this reason they are indeed theoretically good encoders, nevertheless they are very slow during the decoding phase because of the many bit related operation they require.

Elias-γ coding *Elias-γ* coding is a bit-aligned integer encoder that represents a number x with its binary representation prefixed by the its length expressed in unary. The last bit of the prefix is shared with the first bit of the binary representation (which is always `1`). So, encoding x with Elias-γ encoding requires overall $2\lfloor \log x \rfloor + 1$ bits. We denote the Elias-γ encoding of an integer x with $\gamma(x)$.
Some examples are shown in the following line.

$$\gamma(9) = \texttt{000|1001};\ \gamma(14) = \texttt{000|1110}$$

Elias-δ coding This kind of encoding, denoted $\delta(x)$ represents an integer x with its binary representation prefixed by the γ coding of its length. Thus for example.

$$\delta(9) = \texttt{00|100|1001};\ \delta(14) = \texttt{00|100|1110}$$

2.10.2 Byte-aligned encoders

Byte-aligned encoders are integers encoders such that the cardinality of the binary sequence outputted by them is always a multiple of 8. It is therefore measurable in bytes.
Byte-aligned encoders are theoretically worse than Bit-aligned ones for what concern space occupancy. Anyway they are much faster and easier to implement because the smallest unit of measure of indexing in modern machines is the Byte. In general Byte-aligned encoders provide an excellent tradeoff between time and space.

Variable-bytes coding *Variable-bytes coding* (shortly VB) represents any integer x with a variable number of Bytes. In each of those Bytes there is a *status* bit followed by 7 bits of data. If the status bit is 0 then it means that the algorithm is currently scanning the last byte of the encoded integer, else, also the following Byte has to be scanned. In order to get the binary representation of x we just have to concatenate the payload bits of each of the scanned Bytes. So for example:

$$\mathsf{VB}(2^{16}) = \mathsf{VB}(\texttt{100 0000000 0000000}) = \texttt{1|0000100 1|0000000 0|0000000}$$

VB-Fast coding *VB-Fast coding* (VBFast) has, as its name suggests, a very fast decoding phase. The reader will immediately argue why from its description. $\mathsf{VBFast}(x)$ outputs the leftmost byte as composed by the leftmost six bits of the binary representation of x (if x requires less than 6 bits, the remaining bits are padded) prefixed by 2 bits indicating the number of bytes in addition to the first required to encode the x. In the remaining bytes there are the rightmost bits of the binary representation of x properly padded. So for example:

$$\mathsf{VBFast}(10)=\texttt{00|001010};\ \mathsf{VBFast}(127)=\texttt{01|111111 00000001}$$

2.11 Prefix search

In the following we give the definition of the *prefix search* problem, which is the heart of this work.

The Prefix search problem. *Given a set of strings $\mathcal{D}$ consisting of n strings whose total length is N, drawn from an alphabet Σ, and an input string P, the problem consists on preprocessing $\mathcal{D}$ in order to retrieve the strings of $\mathcal{D}$ that have P as prefix.*

The *prefix search* problem is experiencing huge interest by the algorithmic community because of its applications in web search engines. The *auto-completion* facility currently supported by the most known search engines is one of them. In fact it is a prefix search over a dictionary composed by the millions of most frequent and recent queries issued by the users.
The solution is given on the fly and what id called P in the definition of the problem is actually the query pattern the user is currently inserting in the search bar.
The problem is challenging because it requires that the answers to it must be given in "real time" in order not to try the patience of the user.
Another problem that is strictly linked to prefix search is *substring search* over a dictionary $\mathcal{D}$. Substring search is a sophisticated problem and finds important application for example in computational genomics and asian search engines. It consists on finding all the positions where the query pattern P occurs in as a substring of the strings in $\mathcal{D}$.
This apparently complex problem can actually be *algorithmically reduced* to prefix search

over the set of *all suffixes of the dictionary strings*.
In practice, prefix search is the *backbone* of all important problems related to the searching of strings.
Now, supposing that the strings are lexicographically ordered so that we can give an increasing index to them and supposing that the dictionary $\mathcal{D}$ and the query pattern P are given as in the prefix search definition, we can divide prefix search problem in the following ones.

Weak prefix search. *Returns the range of strings prefixed by P, or an arbitrary value whenever such strings do not exist.*

Full prefix search. *Returns the range of strings prefixed by P, or* NULL *whenever such strings do not exist.*

Longest prefix search. *Returns the range of strings sharing the longest common prefix with P.*

In this work we are mainly interested to find a solution to the *full prefix search* problem.

2.12 Tries

A *trie* is an ordered tree-like data structure, invented by De la Briandais in [5], capable of storing and indexing any *dynamic set*.[5] They are widely used because they allow to prefix search any pattern P on any set of strings $\mathcal{S}$ in $O(P)$ time, thus regardless of the number of strings.
In particular, a trie is a multi-way tree whose edges are labeled by characters of the strings belonging to $\mathcal{S}$, *exactly one character per edge*. Any node u of the trie is associated with a string, denoted by $\mathsf{string}(u)$, obtained by concatenating the characters on the edge labels in the path from the root to u. Therefore, if v is a leaf of the trie, $\mathsf{string}(v)$ is one of the strings in $\mathcal{S}$. In general, for any internal node u, $\mathsf{string}(u)$ is a *prefix* of one or more strings in $\mathcal{S}$.
The following observation is crucial.

Observation 1. *If u is a node of trie $\mathcal{T}$, then $\forall\ v \in \mathcal{T}$ such that v is a leaf and there exist a downward path from u to v,* $\mathsf{string}(u)$ *is a prefix of* $\mathsf{string}(v)$*. In other words* $\mathsf{string}(u)$ *is the longest common prefix shared by all the strings associated to the leaves descending from u*

We may also notice that a trie has exactly $n = |\mathcal{S}|$ leaves, and at most $N = \sum_{i=0}^{n-1} |\mathcal{S}_i|$ nodes ("at most" because some paths can be shared among several strings).[6]

[5] We can considered a dynamic set as a sequence whose elements can be added or removed arbitrarily
[6] Note that $N = \Omega(n)$.

If we need to check wether a string P prefixes some strings in the set, we just have to see if there exist a downward path from the root to any node u "spelling" P, i.e., a path whose edge label characters are the same of P. The main problem in tries is how to efficiently implement the following of the path, that is, determine which is the edge we need to chose in order to follow the characters of P. Indeed, the very best solution is to use a *perfect hash table* which stores only the used characters and the pointers to their associated nodes.

2.12.1 Compacted trie

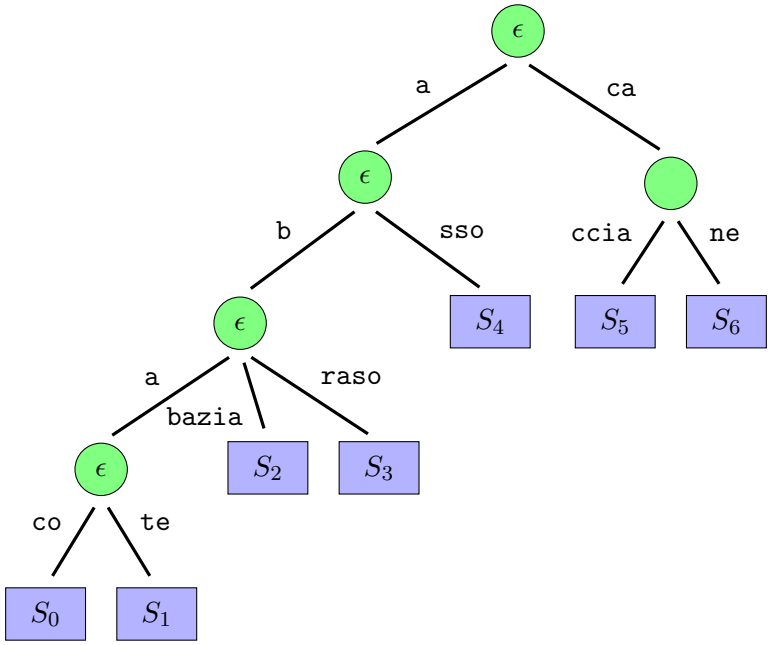
The trie described in the previous section can be of unacceptable size if there are long strings with short common prefixes, thus resulting in a trie with a lot of unary nodes, i.e., nodes with just one son. To circumvent this situation we may *contract* the unary paths into one single edge, making edge labels variable length strings instead of single characters. The resulting trie is called *compacted*, or PATRICIA and any edge label in such a trie is actually a substring of a string $\mathcal{S}_k \in \mathcal{S}$: $\mathcal{S}_k[i, j]$. Thus, edge labels can be represented as a triple $\langle k, i, j \rangle$.
Since any node is at least binary and there are no NULL pointers in internal nodes, we have that the total number of nodes in a compacted trie is $O(n)$. As a consequence, the space occupied by a compacted trie is $O(n)$ too. The compacted trie was devised by Morrison in [30].

2.12.2 PATRICIA trie

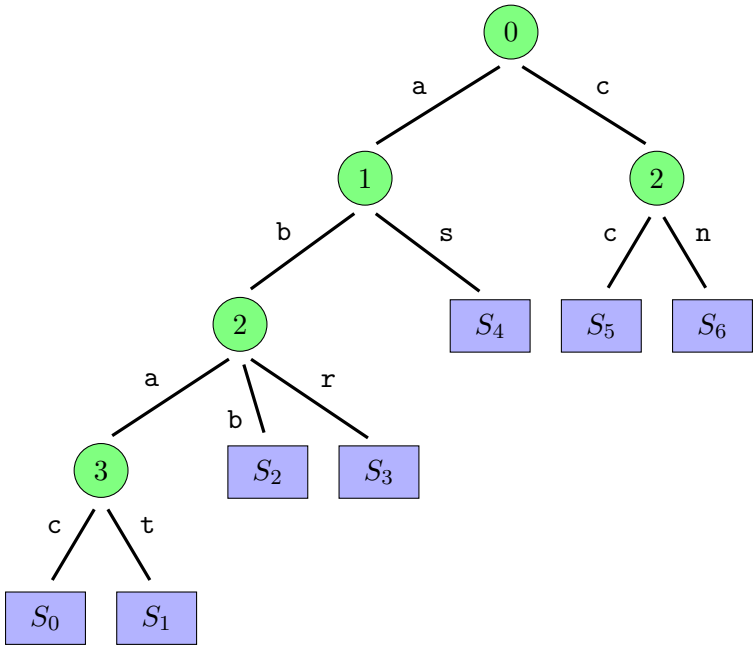
A PATRICIA trie, or blind trie, defined in [13], is a compacted trie whose edge labels are single characters and whose nodes are labelled with the length of their associated strings. The conversion from a compacted trie to a Patricia trie is straightforward and, rather than the Patricia trie strips some information out of the associated compacted trie, it is still able to perform prefix search over a set of strings $\mathcal{S}$ with the same asymptotical complexity: $O(|P|)$, exploiting an algorithm called *Blind search*.
Moreover, the fact that we just have to store a *single* character per edge label greatly increases the possibility to store the whole trie in main memory, avoiding the I/Os needed to access the substring characters in the compacted trie. For this reason, PATRICIA tries are extremely efficient in practice. Indeed, because of the fact that any node in a PATRICIA trie is at least binary and there are no NULL pointers stored in internal nodes[7], we may point out that it occupies $O(n)$ bits of space, thus $O(1)$ bits per indexed string (e.g., 8 bits for the edge label and 32 bits for the node labels). Moreover, for the same reasoning, the time to perform a *blind search* of a prefix of $\mathcal{S}$ on such a trie belongs to $O(\log |\mathcal{S}|)$.

[7] The topology of PATRICIA trie indexing a set $\mathcal{S}$ is the same of the compacted trie indexing the same set.

Figure 2.3: Tries associated to the set of strings $\mathcal{S} = \{$`abaco, abate, abbazia, abraso, asso, caccia, cane`$\}$



(a) Compacted trie associated to $\mathcal{S}$



(b) PATRICIA trie associated to $\mathcal{S}$

2.13 Ternary search trees

Ternary search trees (shortly TST) were described for the first time by Bentley and Saxe in [3]. They combine the features of binary search trees and search tries. Any node of a ternary search tree stores a single character, a pointer to an object (or an object depending on the implementation) and exactly three pointers to its children. Because of the limited number of sons per node TSTs are much more space efficient than tries, whose nodes must store one pointer for each alphabet symbol.
When we search for a string (or a prefix) in this data structure, we compare the character `a` associated to the current node with the currently scanned character of our string, call it `b`. If $\mathtt{a} < \mathtt{b}$ we continue the search in the right son of the current node, if $\mathtt{a} > \mathtt{b}$ we search in the left son; if instead $\mathtt{a} = \mathtt{b}$ then we may examine the next character of the looked string and proceed the search in the middle child of the node.
The search stops when all the character of our string has been matched or when a mismatch is found but the current node has not a pointer to the son we would want to visit.
In order to insert a new string S in the ternary search tree we follow the same procedure used to search S up to we do not find a mismatch. Suppose that $S[i]$ is the mismatch character and `a` is the character contained by the current node u; if $S[i] < \mathtt{a}$ then the left child of u becomes the root of a ternary search tree formed by $|S| - i + 1$ unary nodes with only the pointer to the middle child set, whose labels are the symbols spelling the characteristic suffix of S; else, the same tree is rooted by the right child of u.
Unfortunately space efficiency in TSTs comes at the cost of time. Indeed we have to perform a percolation of binary search tree for each character of the looked string, which may cost from $O(\log|\Sigma|)$ comparisons to $O(|\Sigma|)$ comparisons, where Σ is the alphabet from which the strings are drown. The cost of such a search depends on how well the tree is balanced.
Eventually, a search in a ternary search tree would require from $O(|P|\log|\Sigma|)$ up to $O(|P||\Sigma|)$ comparisons in the worst case, therefore, keeping a ternary search tree balanced is crucial for the performance of the search. The balancing of a TST is directly dependent on the order in which the strings are inserted in the data structure.
In this regard, imagine to insert a set of strings one by one according to their lexicographic order: there would be no node with a left son and the whole data structure would result in a concatenation of linked lists.
In this case looking for a match in each character of the pattern P would require $O(|\sigma|)$ comparison in the *average* case. Moreover, It has been shown in [3] that searching for a string out of n in a *perfectly balanced* ternary search tree requires at most $\lfloor\log n\rfloor + |P|$ comparisons.
If we know all the strings we are going to insert in the tree, then we may insert them in random order to get a *randomised ternary search tree* composed by *randomised binary search trees*, which are binary search trees requiring amortised $O(\log|\Sigma|)$ comparisons for searching in the worst case scenario. For deeper understanding of amortised analysis see

[37].
We may also build a *completely balanced tree* by inserting the median element of the input set, then recursively apply the same criteria of selection inserting all the lesser strings and greater strings.[4]
Ternary search trees suffer from the same "unary path pathology" of digital search tries. Fortunately we can build *compacted* ternary search trees to remove unary paths storing string pointers instead of single characters in each node. We may also build *PATRICIA* ternary search trees which can avoid random I/Os on the string set while performing searches.

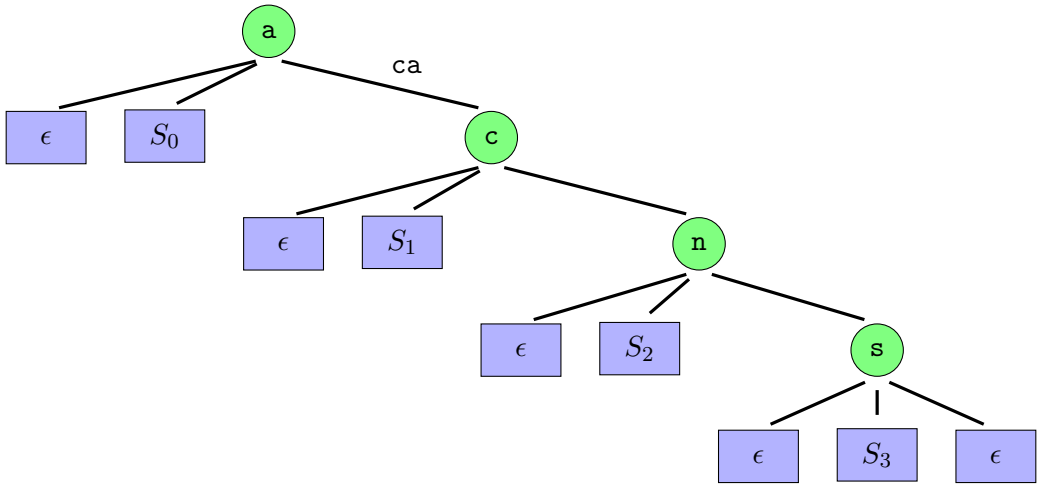


Figure 2.4: Unbalanced compacted ternary search tree according to string set: {abaco,caccia,cane,caso}.

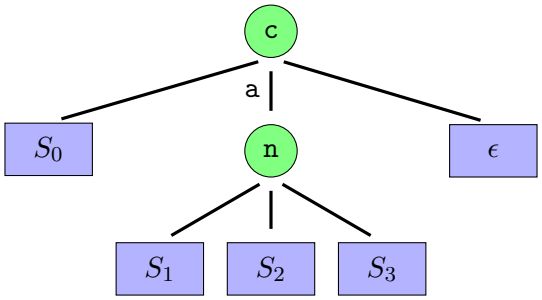


Figure 2.5: Balanced compacted ternary search tree according to string set: {abaco,caccia,cane,caso}.

2.14 Path decomposed tries

Path decomposition is an effective way to reduce the height of any trie. A path decomposition $\mathcal{T}^c$ of a trie $\mathcal{T}$ is a tree whose nodes represent a *path* of $\mathcal{T}$.
Path decomposed tries can be defined recursively in the following way: a root-to-leaf path in $\mathcal{T}$ is chosen as the root of $\mathcal{T}^c$. The same procedure is applied for each sub-trie hanging off the chosen root-to-leaf-path. Any path branching from the root-to-leaf path chosen as the root of the path decomposed trie becomes a child of the root of $\mathcal{T}^c$. There is not a specified order for the children of the root; for instance, in [14] the sub-tries are arranged in lexicographic order, while in [34] they are arranged in bottm-to-top left-to-right order.
The main property of path decomposed tries is that there exist a one-to-one correspondence among the nodes of $\mathcal{T}^c$ and the paths in $\mathcal{T}$. That is, a root-to-node path in $\mathcal{T}^c$ corresponds to a root-to-leaf path in $\mathcal{T}$.
Therefore, if we index a set of strings $\mathcal{S}$ with $\mathcal{T}^c$, each node in $\mathcal{T}^c$ corresponds to a string in $\mathcal{S}$ and the number of nodes in $\mathcal{T}^c$ is exactly equal to the cardinality of $\mathcal{S}$.
The height of the path decomposed trie is surely not larger than the height of the trie $\mathcal{T}$. $\mathcal{T}^c$ has different properties according to the policy adopted when choosing the decomposition paths (i.e., the node-to-leaf paths hanging off the currently chosen one). In the following we describe two of these strategies underlying their consequences on the path decomposition of $\mathcal{T}$.

Leftmost path. *We always choose the leftmost child in path decomposition. If the leftmost path policy is used in path decomposition, the depth first order of the nodes in $\mathcal{T}^c$ is equal to the depth first order of their correspondent leaves in $\mathcal{T}$. Hence, if $\mathcal{T}$ is lexicographic ordered, so $\mathcal{T}^c$ is.*

Heavy path. *Always choose the child who has the most leaves (arbitrary breaking ties). If this kind of strategy is used, the height of the path decomposed trie is bounded by $O(\log |S|)$.*

The decomposition obtained with the leftmost path policy is called *lexicographic path decomposition*, while the one obtained via the heavy path strategy is called *centroid path decomposition*.
When we implement a string dictionary with a path decomposed trie, the leftmost path strategy ensures that the indexes returned by Lookup (and the strings returned by Access) are lexicographic, but incurring in no guarantees on the height of the decomposed trie, which anyway can be no higher than the original trie.
Instead, heavy path ensures logarithmic guarantees on the height of the trie hence it is advantageous when the order of the indexes is not significant.

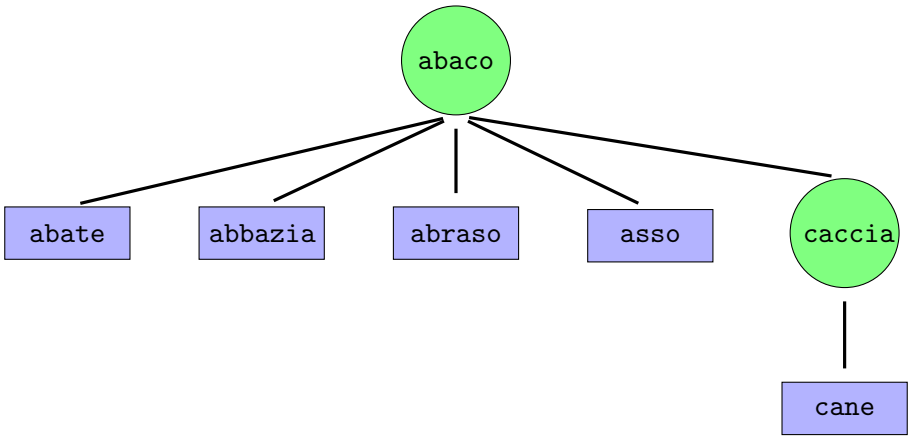


Figure 2.6: Path decomposition trie obtained via lexicographic decomposition of the trie in Figure 2.3a.

Chapter 3

Compressing string dictionaries

In this chapter we deal with the problem of representing a dictionary of strings (see Section 2.8) in a compressed fashion. In particular, in Section 3.1 we will show the methods used to *represent* the dictionary data, i.e., the strings, while in Section 3.2.1 we will illustrate how to store the additional information needed to implement Lookup and Access operations on such a dictionary.

3.1 Dictionary representation

In this section we expose the algorithms exploited in order to compress the strings of the dictionary. All of them suppose that the set of strings are lexicographically ordered so that the longest common pieces of information are shared between two consecutive strings. Moreover, we suppose that the ordered strings are stored *contiguously* on disk because all the encoders reconstruct any string of the dictionary scanning the ones that precede it according to lexicographic order. In fact, we know from the external-memory model (Section 2.6) that accessing contiguous memory locations yields to great benefits in terms of time.
Section 3.1.1 describes *Front coding*, which is the backbone of all our encoding schemas, even though it does not ensure optimal performance when decoding a generic string of the dictionary. In Section 3.1.2 we describe another algorithm: *Front coding with bucketing* or *Bucket coding*, which is nowadays the most used dictionary compression algorithm, enabling good empirical performance in decoding but not any clear theoretical bounds on space and time. Section 3.1.3 contains the description of the *Locality preserving front coding* algorithm, which ensures *optimal* efficiency both in space occupancy and in decompression time. Lastly, in Section 3.1.4 we illustrate a method that, inspired by locality preserving front coding, allows to encode a set of string ensuring the optimal access to any *prefix* of the string set, yet ensuring optimal space.

3.1.1 Front coding

One of the most effective but also intuitive way to compress an array of lexicographically ordered strings is to exploit their common prefix.
The ordering of the strings will likely make happen that if we pick any string at random, the ones that succeed it will probably share some of its first characters. We know that any string S_i of the dictionary can be though as composed by two parts: the first p characters are the ones that are shared with at least one of the strings in the dictionary; the last $|S_i| - k$ characters are not shared with any other string and they are called the *characteristic suffix* of S_i. Indeed, if the strings are ordered, p is equal to the number of characters that S_i shares with S_{i-1} or S_{i+1}, so we do not need to find the strings sharing the longest common prefix with S_i in all the dictionary.
Therefore, we can represent any string in a dictionary as composed by a numerical part encoding the length of the shared prefix and by a literal part representing the its characteristic suffix. For instance, if we examine the following small set of strings

$$\{\texttt{abaco, abate, abbazia, asso, casa}\}$$

we may compress it with the method mentioned above, obtaining

$$\texttt{(0,abaco),(3,te),(2,bazia),(1,sso),(0,casa)}$$

or *equivalently*, starting from right and proceeding to the left

$$\texttt{(3,co),(2,ate),(1,bbazia),(0,asso),(0,casa)}$$

This compression method is called *front coding* (shortly FC) and the encoding of a dictionary $\mathcal{D}$ obtained with this algorithm is denoted $\mathsf{FC}(\mathcal{D})$. Front coding is a well known and widely used method to encode an ordered set of strings, it was introduced by Witten and al. in 1999 [39].
Note that the total number of characters and the set of integers are the same in both the examples; actually, this property holds for the front coding of any dictionary. Conventionally, in this work we assume to front code any dictionary starting from the first string according to lexicographic order. Formally, we may define the front coding of a dictionary $\mathcal{D} = \langle s_1, s_2, ..., s_k \rangle$ as the sequence

$$\mathsf{FC}(\mathcal{D}) = \langle (l_1, \hat{s}_1), (l_2, \hat{s}_2), ...(l_k, \hat{s}_k) \rangle$$

where l_i for $i = 1, ..k$ represents the length of the prefix that string s_i shares with s_{i-1}: lpc(i,i-1)[1], while $\hat{s}_i$ is the remaining suffix of s_i, i.e., its last $\mid s_i \mid - l_i$ characters.
Also note that we do not store only the characteristic suffix of each string, otherwise we would lose some symbols, anyway we are sure that the characters belonging to the longest common prefix of each string are stored just *once*. So, the following observation holds.

[1] See section 2.4

Observation 2. *Given an ordered set of string S and its encoding $\mathsf{FC}(S)$ obtained by front coding, then $\forall s \in S$, if p is the longest common prefix of s among all the strings of S, we have that if $a \in p$, there exists one and only one suffix $\hat{s}$ in $\mathsf{FC}(S)$ such that $a \in \hat{s}$.*

Obviously, using this kind of technique, we have a gain in compression in the case the binary representation of the numerical part does not exceed the size of the binary representation of the shared prefix. For example, if we represent the integers of the dictionary above as 4 Byte integers, the compressed dictionary would actually require more space than the not compressed one.[2] For further understanding, an estimation of the storage cost of this algorithm has been done in [14]. In that work it has been introduced the concept of *Trie size* of any dictionary $\mathcal{D}$ as

$$\mathsf{Trie}(\mathcal{D}) = \sum_{i=1}^{k} |\hat{s}_i| \tag{3.1}$$

which stands for the total number of characters front coding emits as output. Moreover, the authors of the paper devised a lower bound $\mathsf{LT}(\mathcal{D})$ that is valid for any dictionary of strings. In particular $\mathsf{LT}(\mathcal{D}) = \mathsf{Trie}(\mathcal{D}) + \log \binom{\mathsf{Trie}(\mathcal{D})}{t-1}$, where t is the the number of nodes in the compacted trie built on $\mathcal{D}$[3].
In order to have a clearer understanding of the quantity $\log \binom{\mathsf{Trie}(\mathcal{D})}{t-1}$, in [15] it has been proved that $\log \binom{\mathsf{Trie}(\mathcal{D})}{t-1} = o(\mathsf{Trie}(\mathcal{D})) + O(K)$.
As a final result, in [14], the authors found also that the space occupied by a front coded ordered set of strings could be put in the following range.

$$\mathsf{LT}(\mathcal{D}) \leq \mathsf{FC}(\mathcal{D}) \ \leq \mathsf{LT}(\mathcal{D}) + O(K \log \frac{N}{K}) \tag{3.2}$$

Where N is the total number of characters in $\mathcal{D}$.
In order to estimate the size of a file[4] produced by front coding, we can sum the average suffix length of each string, we call this number $E[\hat{s}]$. If we multiply $E[\hat{s}]$ with the number of strings K we get the total amount of characters outputted by FC, which is $\mathsf{Trie}(\mathcal{D})$.

$$\mathsf{Trie(D)} = KE[\hat{s}]$$

If we call I the sequence of the binary representations of integers outputted by FC, which we will see can have different encodings, we can get an estimation of the size of a file generated by from coding using the following formula.

$$|\mathsf{FC(D)}| = KE[S] + |I| \tag{3.3}$$

[2] Assuming each character is encoded with 1 Byte.
[3] See Figure 2.3a to see an example of compacted trie built on a dictionary.
[4] A file can be thought as a binary representation.

Supposing that we have a way to access the position of any pair in the dictionary compressed by front coding, the time to decode a randomly accessed string in such a dictionary is directly proportional to the density of the not compressed strings (the ones that have $l_i = 0$, i.e., the ones that do not share characters with their previous string). In fact, once we have had access to the position of the i-th encoded string, if it has $l_i > 0$, we need to decode its previous string s_{i-1} in the dictionary in order to reconstruct s_i. s_{i-1} could in turn be encoded, forcing us to repeat this proceedings up to the first string that has been inserted uncompressed in the dictionary. Therefore, the time to access a string in a dictionary compressed in such a way could be not negligible at all, and in the worst case could force us to scan the entire string set.[5]

Rear coding *Rear coding* (shortly RC) encodes the set of strings in a way that is very similar to front coding. The suffixes are exactly the same that FC would have outputted, but instead of pairing each suffix with the length of the longest common prefix for that string, we put for the i-th string s_i the value $|si - 1| - \mathsf{lpc}(i, i-1)$, that is the number of characters we have to *discard* from s_{i-1} in order to get the longest common prefix between s_i and s_{i-1}. Formally, calling $l_i = \mathsf{lpc}(i, i-1)$ we have that

$$\mathsf{RC}(\mathcal{D}) = \langle (0, s_0), (|S_0| - l_1, \hat{s}_1), (|s_1| - l_2, \hat{s}_2), ..., (|s_{n-2}| - l_{n-1}, \hat{s}_{n-1}) \rangle$$

Therefore, if we encode the same set of strings previously encoded via FC with rear coding we would get the following.

```
(0,abaco),(2,te),(3,bazia),(6,sso),(4,casa)
```

For simplicity, in this work we will not make any distinction between front coding and rear coding. We will denote both the algorithm with FC and we will use the one or the other according to practical convenience. Indeed the only difference between these two algorithms resides in the numerical part of the pairs and both the techniques can be exploited in any of the compression algorithms exposed in the following subsections, which instead modify the number of not compressed strings.

3.1.2 Front coding with bucketing

Front coding with bucketing or *Bucket coding*, is a *parametric* compression algorithm, denoted with $\mathsf{BC}(\mathcal{D}, x)$ $x \in \mathbb{N}$, $\mathcal{D} \in \Sigma^{**}$ or $\mathsf{BC}(x)$ when the encoded dictionary is clear from the context, simply stating that there must be an uncompressed string every x. In this way, we never have to decompress more than x strings in order to reconstruct any string in the compressed dictionary (provided that we can access to all the uncompressed ones). Bucket coding is the easiest algorithm that deals with the number of fully copied strings in a compressed dictionary and it is also the most used, even though *it does not ensure any*

[5] In other words the time to decode a string in $\mathcal{D}$ is $\Theta(|\mathsf{FC}(\mathcal{D})|)$.

theoretical bound both on space and on time because it does not allow any control on the number of characters that must be scanned backwards to decode a string. For instance, we may have to compress a file which has mostly short strings and some very long ones, thus, we may stuff in a bucket some short strings and many very long ones. In order to reconstruct a short one, call it s_i we may have to scan one or more long strings entirely, preventing to reconstruct s_i in $O(|s_i|)$ I/Os.
Also note that the scan of a bucket of x strings produced by $\mathsf{BC}(x)$ might be increased in time complexity from $O(x)$ to $O(x^2)$ because of the decompression of that block. In fact, if we take the following sequence of strings.

```
(a,aa,aaa,...)
```

which is front coded as

```
((0,a),(1,a),(2,a),(3,a),...)
```

There are $\Theta(x)$ pairs in this block, which represent $\Theta(x)$ strings whose total length is $\sum_{i=0}^{x} \Theta(i) = \Theta(x^2)$ characters. Despite these pathological cases, in practice the space reduction consists of a constant factor so the time increase incurred by a block scan is negligible.
Overall this approach introduces a time/space trade-off driven by the parameter x. As far as time is concerned we can observe that the bigger is x, the better is the compression but the higher is the decompression time; conversely, the smaller is x, the faster is the decompression, but the worse is the compression because of a larger number of fully-copied strings. Moreover, the lengths of the uncompressed strings are *unbounded* hence, the compression made by BC can be ineffective.
Bucket coding allows to argue the number of copied strings C_s directly from the parameter x and from the total number of string K with the following formula.

$$C_s = \left\lfloor \frac{K}{x} \right\rfloor \tag{3.4}$$

While, if we consider the subset of the uncompressed strings produced by $\mathsf{BC}(\mathcal{D}, x)$, that is $\{U_0, U_1, ..., U_j, ..., U_{C_s-1}\} \subseteq \mathcal{D}$, we may notice that $U_j = \mathcal{D}_{j \cdot x}$.

3.1.3 Locality preserving front coding

Locality preserving front coding, denoted with $\mathsf{LPFC}(\mathcal{D}, x)$ $\mathcal{D} \in \Sigma^{**}$, $x \in \mathbb{N}$, proposed by Bender *et al.* in [2], is another parametric compression algorithm that deals with the number of uncompressed strings in a dictionary. The best feature of this algorithm is that it provides a controlled trade-off between space occupancy and the time to decode a string. The underlying algorithmic idea of locality preserving front coding is the following: *a string is front-coded only if its decoding time is proportional to its lengths, otherwise it is written uncompressed.* The outcome in time complexity is clear: we compress only if decoding is

optimal. Actually, this "constant of proportionality" controls also the space occupancy of the dictionary compressed with such an algorithm. Formally, suppose that we have front coded the first i strings $(s_0, ..., s_{i-1})$ into the compressed sequence
$F = \langle (0, \hat{s}_0), (l_1, \hat{s}_1), ..., (l_{i-1}, \hat{s}_{i-1}) \rangle$; we want to compress s_i so that we have to scan backwards at most $x|s_i|$ characters of F. So, we check wether $\sum_{j=0}^{i-1} |\hat{s}_j| \leq x|s_i|$. If this happens, we front code s_i into $(l_i, \hat{s}_i)$ and we append this pair to the sequence F; otherwise s_i is copied uncompressed and we append the pair $(0, s_i)$ to F.
Surprisingly, the strings that are left uncompressed, and were instead compressed by the classic front-coding scheme, have a length that can be controlled by the means of the parameter x, as it is shown in the following.

Theorem 6. *Locality preserving front coding* $\mathsf{LPFC}(\mathcal{D}, x)$ *takes at most* $(1+\epsilon)\mathsf{FC}(\mathcal{D})$ *space, and supports the decoding of any dictionary string* s_i *in* $O(\frac{|s_i|}{\epsilon B})$ *optimal I/Os. Where* $\epsilon = \frac{2}{x-2}$ *and* B *is the page size according to external memory model.*

It is very noteworthy the close relation between the *time* constant x which express the maximum number of characters to scan a string s_i and the *space* constant ϵ that provides a bound to the space occupied by $\mathsf{LPFC}(\mathcal{D}, x)$.

So, locality-preserving front coding is a compressed storage scheme for strings that can substitute their plain storage without introducing *any asymptotic slowdown* in the accesses to the compressed strings. In this sense it can be considered as a sort of *space booster* for any string indexing technique.

Since it will be useful in the next chapters, in the following lines we describe how to get the number of copied strings C_s produced by $\mathsf{LPFC}(\mathcal{D}, x)$.
Suppose that the average string length of the dictionary is $E[s]$, and suppose that the average length of a suffix is $E[\hat{s}]$, that is the average number of characters which each string does not share with the preceding one.
Once we have inserted a copied string, the algorithm will emit on average $E[\hat{s}]$ characters for each appended one. Hence, we can estimate the number of compressed strings between two copied strings S_c as follows.

$$S_c = \left\lfloor \frac{(x-1)E[s]}{E[\hat{s}]} \right\rfloor \tag{3.5}$$

On average we have one copied string every $S_c + 1$. If we know that the number of strings in the dictionary is K, we can evaluate the number of copied strings C_s with the following.

$$C_s = \left\lfloor \frac{K}{S_c + 1} \right\rfloor \tag{3.6}$$

Note that the average length of the suffixes of a dictionary $\mathcal{D}$ can be calculated empirically

by applying $\mathsf{FC}(\mathcal{D}) = \mathsf{LPFC}(\mathcal{D}, \infty)$[6] and then computing the average length of the literal part of each string outputted by the algorithm.

3.1.4 Optimal prefix retrieval

In the previous section we presented a solution which is optimally compressed and still supports the decoding of every string with optimal I/Os.
The only issue with that solution is that it does not guarantee optimal decompression time for *prefixes* of the strings, in fact, the decompression of a prefix p of a string s_i may cost up to $\Theta((1 + \frac{1}{\epsilon})|s_i|)$, rather than $\Theta((1 + \frac{1}{\epsilon})|p|)$ I/Os.
In order to acquire the optimal decompression time for any prefix of any string in the sorted set $\mathcal{D}$ we modify locality preserving front coding according to a method reported in [15], in the following way.
We construct the superset $\hat{\mathcal{D}}$ of $\mathcal{D}$ which contains *all the possible prefixes* of all the strings in the ordered set. $\hat{\mathcal{D}}$ is actually another ordered set of strings: the one which is obtained by the DFS[7] visit of the trie associated to $\mathcal{D}$.[8] Figure 3.1 shows an example of such a visit.

We call $\mathcal{T}_{\mathcal{D}}$ the trie associated to $\mathcal{D}$; we assume that its nodes are numbered according to its DFS visit. Any node u in this trie is associated with a label $\mathsf{label}(u)$, which is the string of variable length on the edge $(p(u), u)$, where $p(u)$ is the father of u in $\mathcal{T}_{\mathcal{D}}$.
Note that any node u uniquely identify a string: $\mathsf{string}(u)$, obtained by juxtaposing the labels on the edges of the path from the root of the trie to u. $\mathsf{string}(u)$ is actually the prefix of *all* the strings $\mathsf{string}(v)$ associated to the nodes v descending from u.
It is possible to compress the superset $\hat{\mathcal{D}}$ with locality preserving front coding which assures optimal decoding time for any strings belonging to $\hat{\mathcal{D}}$, hence to any prefix of any string belonging to $\mathcal{D}$.
In order to prove an important property of $\hat{\mathcal{D}}$, we recall that the quantity $\mathsf{Trie}(\mathcal{S})$, that is related to any dictionary $\mathcal{S}$ having $\mathcal{T}$ as associated trie is the following.

$$\mathsf{Trie}(\mathcal{S}) = \sum_{u \in \mathcal{T}} l(p(u), u)$$

In practice, $\mathsf{Trie}(\mathcal{S})$ is the sum of the lengths of the edge labels in $\mathcal{T}$. The following observation is crucial.

Observation 3. *Given a set of string $\mathcal{S}$ and its superset $\hat{\mathcal{S}}$ obtained by the DFS-visit of its associated $\mathcal{T}$, we have that*

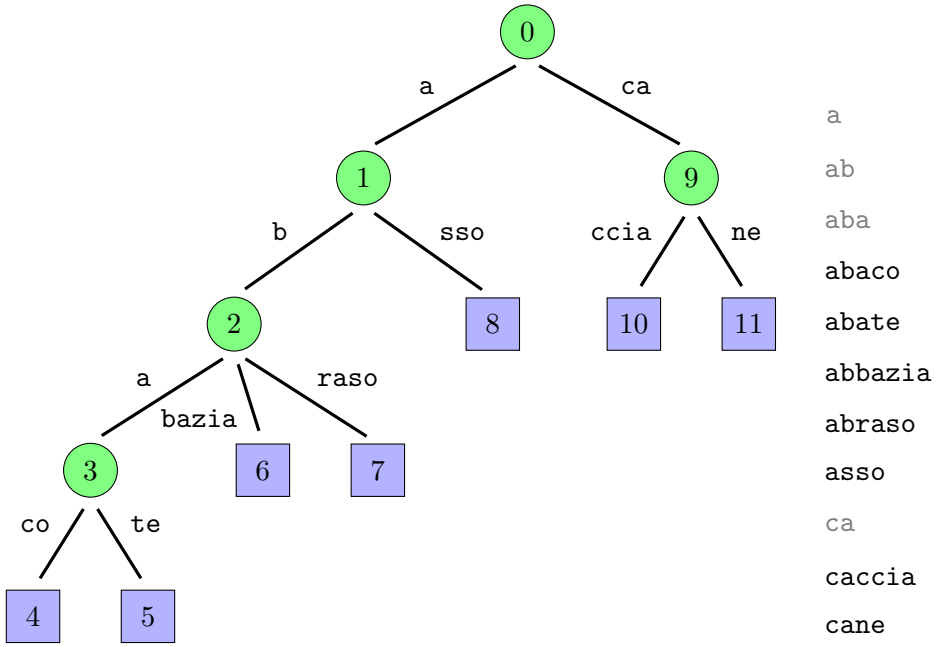
$$\mathsf{Trie}(\hat{\mathcal{S}}) = \mathsf{Trie}(\mathcal{S}) \tag{3.7}$$

[6] Obviously we may pass as parameter of LPFC the total number of characters of the dictionary instead of ∞.

[7] DFS stands for *Depth First Visit*; it is an algorithm for traversing a tree starting from the root and exploring as far as possible along each branch before backtracking.

[8] See Section 2.12 for the definition of a Trie associated to a set of strings.

Figure 3.1: Depth-first visit of the trie associated to dictionary $\mathcal{D}$ ={`abaco, abate, abbazia, abraso, asso, caccia, cane`}. A node with label n is the n-th node visited by DFS.



The observation above implies that the sum of the lengths of the literal parts outputted by $\mathsf{LPFC}(\mathcal{S}, \mathsf{x})$ is the same of the one produced by $\mathsf{LPFC}(\hat{\mathcal{S}}, \mathsf{x})$, hence

$$|\mathsf{LPFC}(\hat{\mathcal{S}}, \mathsf{x})| = (1+\epsilon)|\mathsf{FC}(\mathcal{S})| + \mathsf{O}(|\mathcal{S}|)$$

because the number of prefixes of $\mathcal{S}$ is bounded by the number of internal nodes of $\mathcal{T}$, that are $O(|\mathcal{S}|)$.
So, if we take as example the set of strings taken in Figure 3.1, we have that the front coding of such a set is the following

```
(0,abaco),(3,te),(2,bazia),(2,raso),(1,sso),(0,caccia),(2,ne)
```

while the front coding of the set procured by its DFS-visit is the following.

```
(0,a),(1,b),(2,a),(3,co),(3,te),(2,bazia),(2,raso),(1,sso),(0,ca),(2,ccia),(2,ne)
```

It is easy to note from the example above that the sums of the lengths of the literal parts of the pairs are equal to 27.

Superset creation In this paragraph we discuss how to build up the superset $\hat{\mathcal{S}}$ from the sorted set of strings $\mathcal{S}$.
In order to obtain such a superset, we need to visit the trie associated to $\mathcal{S}$ in Depth-first search (DFS) order, because it is the visiting order that ensures the lexicographic ordering of the strings associated with each node u of the trie, namely $\mathsf{string}(u)$.
The trie associated to $\mathcal{S}$: $\mathsf{Trie}(\mathcal{S})$ is too big to be created and visited directly, so, we devised a way to just *simulate* the DFS visit of such a trie, without the need to allocate the huge amount of memory needed to store it. As we will explain later, first of all, in order to perform such a visit, we need to build up what is called the *longest common prefix array* of $\mathcal{S}$, which we denote $\mathsf{LCPA}(\mathcal{S})$.
This array simply stores in position i the longest common prefix between $\mathcal{S}_{i+1}$ and $\mathcal{S}_i$.
To build up LCPA we just have to scan the whole set of sorted strings checking which is the maximum number of shared characters between each pair of consecutive strings. Actually, the time complexity for instantiate such an array is not directly proportional to the the total number of characters in the string set, which we call N, but to the total number of shared characters C.
Of course $C < N$, and $C = \sum_{i=0}^{\mathcal{S}-1} |\mathsf{lcp}(i+1, i)|$, so, we can stop the scanning of each string as soon as we find a mismatch character.
Starting from the longest common prefix array, we devised a recursive algorithm able to output the strings associated to each node of $\mathcal{T}_{\mathcal{S}}$, which is illustrated in Algorithm 1.

Algorithm 1 Trie DFS visit simulation

```
1: S[] = input sorted set of strings
2: T[] = output set of strings
3: LCPA[] = lcp-array
4: i = 0, j = length of LCPA[]
5: procedure DFS VISIT SIMULATION(i,j,S[],T[],LCPA[])
6:     if i==j then
7:         Append S[i] to T[]
8:     else
9:         newi = index of the leftmost minimum element in LCPA[i+1,j]
10:        if LCPA[newi] > LCPA[i] then
11:            Append LCP(S[newi],S[newi-1]) to T[]
12:        end if
13:        Call DFS visit simulation(i,newi-1,S[],T[],LCPA[])
14:        Call DFS visit simulation(newi,j,S[],T[],LCPA[])
15:    end if
16: end procedure
```

Analysing this algorithm, we can realise that it outputs all the prefixes of all the strings

in the dictionary $\mathcal{S}$, and the strings themselves in lexicographic order. Considering that $\mathsf{LCP}(s_k, s_r)$, with $s_k, s_r \in \mathcal{S}$[9] returns the longest common prefix between s_k, s_r, for each range delimited by the couple (i, j), the algorithm outputs the *leftmost* shortest prefix in that range, which is in position newi and whose value is stored in $\mathsf{LCPA}[\mathsf{newi}]$. Then, the algorithm recurs calling itself in the range $i, \mathsf{newi}-1$, hence outputting the leftmost shortest prefix among the strings from s_{i+1} to s_{newi}. As final step, it recurs again to output the leftmost shortest prefix among the strings from s_{newi} to s_j. The recursion stops when the range extrema coincide: $i = j$. At that point, the algorithm has already outputted all the possible prefixes of s_i and it must output string s_i itself.
The complexity of the algorithm is actually given by the search for the leftmost minimum element in the various ranges and by the number of characters we need to "copy" in the output buffer T. For the latter, we can do nothing, since that quantity is exactly the total number of characters we need to output, the former instead can be optimised in order to avoid a linear scan of the longest common prefix array from position i to position j at each recursive step, which would make the overall time complexity become $O(|\mathcal{S}|^2)$, absolutely unacceptable for large dictionaries.
So, an efficient implementation of line 9 of Algorithm 1 is important. Fortunately, finding the minimum element among a collection of them is a very diffused and studied problem, commonly called Range minimum query (RMQ).[10] The solution we have adopted in our implementation allows to answer RMQ in *constant time* with the help of a data structure commonly called *Cartesian tree* that occupies $O(|\mathcal{S}|)$ space. The description of the RMQ solution adopted can be found in [23] and its implementation in the *Succinct library* by Ottaviano [33]. This data structure allows us to build up $\hat{\mathcal{S}}$ in $O(|\mathcal{S}|)$ time and space, which is optimal.
Moreover, we may modify algorithm 1 to reduce the number of recursive calls, for example outputting the strings in their lexicographical order when we have a series of equal consecutive values in the LCPA.
The improved version of Algorithm 1 is illustrated by Algorithm 2 located in the following page.

[9] While $\mathsf{lcp}(i, j)$ outputs the length of $\mathsf{LCP}(s_i, s_j)$.
[10] See Section 2.9.

Algorithm 2 Trie DFS visit simulation with reduced number of recursive calls

```
1: S[] = input sorted set of strings
2: T[] = output set of strings
3: LCPA[] = lcp-array
4: i = 0, j = length of LCPA[]
5: procedure DFS VISIT SIMULATION(i,j,S[],T[],LCPA[])
6:     if i==j then
7:         Append S[i] to T[]
8:     else
9:         newi = index of the leftmost minimum element in LCPA[i+1,j]
10:        Append LCP(S[newi],S[newi-1]) to T[]
11:        Call DFS visit simulation(i,newi-1,S[],T[],LCPA[])
12:        while newi <j AND LCPA[newi+1] == LCPA[newi] do
13:            Append S[newi] to T[]
14:            newi = newi+1
15:        end while
16:        Call DFS visit simulation(newi,j,S[],T[],LCPA[])
17:    end if
18: end procedure
```

Calling $\mathsf{newi} = \mathsf{RMQ}(i+1, j)$, lines from 12 to 15 of Algorithm 2 output all the strings which follow s_{newi} that share the same prefix, and s_{newi} itself. Those strings are outputted after all the prefixes and all the strings that lexicographically precede s_{newi} have been already put in the output buffer and before all the prefixes and the strings that are lexicographically greater than them. The last two statements are guaranteed by the procedure callings in line 13 and 18.
Note that in this version of the algorithm we need no more the "if" statement in line 10 of algorithm 1 because the "while" related code (lines 13-18) ensures to take care of blocks of consecutive identical lcps in LCPA.
In fact, without the check in line 10 on algorithm 1 it may happen that we output the same prefix that the algorithm has already outputted during the preceding recursive call in the case that we have a sequence of local minima after the position returned by the range minimum query.
In Table 3.1 we show the execution of algorithm 1 on a very simple sorted set of strings. We may notice that visiting the trie in Figure 3.1 in DFS order we obtain the same set of strings outputted by the algorithm, shown in table 3.1 (b). Lastly, Figure 3.2 shows the implementation of Algorithm 2, written in C++.

```
1  void TrieVisitUtil(const uint * lcpArray, succinct::cartesian_tree &t,
       ↪ succinct::elias_fano &E,const char *f, const std::ofstream *output,
       ↪ uint64_t i,uint64_t j ){
2  uint64_t si, si_end, newi;
3          if(i == j){
4                  si = E.select(i);
5                  si_end = E.select(i+1);
6                  string a (&f[si],si_end-si-1) ;
7                  *output << a.append("\n");}
8          else {
9                  //find the index of the  minimum lcpArray
10
11                 newi = t.rmq(i+1,j);
12
13                 if (lcpArray[newi] != 0 && lcpArray[newi] > lcpArray[i]){
14                         *output << EmitString(newi,lcpArray,E,f)<< "\n";
15                         }
16                 TrieVisitUtil(lcpArray,t,E,f,output,i,newi-1);
17                 while(lcpArray[newi+1] == lcpArray[newi] && newi+1<=j){
18                         si = E.select(newi);
19                         si_end = E.select(newi+1);
20                         string a (&f[si],si_end-si-1);
21                         * output << a.append("\n");
22                         newi = newi+1;
23                 }
24                 TrieVisitUtil(lcpArray,t,E,f,output,newi,j);
25         }
26 }
```

Figure 3.2: Actual implementation of Algorithm 2. The positions of the strings are stored in the Elias-Fano bitvector E (See section 2.7). The function EmitString(i,...) simply returns the fist $\mathsf{LCPA}[i]$ characters of the $i-th$ string in the set of strings

3.2 Storing additional information

In this section we deal with the problem of storing efficiently the additional information needed to access any string of a dictionary compressed by the algorithms described in the previous section. We have seen that all those algorithms store some strings compressed and others uncompressed and that we can not decode a compressed one without having access to its closest uncompressed string U_j. Therefore, we need some information to access such a string and some others to decode the strings appearing after U_j. Some of both these kinds of information can be stored together with the representation of the strings while some others must be stored separately. The information stored *outside* the memory area dedicated to the representation of the strings are treated in Section 3.2.1, whilst the ones blended with the strings are discussed in Section 3.2.2.

Table 3.1: Simulation of the execution of Algorithm 1 on a string set

Set of strings:	abaco	abate	abbazia	abraso	asso	caccia	cane
lcp-array:	0	3	2	2	1	0	2
indexes:	0	1	2	3	4	5	6

(a) Description of the input string set with the associated lcp-array

Call depth	i	j	RMQ(i+1,j)	Left Range	Right range	Emitted
1	0	6	5	[0,4]	[5,6]	ϵ
2	0	4	4	[0,3]	[4,4]	a
3	0	3	2	[0,1]	[2,3]	ab
4	0	1	1	[0,0]	[1,1]	aba
5	0	0	/	/	/	abaco
5	1	1	/	/	/	abate
4	2	3	3	[2,2]	[3,3]	ϵ
5	2	2	/	/	/	abbazia
5	3	3	/	/	/	abraso
3	4	4	/	/	/	asso
2	5	6	6	[5,5]	[6,6]	ca
3	5	5	/	/	/	caccia
3	6	6	/	/	/	cane

(b) Execution of the algorithm on the input described in table (a), the output set can be read by reading the last column from top to bottom.The symbol ϵ stand for the empty string.

3.2.1 Storing the references to the strings

In this part of our work we tackle the problem of storing the data enabling the first access to the compressed dictionary $\mathfrak{C}(\mathcal{D})$, where $\mathfrak{C}$ is any of the compression algorithms illustrated in Section 3.1. If we call $U \subseteq \mathcal{D}$ the set of fully-copied strings outputted by $\mathfrak{C}$, we have that in order to implement Access(i)[11] we need to retrieve the *position* of $U_j = \mathcal{D}_k$ such that $k \leq i$ and $U_{j+1} = \mathcal{D}_s$ is such that $s > i$. In other words, we need to retrieve the position of the encoding of the first uncompressed string preceding the i-th in $\mathfrak{C}(\mathcal{D})$.
Typically, those information are stored separately from the dictionary and increase the time to perform Access(i) of at least a constant number of I/Os, nevertheless, they are fundamental because without them we would have to scan the compressed dictionary from the beginning whatever is the position of the string to be decoded. In the following paragraphs, we will denote the total number of strings of the dictionary with n while the total amount of characters will be indicated by N.

[11] See Section 2.8.

Array of string pointers The simplest solution to have access to any string of a dictionary $\mathcal{D}$ is to store the pointers of the starting character of each string in an array of pointers called $A[0, n-1]$. We denote the fact that the i-th cell of A stores the position of the $i-$th string with $A[i] \rightarrow \mathcal{D}_i$, or $A[i] = \&(s_i[0])$ where & is the symbol, inspired by the C programming language, indicating the function returning the virtual address of an abstract memory location.
For now, we assume that each cell of the array A is w bits long with $w = \Omega(\log N)$, otherwise we may not have references of the strings whose address is too long. Typically w is 4 or 8 Bytes in modern machines. Moreover, we would need another array, $B[0, n-1]$ such that $B[i]$ stores the index of the first uncompressed string U_j appearing before $\mathcal{D}_i$. After all, assuming we compress the dictionary with locality preserving front coding, exploiting arrays A and B we would need $O(1)$ I/Os to access the position of U_j and $O(|\mathcal{D}_i|/\mathrm{B})$ I/Os to decompress the i-th string, which is optimal. Nevertheless, the total space would be $|\mathsf{LPFC}(\mathcal{D}, x)| + 2wn$, which would be not acceptable because the term $2wn$ would be the dominant one for many dictionaries. In fact, it is quite reasonable that in real world the average size of the suffixes outputted by LPFC is greater than $2w$. At the end we would need more space to index the strings than to store them.
In the following paragraphs we will expose some solutions that try to avoid this issue. It is also possible to point out that we may avoid to store the array B in case we compress the dictionary with FC or BC, anyway, both the solutions are not optimal because the decoding of a string would require $O(|\mathsf{FC}(\mathcal{D})|)$ I/Os.

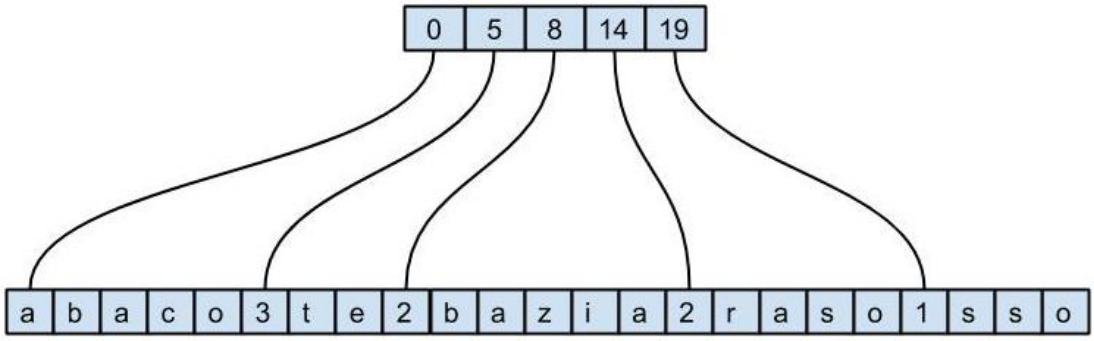


Figure 3.3: Array of pointers of $\mathsf{FC}(\langle$`abaco, abate, abbazia, abraso, asso`$\rangle)$

Succinct array of string pointers In this section we describe a solution, proposed in [15], which allows to efficiently perform a variant of the $\mathsf{Access}(i)$ operation, called $\mathsf{Retrieval}(i,l)$ which returns the first l characters of the string $\mathcal{D}_i$. In particular, this solution allows to represent the arrays A and B described in the previous paragraph in a succinct

way, reducing the space occupancy of the indexing data structure of the dictionary.
Now we call R the binary representation of the output of the dictionary compressed via the locality preserving front coding, which we remind it is a sequence of couples composed by an integer and by the suffixes of the strings in $\mathcal{D}$. At this point, we define the following binary arrays:

1. The binary array $\mathrm{E}[0, |\mathrm{R}| - 1]$ which has a bit set to `1` in correspondence of the starting of the encoding of some string $\mathcal{D}_i$. That is, if $\mathrm{E}[k] = \texttt{1}$ then, assuming that $\mathrm{E}[k]$ contains the i-th `1`, $R[k] = \mathcal{P}_i[0]$, where $\mathcal{P}_i$ is the binary representation of the i-th pair produced by the compression algorithm. k is exactly the index of the starting position of the i-th pair $(l_i, \hat{\mathcal{D}}_i)$ of the compressed dictionary. E contains n bits set to one and it is encoded via the Elias-Fano encoding, enriched with some bits providing the Select_1 operation[12], thus taking $2n + n \log(|\mathrm{R}|/n) + o(n)$ bits of space.

2. The binary array $\mathrm{V}[0, n-1]$ such that $\mathrm{V}[i] = \texttt{1}$ if $\hat{\mathcal{D}}_i = \mathcal{D}_i$, $\mathrm{V}[i] = \texttt{0}$ elsewhere; where $\hat{\mathcal{D}}_i$ is the suffix of the i-th string outputted by locality preserving front coding.
Some bits are added to V in order to provide Select_1 and Rank_1 operations, as proposed by Munro in [31], so that overall V is $n + o(n)$ bits long.

The space occupancy of our indexed ordered set of strings $\mathcal{D}$ is now equal to $(1 + \epsilon)\mathsf{FC}(\mathcal{D}) + O(n)$ bits, where the factor *O(n)* is here actually negligible thanks to the succinctness of E and V.

Now we have all what we need to implement efficiently $\mathsf{Retrieval}$*(i,l)*. In fact, a query $\mathsf{Select}_1(\mathrm{E}, i)$ gives us in *constant time* exactly the starting position of $\mathcal{P}_i$ in R. If the i-th string belongs to the set of the uncompressed ones, we are done, since it is enough to report the characters between $\mathsf{Select}_1(\mathrm{E}, i)$ and $\mathsf{Select}_1(\mathrm{E}, i+1)$.
In order to know wether $\mathcal{D}_i$ belongs to such a set, we perform a $\mathsf{Predecessor}_1(\mathrm{V}, i)$ that according to the definition given in Section 2.4 is equal to $\mathsf{Select}_1(\mathrm{V}, \mathsf{Rank}_1(\mathrm{V}, i+1))$.
Indeed, $\mathsf{Predecessor}_1(\mathrm{V}, i)$ returns i if the i-th string is uncompressed, or the index of the first uncompressed string preceding $\mathcal{D}_i$, call it j, if $\mathcal{D}_i$ is stored compressed.
Now we can reconstruct $\mathcal{D}_i$ starting from $\mathcal{D}_j$ by copying characters from R starting from the position in which $\mathcal{D}_j$ starts.
First of all, we put the characters belonging to $\mathcal{D}_j$ in a buffer B, then, for $u = j+1, ..., i$ we overwrite the last $m = |\mathcal{D}_{u-1}| - \mathsf{lcp}(\mathcal{D}_u, \mathcal{D}_{u-1})$ characters of B. Note that we always know the length of each string because of the bitvector E[13] and because of the encoded lengths of the shared prefixes.
We can avoid the calculation value m at each step of the algorithm simply modifying the locality preserving front coding in locality preserving *rear* coding, outputting, instead of the longest common prefixes, the value m described above.

[12] See Section 2.4.
[13] To get the position of the string after s_u it is enough to perform $\mathsf{Select}_1(\mathrm{E}, u+1)$.

This fact does not break the LPFC guarantees on the optimality of time in decoding because we do not modify the outputted suffixes $\hat{\mathcal{D}}_i$, therefore, reconstructing $\mathcal{D}_i$ does not need more than $O(|\mathcal{D}_i|)$ I/Os.

Having a look to reality, this solution is actually good with respect of storing the plain array of string pointers, which we remind it is wn bits long. Nowadays typically $w = 64$, which implies that we achieve gain in compression with our solution if

$$2n + n\log(\tfrac{|\mathrm{R}|}{n}) < 64n;\ 2 + \log(\tfrac{|\mathrm{R}|}{n}) < 64$$

The quantity $\frac{|R|}{n}$ can be upper bounded by $\frac{N}{n}$, which is equivalent to the average string length of the dictionary. Having noted this, it is quite straightforward to point out that it is quite difficult to find strings which are longer than 2^{62} bits (2^{59} characters!).
Moreover, while the number of strings remains constant augmenting the locality preserving front coding parameter, the outputted sequence of bits R shrinks, reducing in turn the dimension of the bitvector E.
The binary array V remains unchanged because it depends only by n; obviously it is much convenient in terms of space because $|\mathrm{V}| = n + o(n) < wn = |B|$.
Lastly, time efficiency is guaranteed because both the succinct data structures provides constant time Select and Rank on them.

Actually, we may study method to estimate a lower and upper bound relative to the storage of the Elias-Fano bitvector E.
We recall the formula for the size of an Elias-Fano bitvector is the following, calling $n = |\mathcal{D}|$ and $M = |R|$.

$$|\mathrm{E}| = \mathrm{E}(n, M) = 2n + n\lceil\log\tfrac{M}{n}\rceil + o(n) \approx 2n + n\log M - n\log n + o(n) \text{ bits}$$

Since the number of strings n remains invariant whatever the compression algorithm, we may notice that the size of the bit-vector E increases *logarithmically*, or equivalently
$\mathrm{E}(M) = \Theta(\log M)$ with M standing for the dimension of the compressed file. Thus, we get the minimum value for M when the size of the binary representation of the compressed dictionary is minimal, which happens when we compress it with FC. So, we may use formula 3.3 to estimate this number.
If we know the average suffix length for a single string $E[\hat{s}]$ (the number of characters which it does not share with the previous string), we have that the following holds.

$$nE[\hat{s}] + |I| < M \leq |B(\mathcal{D})|$$

Where I is the *binary representation* of the set of integers outputted by FC whilst $B(\mathcal{D})$ is the binary representation of $\mathcal{D}$.
At this point we have a lower and upper bound for M, hence we can evaluate the lower and upper bound for |E| when n is fixed as follows.

$$\mathrm{E}(nE[\hat{s}] + |I|) < \mathrm{E}(M) \leq \mathrm{E}(|B(\mathcal{D})|) \tag{3.8}$$

Storing only the positions of the copied strings with Elias-Fano We propose now a different way to store the positions of the strings that can be competitive with respect to the one described in the previous paragraph. In practice we keep stored in E only the positions of the *uncompressed strings* of $\mathsf{LCFC}(\mathcal{D},\mathrm{x})$, while the *lengths* of the suffixes are stored beside the encodings of their prefixes (or rears). The lengths are stored in such a position instead that in a separate array in order *to avoid one random I/O per* Access. The binary array V remains unchanged.
We recall that in order to implement $\mathsf{Access}(i)$ over this data structure we need to retrieve the index of the first copied string preceding the $i-th$.
So, we need to find the number of copied strings occurring before the $i-th$, we call this number v, which can be obtained by $v = \mathsf{Rank_1}(\mathrm{V}, i+1)$.
At this point, since E stores only the positions of the copied strings, we need a $\mathsf{Select_1}(\mathrm{E}, v)$ to retrieve the position of the v-th string in the compressed dictionary. Instead, to get the index of the v-th string, call it j, we just need to perform $j = \mathsf{Select}(\mathrm{V}, v)$. Note that j is needed because otherwise we would not know how many strings we have to scan before accessing the i-th one.
The number of operations to get the position and the index of the copied string closest to the i-th is the same of the version illustrated in the previous paragraph, but we can achieve some gain in time if the decoding of the lengths is faster than the Selects needed to get the positions of strings which are stored contiguously after the i-th.
Moreover we surely achieve some gain in space in the representation of E because we just have one bit set to `1` for each copied string, whilst in the previous version we had a bit set to `1` *for each string*. The gain can be relevant because the number of copied strings may be much lower than the number of the strings.
Recalling the formula to calculate the space occupied by an Elias-Fano bitvector.

$$\mathrm{E}(n, M) = 2n + n\left\lceil \log \tfrac{M}{n} \right\rceil + o(n)$$

We can express the following observation.

Observation 4. *If M is the size of the compressed file and n is the number of ones in the bitvector E, if $k < n < M$*

$$\mathrm{E}(k, M) < \mathrm{E}(n, M) < \mathrm{E}(M, M) \tag{3.9}$$

Proof. We assume for simplicity that the function E is continuous in $[0, M]$ thus, we get rid of the ceiling. At this point we calculate when the partial derivative of $\frac{d\mathrm{E}}{dn}$ is equal to zero.

$$2 - \tfrac{1}{\ln 2} + \log M - \log n = 0;\ \log n - 2 \approx \log M - 1.45;\ \log n \approx \log M + 0.55;\ n \approx 1.46M$$

We may argue that this point is a maximum from the fact that when $n < 1.46M$ the derivative is positive. Therefore if n goes from 0 to M, $\mathrm{E}(n)$ is monotonically increasing. It has been proved that reducing the number of ones to be inserted in our Elias-Fano bitvector of size M (which are at most M) reduces the space occupied by it. □

Figure 3.4 illustrates the number of bits occupied by by the Elias-Fano data structure when M is equal to 100.

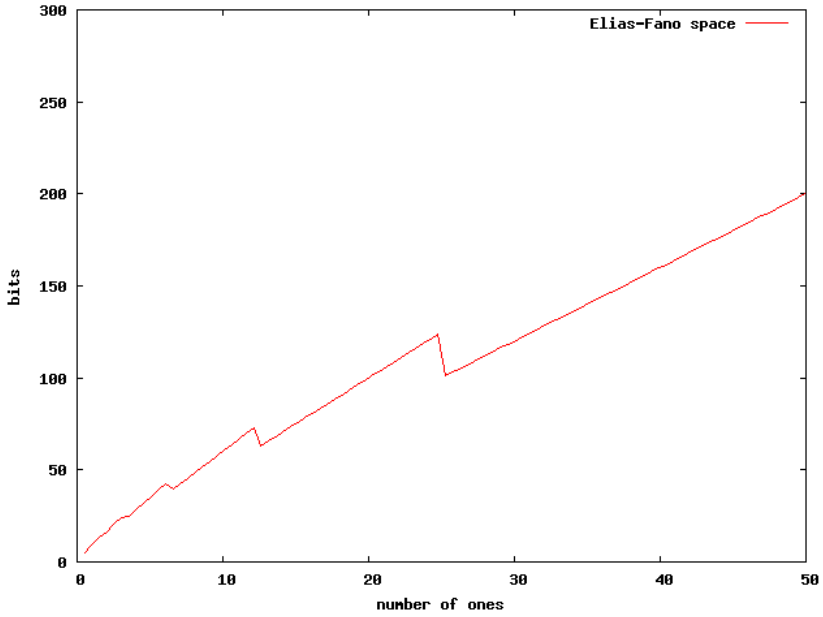


Figure 3.4: Number of bits occupied by an Elias-Fano bitvector with $M = 100$

Array of positions and array of indexes It could be also interesting to store the positions of the copied strings and the indexes of those, as *plain arrays*, without exploiting any succinct data structure.
Indeed, increasing the parameter x of $\mathsf{LPFC}(x)$, we decrease the number of copied strings, which can be very low when the parameter is high enough.
It could be so low that we may save both space and time. For instance, we know that to store the indexes of the copied strings with a bitvector V we need approximately $n + o(n)$ bits, where n is the number of strings, therefore $n/8 + o(n)$ Bytes.
If we store each cell of the array as a 4-Byte integer, we need $4m$ bytes to store such an array, where m is the number of the *copied strings*. Indeed it may happen that the space required to store V as a plain array is lower than the one required to store if succinctly.

$$4m < n/8 \ , \ m < n/32$$

So, we save in space if the number of copied strings is approximately less than one 32^{th} of the total number of strings (actually, the copied strings could be a bit more, since the bitvector requires a bit more than n bits).

We can also save in time because we can get the index of the copied string closest to the i-th one via a simple variant of binary search, which requires exactly at most $log(m)$ operations and I/Os. Since the Select and Rank on the bitvector require a constant amount of time, independent from the number of the copied strings, it may be faster to perform a binary search than those operations when the number of copied strings shrinks enough.
The *positions* of the copied strings can be stored in a plain array too. In this case it is quite difficult to achieve some savings in space because the Elias-Fano bitvector storing them is very sparse, very small and tends to reduce its space linearly as the number of copied strings decreases.
Anyway, we may save time because we would get rid of the initial Select on such a bitvector[14], substituting it with a simple Access performed on the plain array storing the positions.
Moreover, despite the fact that we hardly save space storing such positions as an array, in any case the loss in space would be negligible because again the size of the array depends on the number of copied strings, which can be very low. Indeed we expect that the size of this array will be negligible with respect to the size of the compressed dictionary.
In our implementation, we exploited the upper_bound function of the C++ standard library in order to find the index of the copied string preceding the i-th one in the array of indexes. The following listing shows its signature.

```
1 template <class ForwardIterator, class T>
2 ForwardIterator upper_bound (ForwardIterator first, ForwardIterator last,
    ↪ const T& val);
```

upper_bound returns an iterator[15] pointing to the first element in the range [first,last) which compares greater than val.
So, the following lines of code allow to find the index we are looking for.

```
1 std::vector<uint32_t>::iterator it = std::upper_bound(Copied_indexes.begin(),
    ↪ Copied_indexes.end(),i);
2 uint32_t key = it -1 -Copied_indexes.begin();
```

Where Copied_indexes is the vector storing the indexes of the copied strings, while Copied_indexes[key] is the searched index.
Note that in our case the iterator returned by upper_bound points always to the element immediately after the one we are looking for. In fact, independently of i belonging or not to Copied_indexes, upper_bound returns the iterator pointing to the *leftmost* element which is greater than i, that is, the index of the leftmost copied string greater than the i-th one. If we take the element preceding it in Copied_indexes, we get i if i belongs to Copied_indexes, or the index of the *rightmost* copied string smaller than the i-th one if i does not belong to Copied_indexes.
For example take the following array V = $[0, 2, 5, 7, 9, 10]$. We may perform

[14] Which is implemented with a constant number of random memory accesses.
[15] The concept of iterator is a generalisation of the concept of pointer. In this context it is possible to think an iterator simply as a memory area storing the address of another memory location.

$$\mathsf{upper\_bound}(L, 7) = 4$$

In this case 7 belongs to V and indeed $L[3] = 7$ is correct because to decompress the seventh string we have to start scanning from the seventh one, which is the fourth uncompressed string. Otherwise we may perform

$$\mathsf{upper\_bound}(L, 6) = 2$$

And in this case 6 does not belong to V but $L[2] = 5$ is still correct, in fact we decompress the sixth string starting from the fifth one because they are in the same block.

Combinations Until now, we proposed storage schemes that forced the memory layout of both the two data structure needed to implement Access on the compressed dictionary. Actually, this is not mandatory: we can mix the solutions illustrated until this point because storing the indexes and storing the positions are two separated problems.
For instance we can store V as a plain array and E as an Elias-Fano bitvector, which can index only the positions of the copied strings or the positions of all the strings.
On the contrary nothing stops us from storing V as a succinct bitvector and E as a plain array (even though in this case it does not make sense to store the position of all the strings in E).
In the following chapter, we are going to study the consequences of the applications of the various indexing methods to actual compressed dictionaries.

3.2.2 Integers encoding

Compressing a dictionary using the methods we have described until now requires some integers to be encoded: for example the prefixes or the rears respectively in Front coding or in Rear coding. If also the lengths of the suffixes are stored directly in the compressed file of strings, as proposed in Section 3.2.1, they must be encoded too.
There are actually two big families of integers encoders: the *bit-aligned* ones and the *byte-aligned* ones.
Depending on the distribution of the integers to be encoded, a bit-aligned encoder could be better than a byte aligned one or viceversa. Often in theory bit-aligned encoders would be a good and winning choice to encode integers, anyway, nowadays machines cannot index single bits, they can index only bytes. For this reason byte aligned encoders are the most used when the real target is performance, despite sometimes they are not as close to the optimum space occupation as bit-aligned ones.
One of the encoders that we used in our implementation is VBFast[16].VBFast is very fast in decoding, since it just have to scan the first two bits of the encoded number to decide how many bytes it has to scan after the first.

[16] See Section 2.10.2

The preamble bits allow us to use a "for" loop instead of the "while" that would be requested by a generic variable byte code. The only limitation of this algorithm is that we cannot represent integers higher than $2^{30} - 1$, which is actually not a limitation since it's quite impossible to find prefixes, suffixes or rears of such a length.
In general, since the strings to be compressed are *sorted*, subsequent strings tend to share a long prefix and to differ in few characters, thus encoding rears instead of prefix could be a good choice.
If we are sure that we will not encode numbers greater than $2^{15} = 32768$, which is plausible, we can modify a little VBFast reducing the preamble to one single bit. That would allow to encode much more lengths with one byte only, i.e., all the ones that would need 7 bits to be binary represented. In this work, we will call the version of VBFast with two preamble bits VBFast(32), and the version which requires just one preamble bit VBFast(16).

Another possibility is to use a bit-aligned coder with padding, so that the compressed dictionary is kept Byte-aligned. Such a scheme can be particularly useful if we encode the lengths of the rears and of the suffixes one after the other consecutively in memory (therefore E must index only the positions of the copied strings). In fact, in case of byte aligned encoders, we always need at least two Bytes to encode two integers, while in this case one can be enough.
Our Gamma-encoder, which we call Gamma Padding (shortly GP) is a variant of Elias-γ coding, described in [39], and it allows to encode the numbers from 1 to 8 with 4 bits. Therefore if we have to encode 2 consecutive numbers in that range we use exactly one Byte rather than the two that would be needed by VBFast, without any space overhead.
Since in sorted files it is very common to encode rears and suffixes lower than 8, often we can approximately halve the space that would be needed by VBFast to encode the integers.
The padding is needed because otherwise we would incur in too much overhead during de decoding of any strings, since we would need many shift and logical operations during the scan of a block.
Unfortunately, the Gamma padding encoder requires more time in the decoding than VBFast, and also more time than Elias-γ, because we have to insert the padding in the correct places. Moreover if the numbers to be encoded are large on average, we may use more bytes than VBFast.
Now we will spend some more words about our Gamma Padding implementation because it is not simply the byte aligned variant of Elias-γ coding. Suppose we have to encode an integer x via Gamma Padding, the implementation looks at what *cost class* x belongs to, and encodes it accordingly.
The cost classes are stored as an array, call it $CS[]$. The algorithm checks what is the i such that $CS[i] < x \leq CS[i+1]$ and then it accesses to the i-th position of another array, which is the *binary width* array $BW[]$.
At the end, i is the number of preamble bits, whilst $BW[i]$ contains the number of bits that must be used to represent $x - CS[i]$.

The array of cost classes can be tuned in order to optimise the numbers we are going to encode. The array which we used in our implementation tries to be the best for encoding couples of small numbers. It is reported in the following listing, together with the array of binary width.

```
1  std::array<unsigned int, 16> soda09_len::cost_classes =
2  {
3          0U, 8U, 16U, 24U,
4          32U, 48U, 64U, 80U,
5          112U, 176U, 304U, 560U,
6          1072U ,2096U, 4144U, 1052720U
7  };
8
9  std::array<unsigned int, 15> soda09_len::binary_width =
10 {
11         3U, 3U, 3U, 3U,
12         4U, 4U, 4U, 5U,
13         6U, 7U, 8U, 9U,
14         10U, 11U, 20U
15 };
```

In the following, we show some examples.

$$\text{Gamma Padding}(3) = \texttt{1010 0000}\ ,\ \text{Gamma Padding}(4) = \texttt{1011 0000}$$
$$\text{Gamma Padding(3,4)} = \texttt{1010 1011}$$

The last example shows that we can fit two integers in the same byte using this kind of encoder (in that case without padding bytes). It would have been impossible with any byte-aligned encoder.

Chapter 4

Retrieval Experiments

In this chapter we will analyse in depth the implementation of the various methods we devised in order to *compress* and *index* dictionaries of strings. We will propose various theoretical considerations on the best algorithm to be used in order to achieve a good spacetime tradeoff concerning the Retrieval operation. All these considerations will be supported by the results obtained by the various benchmarks implemented to know the time and space needed to perform the sub-operations which constitute the Retrieval. Finally, we will compare and comment the time and space results obtained by our implementation with the ones got with competitor data structures.
To give an idea of how this chapter is structured, we anticipate that in Section 4.1 we discuss about some of the details of the implementation of the compression algorithms described in the previous chapter, in Section 4.2 we introduce the actual dictionaries on which our experiments have been performed, in Section 4.3 we expose a brief description of the algorithms with which we compared our solutions and in Section 4.4 we illustrate the results of the experiments and the comparison with other efficient data structures.
The following table shows all the most important features of the machine on which the tests have been performed.

Processor	2GHz Intel core i7 quad core
L2 cache (per core)	256 KB
L3 cache	6 MB
Main memory	8 GB, 1.6 GHz, DDR3

Since it will be widely used in this chapter and in the following ones, we give the following definition.

Definition. *We define the compression ratio (CR) between the binary representation of a dictionary $B(\mathcal{D})$ and its encoding via any compression algorithm $\mathfrak{C}(\mathcal{D})$ as*

$$\mathrm{CR} = \frac{|\mathfrak{C}(\mathcal{D})|}{|B(\mathcal{D})|}$$

We recall that the cardinality of the binary representation of any set is given by the number of bits composing it.

4.1 Implementation

The implementation has been written in C++, exploiting some features of the C++11 standard. The code was developed trying to separate the indexing method from the integers encodings algorithm.
For this purpose, the C++ `struct` which implements the Retrieval(u, l) function (call it `Retriever`) has a template argument: the indexer `IND`, which in turn has another template argument : the integer encoder `ENC`.
When the constructor of the `Retriever` class is called, one instance of the class `IND`: `ind` is created and put as fields of the instance of the `Retriever` class. In order to get the position and the index of the rightmost copied string preceding the u-th in the compressed dictionary, the Retrieval(u,l) implementation calls the function `ind.Get`(u), which executes the right operation according to the representation of the indexes and of the positions. We can say that `Retriever` *delegates* to `ind` the management of the indexing and `IND` delegates to `ENC` the management of the integers encoding.
For example, if the indexes are stored in an array, `Retriever` calls a binary search on it and if the positions are stored as an Elias-Fano bitvector, it calls a Select.
All the functions calling performed by the Retrieval code incur in no overhead caused by the subroutines invocations because these subroutines have been qualified with the `inline` keyword. Such a keyword allows the compiler to substitute the invocation of subroutine `f` inside a routine `F` with the actual assembly code of `f`.
For the same reasons we never use any recursive function to implement Retrieval.

In order to optimise the scan of the blocks, the copy of the characters from the compressed input to the output buffer are performed 8 Bytes at a time instead of 1 Byte at a time. This would be good because it would reduce the number of iterations in the `for` loop that copies the characters and therefore the number of checks needed to see if the condition of the loop is still satisfied. To do that, we set a pointer `st` to a 64 bit unsigned integer pointing to the first Byte to be copied and another one, `sc`, pointing to the first character of the output buffer to be overwritten. Knowing that `suffix` is the amount of Bytes to be copied, the following listing shows the lines of code needed to perform such a copy.

```
1 for (addi = 0; addi <(suffix+7)>>3; addi++){
2         *sc = *st;
3         sc = sc+1;
4         st = st+1;
5 }
```

We point out that in C and C++ the division between two integers x, y gives $\left\lfloor \frac{x}{y} \right\rfloor$. So, given that $\left\lceil \frac{x}{y} \right\rceil = \left\lfloor \frac{x+y-1}{y} \right\rfloor$, x>>$y = \left\lfloor \frac{x}{2^y} \right\rfloor$ and that the * operator is the *dereferencing*[1] operator of the C language, we have that the lines of code shown above allow to copy the correct amount of Bytes. Note that we do not need to copy *exactly* `suffix` Bytes, we may copy more and report only the ones we need.

4.2 Datasets

In this section we will spend some words to introduce the data sets on which the test have been performed. We will show why we choose to use right them instead of others and we will illustrate their most important features, that is, the features that influenced the choice on what compression schema to use in order to compress them.
For example, if we know that the average length of the rears[2] in a dictionary is smaller than the average length of the longest common prefixes, we may choose to encode the dictionary with rear coding instead that with front coding, or vice versa.
Moreover, if we know that the average lengths of the suffixes is low, e.g., three or four characters, we may decide to use an integer encoder which is not very performant in terms of time but is efficient in terms of space, because the amount of saving in space would be not negligible. Obviously, also the number of bits occupied by the uncompressed dictionary is an important feature, because it allows to evaluate the compression efficiency of our solutions. In the following paragraphs we will report such features for each dictionary tested in this chapter.

Google Books 1gram *Google Books 1gram* (shortly 1gram) is a file belonging to the Google Ngram corpus. In general, in the fields of computational linguistic and probability, a n-gram is a *contiguous sequence* of n items from a given sequence. In particular, 1gram is the file containing all the sequences of one words appearing at least forty times in any english book printed from 1800 to 2012 and belonging to the Google Books corpus. The Google Books corpus is simply the dataset of all the books scanned and digitalised by Google.[3] Google 1gram has several entries according to the contextual meaning of each word, for example, in 1gram we find four separated entries for the word "circumvallate", illustrated by the following list.

[1] If `P` is a pointer such that `P` $= \&A[i]$ for some sequence A, then `*P` $= A[i]$.
[2] See section 3.1.1, paragraph dedicated to rear coding.
[3] See `http://books.google.com` for further information

1. `circumvallate`. The word "circumvallate" has been found at least 40 times in the corpus.
2. `circumvallate_ADJ`. The word "circumvallate", used as an adjective, has been found at least 40 times in the corpus.
3. `circumvallate_NOUN`. The word "circumvallate", used as a noun, has been found at least 40 times in the corpus.
4. `circumvallate_VERB`. The word "circumvallate", used as a verb, has been found at least 40 times in the corpus.

The sources are downloadable from `http://storage.googleapis.com/books/ngrams/books/datasetsv2.html`. The most important features of the 1gram dictionary are described in Table 4.1.

Google 1gram	
Size (MB)	275.086
Number of strings(Millions)	24.359
Average string length	10.29
Average LCP	7.30
Average rears length	3.99
Average suffix length	2.99

Table 4.1: Google 1gram features.

ClueWebUrls The *ClueWebUrls* dataset is a huge collection of URLs taken from about one billion web pages written in ten languages, collected in january and february 2009. URL stands for *Uniform Resource Locator*, that is a sequence of characters uniquely identifying the *address* of a *resource* on the internet. Typically, such a resource is located on a server and it is accessible through a *transfer protocol* by any client machine. A *resource* is any kind of digitalised data, such as videos, texts or images.
Since the ClueWebUrls dataset is enormous, we decided to take end examine only a portion of it to perform quicker tests. Taking a large enough portion of a dataset do not alter too much the performance analysis of our algorithms. Indeed, if we take contiguous, lexicographic ordered URLs, we expect that the dimension of the blocks of uncompressed strings are the same both for the complete file and for the partial one.
The whole ClueWebUrls dataset can be found at `http://lemurproject.org/clueweb09` and it contains almost five *billion* URLs, for a space of more than three hundred GigaBytes. Our sample, which we denote for simplicity URLs, contains instead just twenty millions contiguous strings taken from ClueWebUrls, its main features are summarised in Table 4.2.

ClueWebUrls sample	
Size (MB)	1,387.956
Number of strings(Millions)	20
Average string length	68.40
Average LCP	54.49
Average rears length	14.66
Average suffix length	15.50

Table 4.2: Features of ClueWebUrls sample

We see from Table 4.2 that the strings belonging to such a dictionary are much longer than the ones in Google 1gram, which are taken from the natural language. URLs tend to share a lot of characters[4], almost the 80% of their length, so, we expect that our encoding schemas will produce compressed files which are approximately 20% the size of the URLs dictionary. Hence, we expect that the additional information needed to decode and index any string of the dictionary will have less impact on the compression ratio with respect to 1gram, whilst the time to scan a block will be obviously longer in this file than in 1gram.

4.3 Competitors

In this section we briefly describe the methods used to compare the efficiency of our data structure. For instance, we will show the comparison between our solutions with other widely used compression algorithm, such as gzip and bzip. We expect that these algorithms will occupy less bits than ours, also because they do not support the indexing of the strings. Indeed, in order to access a single string when the dictionary is encoded via gzip or bzip we have to decode the entire file and then scan it up to the point in which the string to be found is reconstructed.
The comparison with gzip and bzip is significant because these two algorithms are known to be among the most efficient compression algorithms for set of strings, hence they can be considered "empirical" lower bounds for the space of any compressed dictionary.
In particular, the gzip compression algorithm is based on a variant of LZ77 [40], commonly called *deflate*. More information may be found at `http://www.gzip.org/algorithm.txt`.
The bzip algorithm is instead based on the Burrows-Wheeler transform [6], further information about bzip implementation can be found at `http://www.bzip.org`.
We also propose a comparison with the solutions proposed by Ottaviano and Grossi in [34], implemented in [32]. A survey of the techniques used in this solution can be found in section 2.14. From now on, we will refer to the schemas illustrated in that article using

[4] All the strings share the transfer protocol name, which is `http`, indeed all of them share the same prefix `http://`.

the following names.

1. centroid: path decomposed trie through *centroid* path decomposition
2. lex: path decomposed trie through *lexicographic* path decomposition.

The comparison with these data structure is significant because they have been proved to be among the best ones in storing and indexing compressed dictionaries. In particular, the most relevant comparison is the one between our data structure and lex because for both of them Access(i) returns the i-th string according to the lexicographic order. Instead, Access(i) performed on centroid returns just any of the strings of the dictionary, without following any order. If we encode a dictionary $\mathcal{D}$ such that $|\mathcal{D}| = n$ with centroid path decomposition we are only sure that for $i, j = 0, ..., n-1$ $(j \neq i) \Rightarrow (\mathsf{Access}(i) \neq \mathsf{Access}(j))$.

4.4 Experiments

In this section we will report and comment the performance measurement obtained with our compressing schemas. Before any experiment being performed, we will discuss all the possible indexing variants and we will report some benchmarks to evaluate *quantitatively* such variants. We remind that the time and space efficiency of our solution can be influenced by the algorithm used to encode the integers, or by the way we store the positions and the indexes of the strings. Hence, for the rest of the chapter we will give the following names to the data structures storing such values.

1. E: set of the positions
2. V: set of indexes

We recall that E can be made by the positions of copied strings, and in this case it can be represented as an Elias-Fano bitvector or as an array. In the case E is made by the positions of all the strings, it can be represented (space efficiently) only via Elias-Fano. V always consists of the indexes of the copied strings and it can be represented as a bitvector supporting Rank and Select operations, or as a plain array. For more details about the implementations of E and V as succinct data structures, see [21].
We give the following names to the various (binary) representations of E and V.

1. E_{all}: E is represented as an Elias-Fano bitvector indexing the starting positions of the encodings of *all* the strings.
2. E_c: E is represented as an Elias-Fano bitvector indexing only the positions of the copied strings.
3. E[]: E is represented as an array such that E[i] contains the position of the i-th copied string (according to the lexicographic order).

4. V_{RS}: V is represented as a binary array capable of supporting Rank and Select operation.

5. V[]: V is represented as an array such that V[i] contains the *index* of the i-th copied string.

Another important choice is the choice of the integer encoder to encode the numbers located in the compressed representation of the dictionary. Indeed this choice may impact significantly the performance of the implementation. The following list shows the integer encoders which have been implemented and taken into account in our solution. Unless it is specified differently, the description of the encoders can be found in Section 3.2.2.

1. VBFast(32): version of VBFast which has *two* preamble bits to determine the number of bytes to be scanned in order to decode the integer.

2. VBFast(16): version of VBFast which has only *one* preamble bit.

3. Variable Byte: standard Variable Byte encoder[5].

4. GP: byte-aligned variant of Elias-γ coding.

4.4.1 Google 1gram

In this section we will report many considerations on the application of our compressing scheme to the Google 1gram dictionary. Then we will report the results of the experiments and some comments about them.

Choosing the integer encoder We may immediately notice from Table 4.1 that on average the common prefixes are almost two times longer than the rears. This is quite reasonable in any big sorted file since two consecutive strings tend to have shared prefixes longer than their characteristic suffixes.
Therefore we may encode the *rears* instead of the longest common prefixes in our compression algorithm. Table 4.3 reports the space occupancy and the average time to access decode an integer with the various encoding algorithms we have at our disposal. In order to take such measurements, we considered the case in which the number of integers in the compressed dictionary $\mathcal{D}$ is maximal, hence the case in which the compression is done by LPFC($|B(\mathcal{D})|$). Table 4.3 shows that despite a considerable loss of time, with Gamma Padding (GP) we can save a remarkable amount of space. This is confirmed by the following table, showing the most relevant entries of the distribution of the integers in the compressed file.

[5]See Section 2.10.2, paragraph named "Variable-Bytes coding".

Table 4.3: Space and time performance of the integers encoders

Number of integers: 48,718,374		
Name	Space(Bytes)	Time(ns)
Variable Byte	48,718,376	3
VBFast(32)	48,718,384	2.4
VBFast(16)	48,718,376	2.4
GP	26,436,995	10

Distribution of the rears		Distribution of the suffixes	
Value	Frequency	Value	Frequency
1	35.8%	1	33.2%
2	5.8%	2	11.9%
3	3.9%	3	6.5%
4	15.9%	4	24.8%
5	9.3%	5	21.6%
6	8.6%	6	0.8%
7	7.8%	7	0.5%
8	4.4%	8	0.3%

According to the values reported in the table above, since 99.9% of the suffixes and approximately 90% of the rears can be encoded with just four bits, we have that more or less 90% of the compressed strings can have their suffix and rear that fit in one byte, with no padding overhead. While for the same cases, any other considered encoder would use two Bytes. The actual save is about 22.3 MB, a quantity which is *not negligible* considering the file to be compressed, in fact it represents 8.1% of the size of the uncompressed dictionary. As a conclusion Gamma Padding *can be chosen* as integer encoder to compress 1gram.

Choosing the representation of E and V As we did when we had to choose the integers encoder, we have to consider both the space and time performance of the data structures representing E and V. Table 4.4 shows the time to perform the Rank and Select operations when E and V are stored succinctly. Time values are expressed in nanoseconds (ns)[6]. In the second column of this table we show the time to perform the operations when the parameter i of Rank(i) or Select(i) is being increased by one at each iteration, while the results presented in the third column are obtained spanning the parameter i randomly. We may see that the most expensive operation is Select(V_{RS}). In fact this operation is

[6]We recall that one nanosecond is equal to 10^{-9} seconds, while one microsecond (μs) is equal to 10^{-6} seconds

Table 4.4: Time to perform operations on the succinct data structures

Operation	Consecutive queries time (ns)	Random queries time (ns)
Select(E_c)	24	37
Select(E_{all})	24	113
Select(V_{RS})	84	131
Rank(V_{RS})	6	13

not implemented via a constant number of memory accesses like the other ones, but via a *hinted binary search*, see [21].
It is also interesting to see that if the argument i of Select(E_{all},i) is taken randomly we have a much higher execution time than the one obtained with Select(E_c,i), even though the implementations and the data structures are identical. This is simply due to the fact that E_{all} is larger than E_c (whatever the compression algorithm), so it avoids to exploit locality as much as E_c in random memory accesses. Indeed we may say that an higher memory level is needed to store E_{all}.

For what concerns space occupancy, V_{RS} has *always the same size*, because its size depends only on the number of strings in the dictionary, that never changes. V_{RS} needs approximately one bit for each string, hence approximately 3 MB plus the bits needed to support Rank and Select. The empirical value of the size of V_{RS} is given in the following.

$$|V_{RS}| \approx 3.8 \text{ MB}$$

If E is represented as E_c, its size is actually negligible; when E is represented as E_{all}, we can estimate a lower bound relative to its space occupancy exploiting formula 3.8. We know the average suffix length for a single string $E[u]$ and the total number of strings n from table 4.1.

$$nE[u] \approx 72.83 * 10^6 = 72.83 \text{ MB}$$

If we sum to $nE[u]$ the minimum amount of bytes needed to encode the integers, that we may find in the GP entry of Table 4.3, calling M the size of the compressed dictionary, we may get an estimation for the minimum value of M as follows.

$$M \approx 72.83 + 26.44 = 99.27 \text{ MB}$$

Now we have n and M. We can evaluate the *lower bound* for the size of E_{all}.

$$|E_{all}| > 2 * 24.36 + 24.36(\log 99.27 - \log 24.36) = 48.72 + 24.36(6.6 - 4.6) = 48.72 + 48.72 = 97.44 \text{ Mb} = 12.18 \text{ MB}$$

If we set M to its maximum value, which is the size of the uncompressed file, we get an *upper bound* for the size of E_{all}.

$$|\mathrm{E}_{all}| < 48.72 + 24.36(\log \tfrac{275}{24.36}) = 48.72 + 24.36(3.5) = 48.72 + 85.26 = 133.98 \text{ Mb} = 16.75 \text{ MB}$$

Finally, we may conclude that the size of E_{all} fits in the following range.

$$12.18MB < |\mathrm{E}_{all}| < 16.75 \text{ MB} \tag{4.1}$$

We may notice from Table 4.3 and 4.4 that it is not convenient to represent E as E_{all} because if we store in E_c only the positions of the copied strings inserting directly beside each string the length of its suffix encoded with Gamma Padding we get some advantages both in time and space. Indeed the time to know the starting position of the next string in the latter case is equivalent to two consecutive decoding with Gamma Padding which is equal to $20ns$, while with the former technique we need a Select and a decoding with VBFast, which take approximately 26 ns.
Moreover, Gamma Padding is able to fit the largest part of the suffixes lengths in the same bytes in which the rears are stored, allowing to compact most of the information given by VBFast+E_{all} directly in the rear-coded dictionary. The saving in space is consistent because E_c is always much smaller than E_{all}.
Now we must also take into account the possibility of storing V as plain array. The next table shows the space occupied by V[] and the time to perform binary search on it.

Algorithm	Consecutive queries time (ns)	Random queries time (ns)	space (MB)
LPFC(4)	33	204	9.993
LPFC(8)	28	155	5.175
LPFC(16)	25	134	2.745
LPFC(64)	22	111	0.788
LPFC(128)	21	99	0.424

It is possible to note from the table above that in 1gram binary search is not always convenient in terms of time with respect to Rank+Select on V_{RS}. Nonetheless, if we consider only Retrievals performed to retrieve strings which are stored contiguously in memory, we have that binary search beats Rank+Select on V immediately, even with low parameters. In fact, if we search for string $u+1$ after the u-th, it's very likely not to have additional I/Os.
The same table says that binary search is convenient if the parameter is greater than a number in the range (8,16), call it b. Up to that parameter it is better to represent V as succinct bitvector.
If the parameter x of LPFC(x) is such that $x \geq b$, then it is surely convenient that both E and V are memorised as plain arrays. For $x \geq b$, the space occupied by them starts to become negligible and the time for binary search is lower than the time to perform

Rank+Select. In this case it is not convenient to store E as an Elias-Fano bitvector, indeed we would not achieve a consistent gain in space, but there would be some overhead due to the Select operation needed to access the position of the copied strings.

Results After the reasoning performed in the previous paragraph, we may finally report the time and space measurements of our compressing scheme applied to Google 1gram. Such measurements are shown in Table 4.5and in Figure 4.1, both of them report the values obtained with the representations of E and V *we did not discard* in the previous paragraph.

Table 4.5: Performance measurements changing the representation of E and V relative to 1gram. We retrieve $2 * 10^6$ strings at random and all the strings of the file consecutively. Time values are expressed in μ-seconds.
T_{Con} is the average retrieval time when we retrieve strings which are soared contiguously in memory, T_{Ran} is the average retrieval time when we retrieve strings at random. The integer encoder is always GP.

(a) Comparison with E=E[], V=V[]

Name	T_{Con} (μs)	T_{Ran} (μs)	CR %
LPFC(4)	0.226	0.698	48.01
LPFC(8)	0.336	0.771	42.03
LPFC(16)	0.545	0.995	39.14
LPFC(64)	1.650	2.194	36.90

(b) Comparison with E=E[], V=V_{RS}

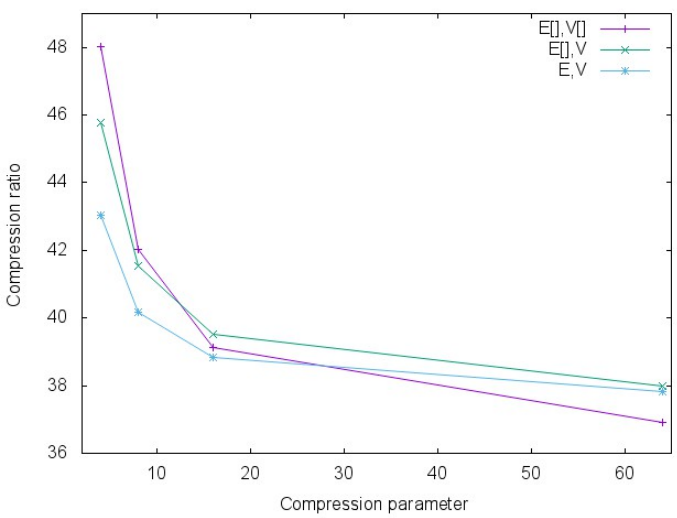
Name	T_{Con} (μs)	T_{Ran} (μs)	CR %
LPFC(4)	0.285	0.594	45.76
LPFC(8)	0.396	0.735	41.53
LPFC(16)	0.606	0.975	39.52
LPFC(64)	1.715	2.262	38.00

(c) Comparison with E=E_c, V=V_{RS}

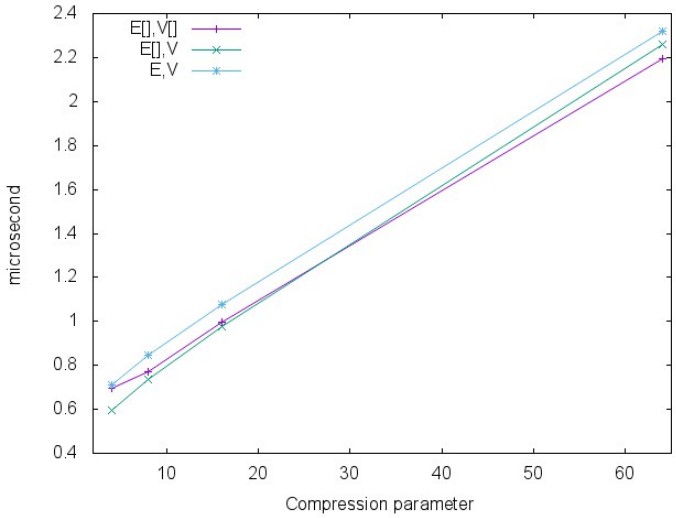
Name	T_{Con} (μs)	T_{Ran} (μs)	CR %
LPFC(4)	0.311	0.713	43.03
LPFC(8)	0.426	0.846	40.17
LPFC(16)	0.634	1.076	38.83
LPFC(64)	1.747	2.320	37.82

We can notice from Table 4.5 and from Figure 4.1 the small overhead in time when E=E_c.We can also note that the version in which E=E[] and V=V[] is *asymptotically* the best both in space (the size of E[] ad V[] tend to become 0) *and time*. It is also the best choice if we want for example to retrieve consecutive pieces of the file because binary search has the best exploitation of locality. If we retrieve strings at random, the other versions seem to be a little better when the parameter x of LPFC(x) is low. In that case the best tradeoff is E=E[] and V=V_{RS} since we do not loose to much space representing E as an array and we may exploit the better performance of V_{RS} with respect to V[].

Figure 4.1: Occupied space and average retrieval time.



(a) Occupied space



(b) Average retrieval time

Finally, Table 4.6 shows the results obtained with other solutions.

Table 4.6: Time and space performance obtained with other solutions applied to 1gram. Both gzip and bzip are applied with the maximum compression level.

Algorithm	Consecutive queries time (μs)	Random queries time (μs)	CR%
gzip	0.075	//	23%
bzip	0.289	//	26.34%
centroid	0.768	1.677	45.14%
lex	0.924	1.973	43.64%

We may argue from Table 4.6 that the improvement in time with respect centroid and lex is due to the fact that LPFC *better exploit memory locality* with respect to path decomposed tries. Indeed, with LPFC we are sure to have an optimal amount of *contiguous* memory accesses up to the string we are looking for, whilst path decomposed tries do not ensure that.
In the same table we see also the results obtained with gzip and bzip. Indeed these algorithms greatly outperform ours but they do not support random string accesses. To retrieve any string s_i of the dictionary, we need to decompress the whole file up to the point in which s_i is reconstructed, which is definitely not convenient.

4.4.2 URLs

In this section we deeply analyse the application of our data structure to the ClueWebUrls sample dictionary described in Section 4.2, following the same outline of the previous section.

Choosing the integers encoder As we did for the Google 1gram dictionary, having a look to Table 4.2 we may argue that it is much more convenient to store the "rears" instead of the prefixes beside the compressed strings. Table 4.7 shows the time and space performance of the various encoders when the number of integers is maximal, that is with FC and with $E \neq \mathrm{E}_{all}$.

Table 4.7: Integer encoders space and time cost for URLS

Number of integers: 39,999,998		
Name	Space(Bytes)	Time(ns)
Variable Byte	46,260,400	3
VBFast(32)	47,281,796	2.5
VBFast(16)	46,260,400	2.4
Gamma Padding	40,895,687	10

In this case, as it is suggested by table 4.7, it is not convenient to use Gamma Padding because the saving in space with respect to VBFast is really negligible: less than 6MB (approximately 0.5% of the size of URLs). Even if the gain in space were not so low, Gamma Padding should not be used because though we could stuff couples of rears and suffixes in one single byte, we would save only 20MB, which are more or less 1.5% of the file, at the cost of a remarkable accretion of the decoding time and therefore also of the time to scan a block.

Choosing the representation of E and V Thanks to the fact that the average string length is noticeably higher than the one in 1gram, we can immediately think that it is convenient to represent both E and V as arrays already when the parameter is low. The copied strings would be very few even in that case, so we will get advantages both in time and space very soon. The interesting fact about storing E and V as arrays is that the lower is their space, the lower is the time to binary search on them. Which is the contrary of what happens when we retrieve a string from the compressed file. Indeed, the block scanning phase becomes increasingly dominant as much as the parameter of the compression algorithm grows.
Regarding the time performance of E and V stored as succinct data structures, we point out that Table 4.4 is valid also for this file. As a matter of fact, the sizes of such data

structures are similar and we know that the operations require the same number of steps. The size of V when it is represented as a bitvector able to support Rank and Select is akin to the one of 1gram, its actual value is given in the following.

$$|\mathrm{V}_{RS}| \approx 3.59 \text{ MB}$$

Using formula 3.8, which we exploited to derive the bounds in 4.1, we may notice that when $E = \mathrm{E}_{all}$ the following condition holds.

$$15 \text{ MB} \leq |E_{all}| \leq 22.5 \text{ MB}$$

Hence, also in this case it is better to store the lengths of the suffixes directly beside the strings, but this time encoded with VBFast(16). The benefits in time are clear: the time to decode an integer with VBFast is much lower than the time to perform Select(E_{all}, i). For what concerns space, storing $E = \mathrm{E}_{all}$ we achieve a gain in space which is equal to few MegaBytes, hence negligible. In order to prove the efficiency of storing $E = \mathrm{E}[]$ and $V = \mathrm{V}[]$, we may notice that their size is quite neglectable with respect to the size of the uncompressed dictionary even when the compression parameter is low. For instance even if we take 2 as compression parameter, and we apply LPFC(2), because of the formulas 3.5, 3.6 we get that the number of fully copied strings in the compressed dictionary is the following.

$$C_s \approx \frac{20*10^6}{5.36} \approx 3.73 * 10^6$$

Thus, representing each entry of V[] as 32 bits integers, we have that $|\mathrm{V}[]| \approx 14.92\text{MB}$ $\approx 1.1\%$ of the file.
The following table shows the time to binary search on V[] and its size varying the LPFC parameter.

Algorithm	Consecutive queries time (ns)	Random queries time (ns)	space (MB)
LPFC(4)	29	164	6.078
LPFC(8)	26	134	2.960
LPFC(16)	23	120	1.511
LPFC(64)	20	99	0.420

The table above points out that binary search on V[] beats Rank(V_{RS},Select(V_{RS},i)) in time when it beats it also in space. From the very same table we see that this fact happens already when the parameter is 8, which is low and in general implies a good space-time tradeoff. We recall the the size of E[] is the same of the one of V[].

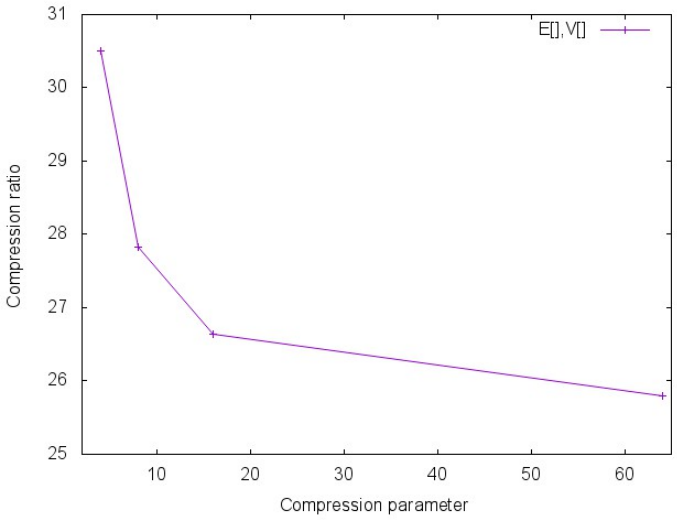
Results Summing up the considerations done in the previous paragraphs, we decided to use VBFast(16) as integers encoder and to represent both the positions and the indexes of the uncompressed strings as arrays of 32 bits cells. Table 4.8 and Figure 4.2 show the

space and time performance obtained performing Retrievals on the URLs file compressed with the settings previously mentioned.

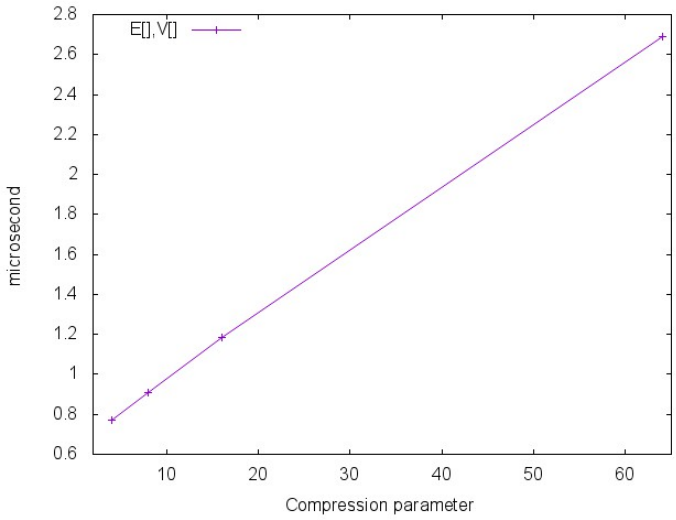
Table 4.8: Time to perform Retrieval(u, l) on compressed URLs. We retrieve 2 million strings at random and all the strings of the file consecutively. Time values are expressed in μ-seconds

Algorithm	Consecutive queries time (μs)	Random queries time (μs)	CR(%)
LPFC(4)	0.240	0.770	30.50
LPFC(8)	0.371	0.908	27.83
LPFC(16)	0.609	1.183	26.64
LPFC(64)	1.930	2.692	25.79

Figure 4.2: Occupied space and average retrieval time.



(a) Occupied space



(b) Average retrieval time

We may notice from Table 4.8 that the compression ratio is quite good with respect to Google 1gram thanks to the fact that the strings share a larger part of their total length. Moreover, the binary representation of the shared characters is much wider than the encodings of the integers.
Finally, Table 4.9 shows the performance obtained with other solutions.

Table 4.9: Time and space performance obtained with other solutions applied to URLs.

Algorithm	Consecutive queries time (μs)	Random queries time (μs)	C.R.%
gzip	0.141	//	9.4%
bzip	1.163	//	8.1%
centroid	1.356	2.707	23.5
lex	2.036	3.756	23.44

Once again, we notice from Table 4.9 that gzip and bzip outperform all the presented solution for what concern space and the time to extract the whole compressed file, though they do not support Access(i) without scanning the entire file up to the point s_i is decompressed.
Comparing our solution with lex or centroid, we see that they beat ours very slightly in space, whilst they are widely beaten by ours in Access time.

Chapter 5

Prefix Search

In this chapter we analyse some strategies to solve the *prefix search problem* (described in Section 2.11) in an efficient way. We briefly recall that we want to find the range of strings in a dictionary $\mathcal{D}$ which are prefixed by an input pattern P. The basic data structures to support such a facility are the ones described in Chapter 3, which are in summary a dictionary $\mathcal{D}$ compressed with locality preserving front coding plus some indexing scheme to access any string of $\mathcal{D}$.
We have divided the current chapter in two sections. In Section 5.1 we analyse the algorithms needed to support prefix search without adding any additional information to the solutions devised in Chapter 3, in Section 5.2 instead, we describe an additional data structure which allows our solution to support prefix search more efficiently.

5.1 Binary search approach

Here we describe the techniques that can be exploited in order to support prefix search on a dictionary compressed via LPFC without storing additional bits. Section 5.1.1 shows an inefficient, yet intuitive way to support such a facility, whilst Section 5.1.2 illustrates a more efficient way to perform prefix search on the compressed dictionary.

5.1.1 Trivial binary search

The data structures described in Chapter 3 indeed offer the opportunity to devise an algorithm to prefix search on them. The array of string pointers, or its succinct version allows in fact to access any of the sorted strings, which can be compared with our lookup pattern P in order to find its possible lexicographic position in the dictionary $\mathcal{D}$. If $|\mathcal{D}| = n$, the order of the strings makes sure that we can "binary search" P in the dictionary, hence the maximum number of strings taken into account in order to determine the lexicographic position of P is $\lceil \log n \rceil = O(\log n)$.
In order to find a solution to the prefix search problem, we augment the alphabet Σ from

which the strings of the dictionary $\mathcal{D}$ are drown with a *special symbol*: "#" such that # is *lexicographically greater* than all the others symbols in Σ. Hence, a new alphabet Σ^+ is given by $\Sigma^+ = \Sigma \cup \{'\#'\}$.
So, supposing that we have at our disposal an algorithm that finds the lexicographic position of any pattern in the compressed dictionary, we can find a solution to the *full prefix search* problem finding the position of P and then the position of $P\#$ in $\mathcal{D}$. Now, we may distinguish two cases.

- *Case 1.* There exists a range $[i, j]$ with $i \leq j$ in $\mathcal{D}$ such that for $k = 0, ..., (j - 1)$; P is a prefix of $\mathcal{D}_{i+k}$. In this case the lexicographic position of P would be i, while the position of $P\#$ would be $j + 1$.

- *Case 2.* There are no strings in $\mathcal{D}$ prefixed by P. In this case the lexicographic positions of P and $P\#$ would coincide.

Hence if we can find the lexicographic position of any patter P in $\mathcal{D}$ we were able to answer full prefix search because we were able to see when such pattern is not a prefix of any string of the dictionary.
The only additional primitive that we need to perform a binary search on the dictionary as it were a binary search on integers[1] is a function that allows to *compare* two strings. We call such a function StrCompare(string x, string y)[2], it returns 0 if x and y are equal, a negative integer if x is lexicographically smaller than y and a positive integer if x is greater than y. We may point out we never need to compare more than $|P| + 1$ symbols, hence at each step of binary search we can extract at most the first $|P| + 1$ of each string to be compared. Now we have all we need to answer full prefix search with a binary search like procedure, which is described by Algorithm 3.

[1] Or in general performed on a sequence of sorted fixed lengths elements.
[2] Whose implementation can be found in the C++ standard library

Algorithm 3 Binary Prefix Search

```
1: procedure BINARY PREFIX SEARCH(P,D)
2:     R[0] = SEARCH(P,D)
3:     R[1] = SEARCH(P#,D)
4:     return R
5: end procedure
6: procedure SEARCH(P,D)
7:     L = 0 , R = |D| − 1;
8:     while L ≤ R do
9:         M = ⌊(R + L)/2⌋
10:        if StrCompare(P, Retrieval_D(|P|,M) > 0) then
11:            L = M + 1
12:        else
13:            R = M
14:        end if
15:    end while
16:    return L
17: end procedure
```

We may note that Algorithm 3 requires $O(|P| \log n)$ operations and, if we compress the dictionary $\mathcal{D}$ with locality preserving front coding, it needs $O(((1 + \frac{1}{\epsilon})/B)|P| \log n)$ I/Os, where ϵ is a constant tuned by the strings compressor and B is the page size.
Obviously, if the strings are stored not compressed, the number of I/Os is reduced to $O(\frac{|P|}{B} \log n)$.

5.1.2 Binary searching only on the uncompressed strings

A way to much further improve prefix search is to apply binary search only to the strings which are directly copied in the compressed dictionary.
Thus, we exploit a two level schema in which as a first step, we find the lexicographic position of pattern P among the copied strings, then we scan the blocks of the compressed dictionary in order to find the actual position of P in $\mathcal{D}$. Regarding the latter point, we must pay attention to the following important observation.

Observation 5. *Given that the lexicographic position of P in the set of the copied strings $\mathcal{C}$ is i, then, if $\mathcal{C}_i$ is the j-th string in the dictionary $\mathcal{D}$ and $\mathcal{C}_{i-1}$ is the k-th string of $\mathcal{D}$; we know that the actual lexicographic position of our searched pattern P is in the range $(k, j]$.*

Thus, scanning the block of compressed strings in that range, that are at most all the strings in a block produced by LPFC, we find the lexicographic position of P among the strings of the dictionary. Obviously we may apply the same technique looking for

the pattern $P\#$ (assuming that the strings of the dictionary are drawn from the alphabet $\Sigma^+ = \Sigma \cup \{\#\}$ described in the previous subsection). At the end, if l is the lexicographic position of P in $\mathcal{D}$ and h is the lexicographic position of $P\#$, we know that the range identifying the strings which are prefixed by P is $[l, h)$.
Figure 5.1 shows the source code for the two-level prefix search, written in C++. Note that:

1. `Num_Copied`: contains the total number of copied string in the dictionary $\mathcal{D}$.
2. `uint64_t Get_Index_of_Copied(int L)`: is the function such that, given the index of a copied string with respect the ordered set of copied string, returns the index of the very same string but with respect to all the strings of the dictionary.
3. `uint64_t ScanBlock(string P, int i,int j)`: is the function scanning the block between the i-th and j-th string and returning the index of the first string which is lexicographically *greater or equal* than `P`.

Figure 5.1: C++ function used to return the range of strings prefixed by P, exploiting binary search.

```
1  void Prefix_search( const string &P,uint64_t*Range){
2  uint64_t LSI=0;
3  Range[0] = ScanBlock(P,Prefix_search_LRI(P,1,NumCopied,&LSI)-1,LSI);
4  string NP = P+(char)0xFFFF;
5  Range[1] = ScanBlock(NP,Prefix_search_LRI(NP,1,NumCopied,&LSI)-1,LSI);
6  }
7
8  uint64_t Prefix_search_LRI(const string &P, uint64_t L, uint64_t R,uint64_t*
       ↪ v){
9          uint64_t mid_Point = 0;
10         int P_size = P.size();
11         while (L!=R){
12                 mid_Point = (L+R)/2;
13                 switch(P.compare(Retrieval_Copied(mid_Point,v,P_size))){
14                         case 0 :
15                         return mid_Point;
16                         case 1 ... INT_MAX :
17                         L = mid_Point+1;
18                         break;
19                         case INT_MIN ... -1:
20                         R = mid_Point;
21                         break;
22                 }
23         }
24         *v = coder.Get_Index_of_Copied(L);
25         return L;
26 }
```

We have chosen to use the `switch` statement instead that a chain of `if,then,else` because the `switch` is much more performant in checking constant expressions. Now we can point out the several benefits of the two-level binary search with respect to the single-level binary search.

1. $|\mathcal{D}| > |\mathcal{C}|$. The number of copied strings is lower than the number of the dictionary strings.
2. During the first level we have that $\mathsf{Retrieval}(u, l)$ requires *exactly* $\frac{|\mathcal{D}_u|}{B}$ I/Os, where B is the page size.

Regarding the first point: the number of copied string can be much lower than the number of strings in the dictionary, so, despite the logarithmic reduction of the I/Os, the utter number of I/Os with the two level scheme can be significantly lower. Very roughly, doubling the compression parameter x of $\mathsf{LPFC}(x)$, we decrease the number of I/Os by one. The second point gives the *fundamental* contribution with respect to the performance of Algorithm 3. Indeed, with the two-level binary search the time to retrieve the strings taken in examination during the binary search is *perfectly* optimal *regardless* of the compression parameter. As a consequence of point 1 and 2, the time to perform binary search on the copied strings decreases with the increase of the compression parameter.

On the other hand, the scan of the block required by the second phase of the algorithm is equivalent to just one additional random $\mathsf{Retrieval}(u,l)$. Its I/O complexity is therefore equal to $O(|\mathcal{D}_u|/B)$ if the file is compressed with $\mathsf{LPFC}(x)$. Actually, If the average length of a string in $\mathcal{D}$ is L, we have that we need on average $\frac{Lx}{2B}$ I/Os to retrieve the lexicographic position of the pattern in the block.
Despite the fact that the number of I/Os is optimal, a source of inefficiency can be the number of comparisons needed to find the first string which is lexicographically greater or equal than P. In order to reduce the number of such comparisons, we may point out the following observation.

Observation 6. *In order to improve the search inside a block it is possible to compare the pattern P we are searching with the currently scanned string s only if $|s| > |P|$.*

Proof. Whatever the pattern P we are searching for, we are assured from the binary search procedure to start the scan of the block from a copied string which is lexicographically smaller or equal than P. □

5.2 Ternary search trie

In this section we show that we can answer full prefix search very efficiently with the help of an additional ternary search tree[3] built upon our dictionary. Our dictionary is

[3]See Section 2.13 for further information.

still compressed via locality preserving front coding and indexed using the data structures illustrated in Section 3.2.1, but we add an additional ternary search tree (TST) built over the the set of the uncompressed strings. In particular we may store the TST as a *PATRICIA ternary search tree*, in which every node contains exactly one character and the length of the longest common prefix shared by the strings associated to all the descending leaves.
In section 5.2.1 we illustrate how to build up such a ternary search tree, while in Section 5.2.2 we point out how to support prefix search on the compressed dictionary with this kind of data structure.
We have chosen to represent the ternary search tree in a PATRICIA fashion because PATRICIA tries are very efficient in reality. In fact they avoid unary paths and random I/Os to be performed on the memory area in which the dictionary is stored, without incurring in serious slowdowns in the search for a particular pattern P.
Actually, performance in ternary search trees (or tries) is assured by their *balancing*. A not balanced tree may make its traversal (and hence pattern search) very expensive. In pathological cases the average time complexity of the traversal may become *linear* with respect to the number of nodes of the tree. For instance if the strings are inserted in *lexicographic order*, there will be no nodes with a left child. The traversal of the tree will actually become the traversal of a series of linked lists.
Fortunately, it has been shown in [4] that inserting the strings in a proper order allows to create an almost perfectly balanced tree. For instance we can insert the median string of the dictionary, split the dictionary in two sub-dictionaries and inserting recursively the median string of these subsets. The trie obtained in such a way is called in that paper *Tournament trie*.
Moreover, a balanced trie takes up less space than a not balanced one. In fact, since we have a constant number of leaves equal to the number of the copied strings, regardless of how the tree is balanced, we also have that the more the trie is unbalanced, the greater is the number of `NULL` pointers.
`NULL` pointers do not bring to any node but they have to be stored in memory. Suppose for instance that the strings have been inserted in lexicographic order, then all internal nodes would store a `NULL` pointer as pointer to the left chid. As a result, if the number of leaves is n, we waste $O(n)$ space for NULL pointers.
In order to exploit locality, we have decided to store the tree in a contiguous portion of memory. To do that, we have represented the trie as a C++ `vector`[4] of nodes. The following listing show the C++ `structs` used to represent any node of the tree.

```
1 struct PATRICIA_Node
2 {       pair<uint32_t,uint32_t> Value;
3         uint32_t left,right;
4         uint16_t length;
5         unsigned char data;}
```

[4] In C++ a `vector` is simply a collection of elements having the same memory layout and stored contiguously.

```
6  struct PATRICIA_TST {
7          std::vector<PATRICIA_Node> V;
8          uint Stack[256][2];
9  }
```

First of all we remind that the kind of tree we have decided to use is a PATRICIA tree, the fields of the `struct PATRICIA_Node` are arranged from the bigger to the smaller in order to minimise the padding. In the following list we show the meaning of PATRICIA_Node fields.

- `Value`: stores the indexes of the leftmost and the rightmost string which are prefixed by the string associated to this node.
- `left,right`: they store respectively the left and the right child of the current node. The middle child is not stored because it is assumed that it is the next one in the vector.
- `length`: stores the length of the string associated with this node. We assume that the strings lengths are bounded by 64KB.
- `data`: it is the character such that the prefixes $\{P_i\}$ for $i = 1, ..., k$ associated to the middle child node and all its descendants have `data` $= P_i[$`length`$]$, $for\ i = 1, ..., k$

Regarding the `Stack` array field of `struct PATRICIA_TST`, we will see that it will be used to memorise the nodes visited during pattern search. It would be enough to allocate $\log |\mathcal{C}|$[5] cells if the ternary search trie were perfectly balanced because we would have a tree with $\mathcal{D}$ leaves in which all internal nodes are at least binary. Actually, the trie we are going to build (that is a *tournament trie*) is not assured to be perfectly balanced (in that case an array of 32 cells would be enough because we suppose the copied strings are at most 2^{32}), but it is "almost" perfectly balanced. So, an array one order of magnitude greater than 32 ensures no segmentation faults with a really negligible space overhead.

5.2.1 Insertion

However, the `structs` defined in the previous subsection are not used during the *construction* of the trie. In order to build the trie from scratch, we need a temporary data structure in which the nodes have *pointers* to their children instead of *indexes*.
This is due to the fact that if we have to add an internal node to the trie, we have to modify the positions of *all* the nodes that are located after the inserted one and all the indexes of all the nodes greater or equal the position in which the insertion was done. Basically, we need to scan all the nodes of the tree. If we store pointers instead of indexes, each insertion requires constant time, thus the construction of the trie would require throughout $O(|C|)$

[5] We remind that $\mathcal{C}$ is the set of uncompressed strings.

operation instead of $O(|C|^2)$.
Since the number of copied strings may be higher than 10^6 as order of magnitude, it follows that inserting nodes in the middle of a vector of nodes is not feasible. Just to take an example, suppose that $|C| = 3 \cdot 10^6$ and that our machine is able to perform 10^9 operations per second; the number of such operations to be performed to build up a ternary search tree implemented as a `vector` of nodes would be approximately $(3 \cdot 10^6)^2 = 9 \cdot 10^{12}$, hence the time needed to complete the procedure would be $(9 \cdot 10^{12})/10^9 = 9 \cdot 10^3$ seconds = 2.5 hours, while if we support insertion in constant time, the construction of the tree would require only some milliseconds. Indeed, once we have inserted all the nodes exploiting the pointers, it is easy to serialise the obtained tree in a vector of nodes.
It is supposed that the inserted strings are *prefix-free* because otherwise a node u such that $\mathsf{string}(u) \in \mathcal{C}$ and $\mathsf{string}(u)$ is a prefix of another string belonging to $\mathcal{C}$ would be not a leaf, which could create issues during the search of a pattern. In order to do that, all the strings are extended with a special character which is not in the alphabet from which they are drawn.
For simplicity, given a node v of the ternary search tree, we may denote its fields using the "." operator, for instance the `length` field of v is denoted by v.`length`.

So, how do we insert a new string in the trie? If the trie is *empty*, then we create a new leaf node which has both the elements of the `pair` that constitutes the `Value` field set equal to the position of the string among the copied ones; the `data` field is set equal to the last character of the string (which is the special a character mentioned before); and the `length` is set equal to the length of the string.
If the trie is *not empty*, it has a *root* r. Suppose that the string to be inserted is S; if the length of the longest common prefix between S and $\mathsf{string}(r)$, which we call p, is smaller than the `length` field of r, a mismatch is found and so a new node u such that $\mathsf{string}(u) = p$ is inserted in the tree.
Such a node has the `Value` field set to the range of copied strings prefixed by p, the `length` field equal to $|p|$, the `data` field equal to $\mathsf{string}(r)[r.\texttt{length}]$, the middle child of u is set to be r and the pointers to the left and right child are set to `NULL`.
Otherwise, that is when $\mathsf{lcp}(S, \mathsf{string}(r)) \geq r.\texttt{length}$, then we simply compare $S[\texttt{length}]$ with the `data` field of r: if they are equal we apply recursively the procedure described until now setting as root the middle child of r; if `data` is greater the root is set to be the left child of r, else it is set equal to the right one.
The following observation gives us a bound on the number of nodes belonging to the ternary search tree.

Observation 7. *If we call $\mathcal{S}$ the strings set on which we build up the PATRICIA ternary search tree $\mathcal{T}$, the total number of nodes N in $\mathcal{T}$ is bounded by $N < 2|\mathcal{S}|$*

Proof. If $\mathcal{T}$ is empty, the insertion of a string $S \in \mathcal{S}$ implies the creation of a single new node. If the trie is not empty, either we find a mismatch which implies exactly the creation

of one and only one internal node plus one leaf, or we do not find any mismatch, which implies the creation of exactly one leaf. Finding a mismatch implies the allocation of an internal node which has exactly two children: the middle one is the node in which the mismatch has been found, the right or left one is the leaf associated to S. □

The only problem left to solve in order to efficiently perform insertions in the tree is how to find the range of strings $[i, j]$ which are prefixed by $\mathsf{string}(u)$, for any node u in the tree.
Suppose that during the insertion of a string S we reach a node v such that
$\mathsf{lcp}(S, \mathsf{string}(v)) < v.\texttt{length}$, we *inductively* know that $v.\texttt{Value}$ contains the range of strings $[s, k]$ that are prefixed by $\mathsf{string}(v)$. Since the new node u to be created at this point is such that $\mathsf{string}(u)$ is a *prefix* of $\mathsf{string}(v)$, we can argue that the range $[i, j]$ of strings prefixed by $\mathsf{string}(v)$ is such that $i \leq s$ and $j \geq k$, hence we can start to scan the strings to find i from $\mathcal{C}_s$ going backwards, while we may find j starting from $\mathcal{C}_k$ going forward. Such a fact greatly reduce the number of comparison needed to find $[i, j]$.

5.2.2 Search

The algorithm to look for a pattern P in a PATRICIA ternary search tree has many analogies with the *blind search*[6] used for the same purpose in a PATRICIA trie and it as subtle as it is.

We start our search from the root r, which is the first element of the vector of nodes; we compare $P[\texttt{length}]$ with $r.\texttt{data}$ in order to move to the next node, exactly the same way we followed for insertions, e.g., given that the current node is node u, if $P[u.\texttt{length}] = u.\texttt{data}$ we do the next comparison on the middle child of u, if $P[u.\texttt{length}] > u.\texttt{data}$, we move to the right son, else we move to the left one. We stop as soon as the `length` field of the current node is greater or equal than $|P|$.

Now, if P is a prefix of one or more copied strings, then the search gives exactly the range $[i, j]$ of *copied strings* for which P is a prefix. I simply have to scan the block $(i - 1, i]$ to find the first string prefixed by P, and block $[j, j + 1]$ to find the last string prefixed by P.
Supposing that the set of copied strings is $\mathcal{C}$ and that the encoded dictionary is $\mathcal{D}$, such that $\mathcal{C} \subseteq \mathcal{D}$ and $\mathcal{C}_i = \mathcal{D}_j$ with $j \geq i$, then we can translate the indexes of $\mathcal{C}$ to the indexes of $\mathcal{D}$ using the V data structure described in Section 3.2.1. If *P is not a prefix of any copied string*, than all the strings prefixed by P are located in *one single block*, but the percolation of the tree can lead to a set of strings which is a subset of the strings sharing the longest common prefix with P.
In fact, since we are traversing a PATRICIA trie, when we move to a new node w from v,

[6] See [13] for further understanding.

it may happen that $w.\texttt{length} > v.\texttt{length} + 1$. Thus, we are not able to see any mismatch from character $P[v.\texttt{length}]$ on, but there could be one. So, to notice if there has been a mismatch we just percolate the tree until we reach a leaf, a mismatch or an internal node u such that $u.\texttt{length} \geq |P|$. We take any of the strings in the returned range, for instance S_i and we check wether $\mathsf{lcp}(S_i, P) < |P|$. If this happen than we may have *not noticed a mismatch* and the longest prefix shared between P and the copied strings can be associated to an *ancestor* of u.

Indeed we can percolate again the tree until we find a node z such that $z.\texttt{length} > \mathsf{lcp}(S_i, P)$, or store the id and the `length` field of the nodes met during the first percolation of the trie. The latter solution is better because allows to perform only *contiguous* I/Os in a very small data structure to simulate the second visit of the tree.
Now, suppose the range indicated in the `Value` field of z is $[k, s]$. It is enough to compare $P[\mathsf{lcp}(S_i, P)]$ with $S_i[\mathsf{lcp}(S_i, P)]$ in order to find the *exact* lexicographic position of P. If P is smaller than S_i then all the strings prefixed by P (if any) are in the block $(k-1, k]$, else, they can be found in the block $[s, s+1]$. We are sure that all such strings are in a single block because P is not a prefix of any copied strings, otherwise we would not have found a mismatch.

It is very important to notice that we must retrieve *only one* copied string per search and that it is enough to compare *one single character* to retrieve the exact lexicographic positions of the strings prefixed by P.
In fact, supposing that $L = \mathsf{lcp}(S_i, P)$, we have that at least the first $L+1$ characters of S_i are the same of those of S_k and of S_s (otherwise there would not have been any mismatch), indeed we have that $k \leq i \leq s$. The $L+1$-th character of those strings is the mismatch character with the pattern P, it is therefore sufficient to compare just that character to find the lexicographic position of P among the copied strings.

Figure 5.2 shows the C++ function used to implement prefix search exploiting the ternary search tree. Notice that the visit of the trie is triggered calling the procedure `Tree.It_search(P.c_str(),P.size())` which is iterative in order to save the time needed for the recursive calls. `Tree.Stack` is the data structure in which are stored the indexes of the visited nodes and their relative `length` field. We can also point out that `string Fp1` is long as $\min |P| + 1, |S_i|$ and that the scan of the block is optimised as said in observation 6 in Section 5.1.2. The function `PScanBlock()` allows to find the first and the last string that are prefixed by P if both are located in the same block, while `coder.Get_Index_of_Copied()` translates the index j of any copied string to its index i among the set of all the strings of the dictionary.

Figure 5.2: C++ function used to return the range of strings prefixed by P, exploiting ternary search trie

```
1  inline void Prefix_search(const string &P, uint64_t* Range){
2
3  pair<uint,uint> f = Tree.It_search(P.c_str(),P.size());
4
5  string Fp1 = this->Retrieval_Copied(f.first+1,P.size()+1);
6  uint CC = LcpS(Fp1.c_str(),P.c_str());
7
8  if (CC == P.size() ) {
9  Range[0] = ScanBlock(P,f.first ,this->coder.Get_Index_of_Copied(f.first+1),
       ↪ false);
10 Range[1] = ScanBlock(P+(char)0xFF,f.second+1,this->coder.Get_Index_of_Copied(
       ↪ f.second+2),true);
11 return; }
12
13 uint i=0;
14 while (Tree.Stack[i][1] <= CC){ i++;     }
15 f = Tree.V[Tree.Stack[i][0]].Value;
16 uint64_t GoC;
17
18 if ((unsigned)Fp1[CC] > (unsigned)P[CC]){
19         GoC = this->coder.Get_Index_of_Copied(f.first+1);
20         PScanBlock(P,f.first ,GoC,Range);   }
21 else {
22         GoC = this->coder.Get_Index_of_Copied(f.second+2);
23         PScanBlock(P,f.second+1,GoC,Range);   }}
```

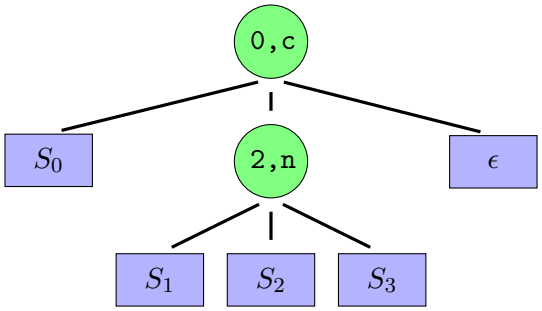


Figure 5.3: Balanced PATRICIA ternary search tree according to string set: {abaco,caccia,cane,caso}.

Chapter 6

Prefix search experiments

In this chapter we report the results of the experiments performed in order to measure the quality of the solutions of the prefix search problem proposed in the previous chapter. In particular we will compare the two-level binary search with the ternary search trie both as solution for prefix search and for Lookup[1]. We will also compare these solutions with the path decomposed tries described in Section 2.14. We anticipate that our solutions, in particular the dictionary compressed by LPFC added to the ternary search trie TST will be particularly competitive. The experiments have been divided according to the dictionary on which they were performed. Hence, in Section 6.1 we report the experiments performed on the Google 1gram dictionary described in Table 4.1, whilst in Section 6.2 we report the results of the experiments performed on the dictionary URLs dictionary described in Table 4.2.

For the experiments reported in this chapter we have choosen to set E (data structure used to store the positions of the copied strings) and V (data structure used to store the indexes of the copied strings) as plain arrays.
With V stored in such a way we avoid the Select operation that would be needed to retrieve the index of the i-th copied string among the set of all the strings, which we know it is the most expensive one. Moreover, we know that storing E and V as arrays we achieve negligible overhead in space when the blocks are sufficiently long without incurring in the performance degradation of the succinct data structures.
In the following, we illustrate the notation that will be used for the various quantities we will deal with in the experiments.

- T_{Bin}. Average search time with binary search on the copied strings.
- TF_{Bin}. Average time to locate the blocks in which the first and last string prefixed by P are located.

[1] See Section 2.8

- S_{Bin}. Space occupied by binary search data structure and the dictionary.
- T_{Tree}. Average prefix search time with ternary search trie..
- TF_{Tree}. Average time to locate the blocks in which the first and last string prefixed by P are located, with ternary search trie.
- S_{Tree}. Space occupied by ternary search trie and the dictionary.

Depending on the context, time values can refer also to $\mathsf{Lookup}(S)$, so for example TF_{Bin} can indicate the average time to locate the LPFC block in which S is located.

All the time measurements reported in this chapter are expressed in microseconds, whilst all the space measurements are expressed in MB. In the rest of the chapter we denote a dictionary $\mathcal{D}$ compressed with an algorithm $\mathfrak{C}$ exploiting a ternary search trie to support prefix search with $\mathfrak{C}(\mathcal{D})$+$\mathsf{TST}$.

In the following paragraphs we illustrate some theoretical considerations that will be useful to understand the trend of the prefix search time for both the algorithms proposed in the previous chapter.

Binary search It is possible to notice that the time to prefix search with the two-level binary search does not increase linearly with the parameter x of $\mathsf{LPFC}(x)$.
In fact, increasing x, the size of the blocks increases but the number of copied strings decreases. Actually, if we double the LPFC parameter we should have roughly on average one random I/O less that "balances" the increased number of contiguous I/Os needed to scan the block.
Given that the page size is B, if we call the set of copied strings $\mathcal{C}$ such that $|\mathcal{C}| = n_c$, we have that the time cost T_{Bin} of prefix search a patter P on a dictionary compressed by $\mathsf{LPFC}(p)$ producing blocks of average size D is bounded by the following.

$$\lceil \frac{P}{B} \rceil \log n_c + \lceil D/B \rceil \leq T_{Bin} < 2(\lceil \frac{P}{B} \rceil \log n_c + \lceil D/B \rceil)\ I/Os \tag{6.1}$$

Indeed, looking for a prefix consists in two binary search on the copied strings and in at most two block scans. Since the pivot strings[2] are always the same for the same intervals; it is very likely that some I/Os are avoided during the search for the pattern $P\#$. Moreover, the range of strings prefixed by P can obviously be *completely contained* in a single block, causing no additional I/Os during the binary search for the pattern $P\#$. There are no additional I/Os *at all* during the search for $P\#$ if P is not a prefix of any string. This last case is represented by the left term in formula 6.1. For the right term of the very same formula we used the "$<$" relation symbol instead that "$\leq$" because at least the first pivot string is the same looking for pattern P or $P\#$.

[2]The strings that are compared with the pattern P at each step of binary search.

We may also point out that is not convenient in terms of I/O to change the starting interval of binary search when we look for pattern $P\#$ because we can reuse the pivot strings previously compared with P to save I/Os. For example, suppose that the starting interval of the binary search is $[0, |\mathcal{C}|)$ and that the lexicographic position of P is $i \geq 0$, we could start to binary search for pattern $P\#$ in the interval $[i, |\mathcal{C}|)$, which is a smaller interval and therefore convenient to binary search on in terms of basic operations; anyway, we are almost sure that we have to perform at least one I/O in each step of the binary search in order to compare $P\#$ with the pivot strings.

It is interesting to calculate the difference δ between the I/Os needed to binary search a pattern on $\mathsf{LPFC}(p)$ and $\mathsf{LPFC}(2p)$. We assume that doubling the parameter we halve the number of copied strings and double the size of the blocks. So δ is given by the following.

$$\delta \approx \log n_c + \lceil \tfrac{D}{B} \rceil - \log \tfrac{n_c}{2} - \lceil \tfrac{2D}{B} \rceil = \lceil \tfrac{D}{B} \rceil - 1$$

And in general if we pass from $\mathsf{LPFC}(p)$ to $\mathsf{LPFC}(yp)$, $y \in \mathbb{N}^+$, $\delta(y)$ is approximately equal to the following.

$$\delta(y) \approx \lceil \frac{D(y-1)}{B} \rceil - (y-1) \tag{6.2}$$

Equation 6.2 explains the possible non-linear increasing in time of the two-level binary search with respect to the LPFC compression parameter. It also interesting to point out that if the size of the block is equal to the page size $D = B$ than $\delta(y) = 0 \ \forall y$. We may even gain time if $D < B$.

Ternary search tree The number of I/Os T_{tree} needed to solve prefix search with a two level scheme exploiting a ternary search tree is directly influenced by the height of the ternary search tree indexing the copied strings. In a perfectly balanced trie the expected number of comparison is equal to $|P| \log |\sigma|$, and if the ternary search tree is built picking the strings randomly, the expected number of I/Os is $2H_{n_c} + |P| + O(1)$. Where $H_n = \sum_{i=1}^{n} \frac{1}{i}$. The "tournament trie" we used in our implementation fits in between. Anyway we point out that actually $T_{tree} \leq |P| \log |\sigma|$. In fact some character comparisons can be skipped because the tree is a PATRICIA ternary search trie and it is not said that the strings use all the characters of the alphabet at each level.
A more correct bound is given in [3], stating that the maximum number of I/Os to look for a pattern in a ternary search tree is $\lfloor \log n_c \rfloor + |P|$. So we may estimate the average number of I/Os performed during a "tournament trie" percolation with the following formula.

$$2H_{n_c} + |P| + O(1) > T_{tree} \geq \lfloor \log n_c \rfloor + |P| \tag{6.3}$$

An advantage given by the PATRICIA ternary search tree is that we do not have to perform any I/O on the dictionary of strings, instead we perform I/Os only on the trie

during the first level search. It is advantageous because this data structure is much smaller than the dictionary, so it is much more likely to exploit locality during the percolation of the tree.
Locality is exploited also because the ternary search trie is stored *contiguously* in memory so that the percolation of the tree shrinks the portion of memory on which the new I/O will be performed. Indeed, if we are currently taking into account a node u which is located in position $T[i]$ in the vector of nodes, then its children are all in positions greater than i. Moreover, since the middle child is located in cell $i + 1$ it is very likely that we do not perform any new I/O accessing it.

6.1 Experiments on Google 1gram

In this section we show some experiments performed on the same file described in Section 4.2, table 4.1.
Table 6.1 shows the comparison between time and space results obtained binary searching on LPFC and prefix searching on LPFC+TST, in particular we compare the behaviour of LPFC(x) with the one of LPFC(x)+TST, varying the length of the searched pattern P. Figure 6.1 shows graphically the comparison mentioned just before. Each plot illustrates the average prefix search time in function of the pattern length. Table 6.2 shows the time to perform Lookup on LPFC+TST (which is proven to be the best solution), while Table 6.3 shows that LPFC+TST is better than path decomposed tries to encode this dictionary because it occupies less space yet providing faster Lookup and Access operations.

Figure 6.1: Comparing LPFC+TST and LPFC on prefix search efficiency.

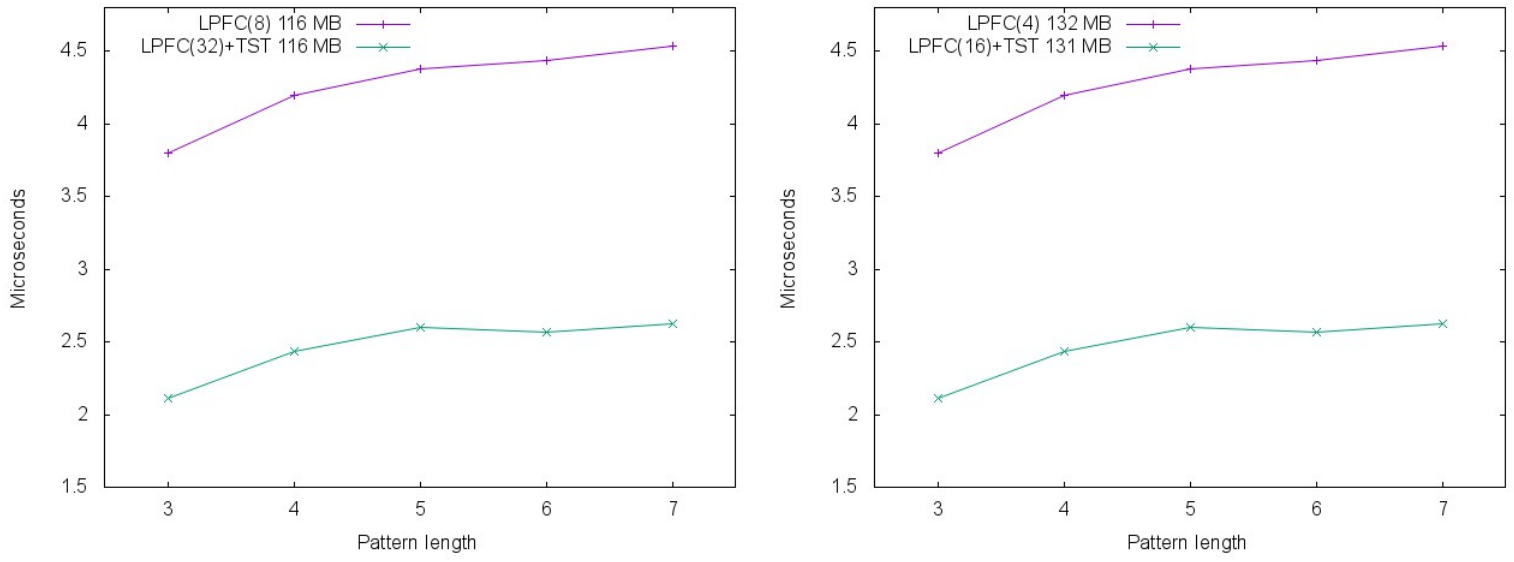


Table 6.1: Comparison between prefix search on LPFC + TST and LPFC + binary search, applied to Google 1gram. We retrieve two millions prefixes at random. Prefixes are actually chosen as prefixes of strings which are in actually the file. The length of the prefixes are indicated in the caption of every sub-table

(a) Comparison setting prefix length equal to 3

LPFC	T_{Bin}	S_{Bin}	TF_{Bin}	T_{Tree}	S_{Tree}	TF_{Tree}
4	3.206	132.07	3.181	0.812	218.62	0.191
8	3.222	115.63	2.949	1.004	160.33	0.183
16	3.342	107.67	2.631	1.389	131.39	0.182
32	3.799	103.62	2.180	2.109	116.35	0.178

(b) Comparison setting prefix length equal to 4

LPFC	T_{Bin}	S_{Bin}	TF_{Bin}	T_{Tree}	S_{Tree}	TF_{Tree}
4	3.521	132.07	3.330	1.325	218.62	0.326
8	3.535	115.63	2.990	1.357	160.33	0.298
16	3.722	107.67	2.651	1.773	131.39	0.262
32	4.195	103.62	2.339	2.431	116.35	0.243

(c) Comparison setting prefix length equal to 5

LPFC	T_{Bin}	S_{Bin}	TF_{Bin}	T_{Tree}	S_{Tree}	TF_{Tree}
4	3.605	132.07	3.208	1.326	218.62	0.422
8	3.618	115.63	2.966	1.652	160.33	0.392
16	3.840	107.67	2.641	1.901	131.39	0.378
32	4.376	103.62	2.321	2.599	116.35	0.280

(d) Comparison setting prefix length equal to 6

LPFC	T_{Bin}	S_{Bin}	TF_{Bin}	T_{Tree}	S_{Tree}	TF_{Tree}
4	3.553	132.07	3.141	1.463	218.62	0.545
8	3.634	115.63	2.922	1.585	160.33	0.430
16	3.835	107.67	2.581	1.957	131.39	0.369
32	4.433	103.62	2.278	2.568	116.35	0.314

(e) Comparison setting prefix length equal to 7

LPFC	T_{Bin}	S_{Bin}	TF_{Bin}	T_{Tree}	S_{Tree}	TF_{Tree}
4	3.607	132.07	3.144	1.419	218.62	0.515
8	3.621	115.63	2.864	1.601	160.33	0.490
16	3.868	107.67	2.553	1.955	131.39	0.376
32	4.538	103.62	2.250	2.625	116.35	0.343

Figure 6.1 proves that in 1gram prefix-searching exploiting ternary search trie is better than prefix-searching via binary search. In fact in both the case depicted in that figure, we have that LPFC+TST (locality preserving front coding plus ternary search trie) is almost twice as fast as LPFC, having approximately the same space cost. The non-monotonic trends in the curves shown in Figure 6.1 are due to the fact that during the block scanning phase needed to find the lexicographic position of our pattern P, we compare P with the currently scanned string S only if $|S| \geq |P|$[3]. Hence, increasing the cardinality of P reduces the probability of finding a string S such that $S \geq |P|$, possibly allowing to reduce the number of comparisons and of I/Os needed to find the position of P.

It is evident also thanks to Table 6.1 that ternary search tree beats two level prefix search. In fact, from that table we can argue that LPFC(16)+TST beats LPFC(4) both in time and space and LPFC(32)+TST is very close in space to LPFC(8) but it requires less time to solve prefix search.

In general we can notice from the very same table, that when binary search is considered, the time to locate the possible blocks in which the range extrema are located is very close to the overall prefix search time. This happens because the blocks to be scanned have probably already been accessed during the binary search. It is even possible, when the blocks are small enough, that during the last I/O needed to compare the pattern P with the copied string, we have already fetched all the rest of the block, causing no I/Os during the linear scan.
The situation described above does not happen when the prefixes are searched with PATRICIA ternary search trie. Indeed, the memory area of the trie is not shared with the dictionary.

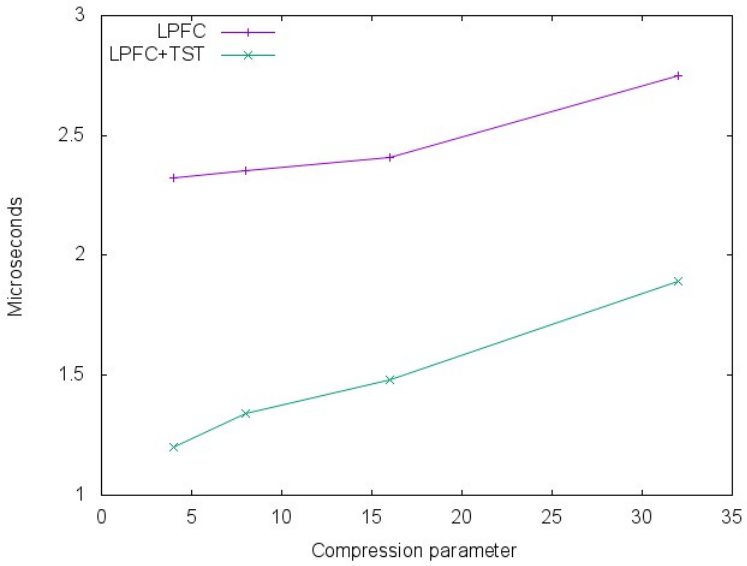
So, having understood that LPFC+TST is better than LPFC to solve prefix-search, we may see how their behaviour is when implementing Lookup. The comparison between LPFC and LPFF+TST on Lookup is illustrated in Table 6.2 and in Figure **??**.

Table 6.2: Comparison on the average time to search for a string between LPFC+two level binary search and LPFC+ternary search trie.

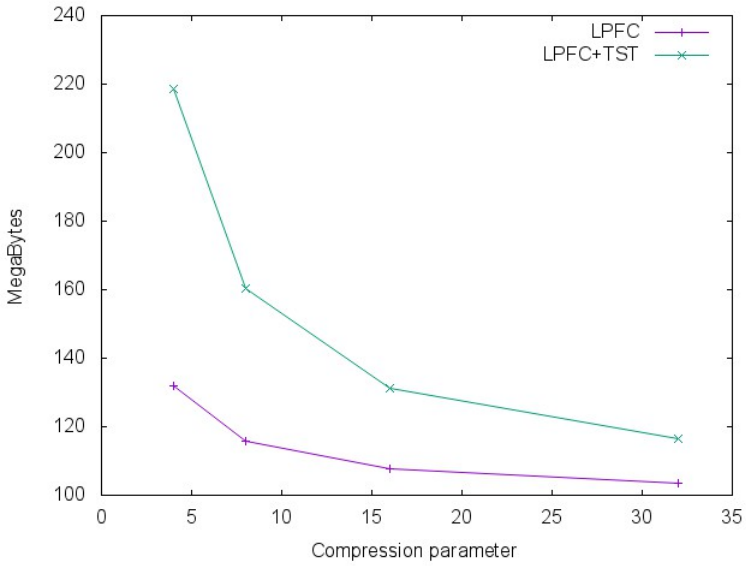
Name	T_{Bin}	S_{Bin}	TF_{Bin}	T_{tree}	S_{tree}	TF_{tree}
LPFC(4)	2.322	132.07	2.079	1.203	218.62	0.602
LPFC(8)	2.355	115.63	1.934	1.343	160.33	0.462
LPFC(16)	2.410	107.67	1.794	1.483	131.39	0.418
LPFC(32)	2.747	103.62	1.479	1.890	116.35	0.352

[3]See section 5.1.2, Observation 6.

Figure 6.2: Comparing space and time of LPFC and LPFC+TST on Lookup.



(a) Average Lookup time.



(b) Space occupancy.

We see also from Table 6.2 that still it is better to exploit ternary search tries to efficiently implement Lookup. Moreover, we can argue both from Table 6.1 and 6.2 that the percolation of the trie is *never* the bottleneck of the prefix search algorithm: most of the time is spent by the scanning of the blocks, and the percolation of the trie becomes more and more negligible as much as the size of the blocks grows. Indeed, the time to percolate the tree boils down logarithmically decreasing the number of the copied strings, while the time to scan the blocks approximately increases linearly with the reduction of the copied strings. Finally, in Table 6.3 we show the comparison between LPFC+TST and the path decomposed tries.

Table 6.3: Comparison between our data structure and others regarding dictionary operations applied to 1gram

Data structures	Lookup time (μs)	Access time μs	Space (MB)
LPFC(22)+TST	1.763	1.018	123.58
Centroid path decomposed trie	1.782	1.917	124.18
LPFC(27)+TST	1.908	1.114	119.4
Lexicographic path decomposed trie	2.133	2.211	120.06

It is possible to notice from Table 6.3 that tuning the LPFC parameter properly allows

to beat lexicographic path decomposition and centroid path decomposition both in space and in time.
Access(i) is particularly fast in our data structures because we need to access very small informative data structures to locate the block where the i-th string is located. Moreover, if we perform Access(i) and then Access(j), with $j \neq i$, it is very likely that all the data needed to complete Access(j) is already in a close level of memory, causing no additional I/Os.
In Path decomposed tries the fact above does not happen because every Access has to percolate the entire tree, which is much larger than the data structures needed to store E and V.[4]

The efficiency of the implementation of the 1gram dictionary via LPFC+TST is very remarkable. We see that with respect to lexicographic path decomposed trie we achieve with almost the same space 12% speedup in Lookup and 98% speedup in Access. It is also impressive the win against centroid path decomposed trie, because centroid path decomposition ensures optimal bounds on the height of the path decomposed tree and therefore to both Lookup and Access time.

6.2 Experiments on URLs

In this section we comment the results obtained performing string and prefix search on the URLs file described in Section 4.2, Table 4.2.
In Table 6.4 we compare the time and space values obtained with LPFC+TST and with binary search on LPFC, while in Table 6.5 we illustrate the comparison between BC and BC+BC+TST. All the data structures mentioned up to this point are compared in Figure 6.3, where we can see the behaviour of various solutions occupying similar space solving prefix search with patterns of different lengths. Table 6.6 shows instead the comparison between LPFC and LPFC+TST, but regarding the implementation of the Lookup facility. Finally, Table 6.7 illustrates the comparison between LPFC+TST (which is proven to be the best solution) and the path decomposed tries.

[4] See Section 3.2.1.

Table 6.4: Comparison between prefix search on LPFC + TST and LPFC + binary search applied to URLS. We retrieve 1 million prefixes at random. Prefixes are actually chosen as sub-strings of strings which are in the file.

(a) Prefix length equal to 11

LPFC	T_{Bin}	S_{Bin}	TF_{Bin}	T_{Tree}	S_{Tree}	TF_{Tree}
4	2.374	388.56	2.024	0.608	431.34	0.087
8	2.484	354.35	1.853	0.828	375.09	0.086
16	2.727	339.16	1.702	1.216	349.73	0.085
32	3.474	331.94	1.618	1.989	337.47	0.084

(b) Prefix length equal to 13

LPFC	T_{Bin}	S_{Bin}	TF_{Bin}	T_{Tree}	S_{Tree}	TF_{Tree}
4	2.787	388.56	2.392	0.743	431.34	0.122
8	2.885	354.35	2.328	1.082	375.09	0.110
16	3.289	339.16	2.096	1.587	349.73	0.106
32	4.172	331.94	1.947	2.712	337.47	0.111

(c) Prefix length equal to 15

LPFC	T_{Bin}	S_{Bin}	TF_{Bin}	T_{Tree}	S_{Tree}	TF_{Tree}
4	3.167	388.56	2.668	0.910	431.34	0.141
8	3.265	354.35	2.514	1.233	375.09	0.128
16	3.625	339.16	2.285	1.859	349.73	0.115
32	4.964	331.94	2.122	3.047	337.47	0.111

(d) Prefix length equal to 17

LPFC	T_{Bin}	S_{Bin}	TF_{Bin}	T_{Tree}	S_{Tree}	TF_{Tree}
4	3.277	388.56	2.767	0.946	431.34	0.147
8	3.428	354.35	2.550	1.370	375.09	0.149
16	3.900	339.16	2.315	1.932	349.73	0.138
32	5.003	331.94	2.118	3.236	337.47	0.131

Table 6.5: Comparison between prefix search on BC + TST and BC + binary search applied to URLS. Same experiment settings of Table 6.4.

(a) Prefix length equal to 11

BC	T_{Bin}	S_{Bin}	TF_{Bin}	T_{Tree}	S_{Tree}	TF_{Tree}
17	2.345	392.09	1.984	0.874	433.88	0.084
34	2.643	358.44	1.715	1.169	379.35	0.083
68	3.518	341.60	1.583	2.109	352.06	0.082
136	4.832	333.19	1.488	3.610	338.41	0.081

(b) Prefix length equal to 13

BC	T_{Bin}	S_{Bin}	TF_{Bin}	T_{Tree}	S_{Tree}	TF_{Tree}
17	2.871	392.09	2.380	1.002	433.88	0.114
34	3.113	358.44	1.992	1.454	379.35	0.107
68	3.938	341.60	1.827	2.346	352.06	0.109
136	5.602	333.19	1.673	4.142	338.41	0.105

(c) Prefix length equal to 15

BC	T_{Bin}	S_{Bin}	TF_{Bin}	T_{Tree}	S_{Tree}	TF_{Tree}
17	3.204	392.09	2.553	1.079	433.88	0.128
34	3.526	358.44	2.338	1.635	379.35	0.122
68	4.353	341.60	2.140	2.566	352.06	0.119
136	6.140	333.19	1.869	4.392	338.41	0.111

(d) Prefix length equal to 17

BC	T_{Bin}	S_{Bin}	TF_{Bin}	T_{Tree}	S_{Tree}	TF_{Tree}
17	3.283	392.09	2.629	1.101	433.88	0.143
34	3.613	358.44	2.366	1.700	379.35	0.139
68	4.419	341.60	2.190	2.699	352.06	0.123
136	6.150	333.19	1.932	4.436	338.41	0.122

Figure 6.3: Comparing LPFC and BC on prefix search time.

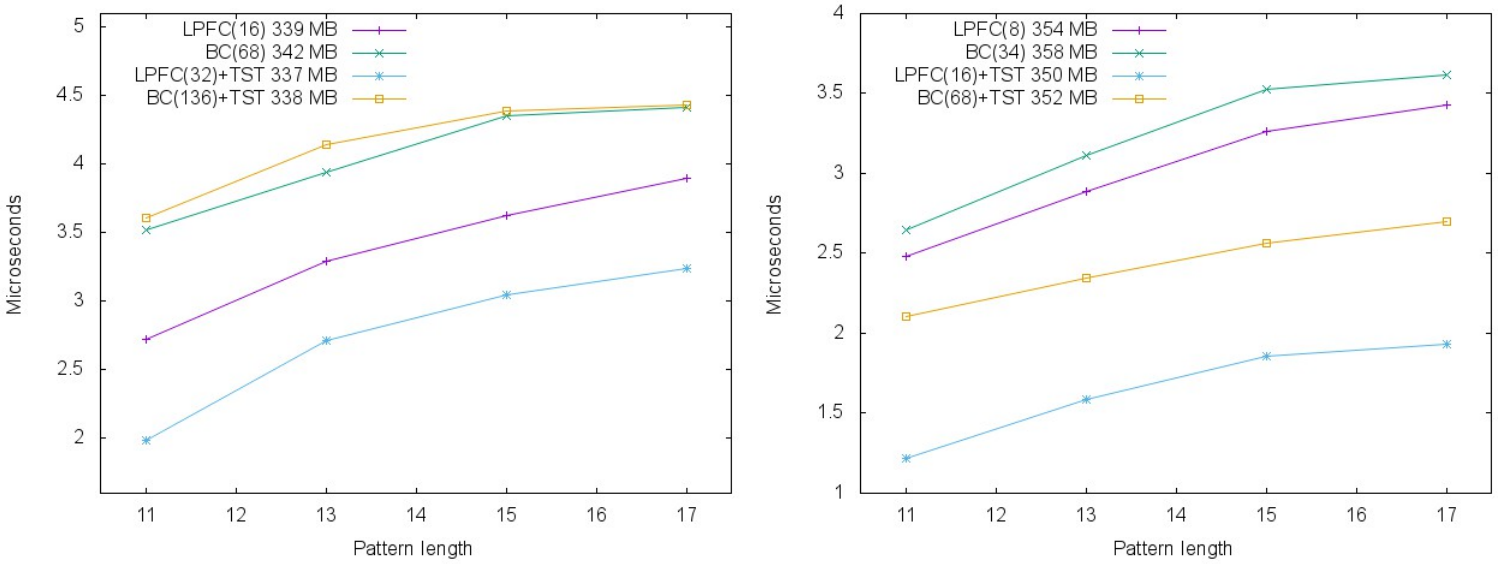
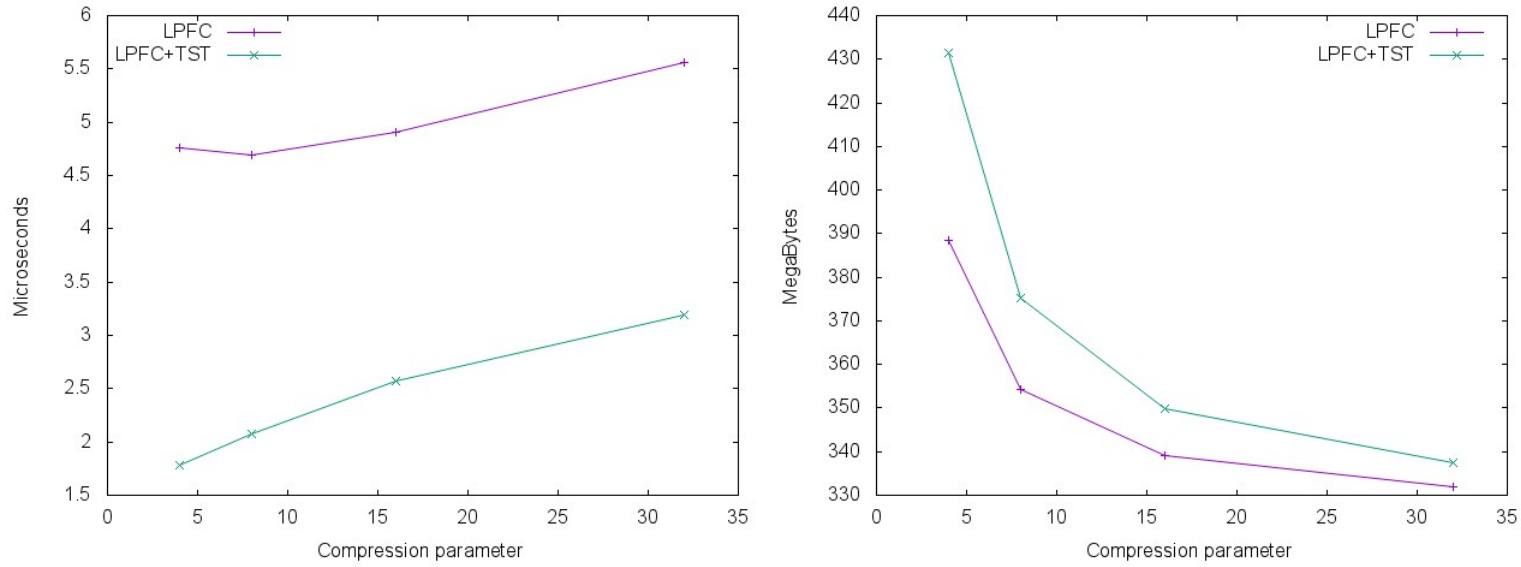


Figure 6.4: Comparing space and time of LPFC and LPFC+TST on Lookup.



(a) Average Lookup time.

(b) Space occupancy.

Table 6.6: Comparison on the average time to search for a string between LPFC+two level binary search and LPFC+ternary search trie.

Name	T^S_{Bin}	S_{Bin}	TF^S_{Bin}	T^S_{tree}	S_{tree}	TF^S_{tree}
LPFC(4)	4.758	388.56	4.509	1.778	431.34	0.633
LPFC(8)	4.693	354.35	4.097	2.077	375.09	0.599
LPFC(16)	4.902	339.16	3.842	2.567	349.73	0.473
LPFC(32)	5.557	331.94	3.470	3.193	337.47	0.389

Table 6.7: Comparison between our data structure and others regarding dictionary operations applied to URLS

Data structure	Lookup time (μs)	Access time μs	Space (MB)	CR (%)
LPFC(8)+TST	2.077	0.925	375.09	27.02
LPFC(16)+TST	2.567	1.264	349.73	25.20
LPFC(26)+TST	3.043	1.648	340.40	24.52
LPFC(32)+TST	3.193	1.862	337.47	24.31
Centroid path decomposed trie	2.437	2.710	326.22	23.50
Lexicographic path decomposed trie	3.496	3.765	325.38	23.44

We notice from Tables 6.4, 6.5 and from Figure 6.3 that LPFC is in general better than BC thanks to the fact that LPFC has smaller blocks for analogous space occupancy. It is also clear that LPFC+TST is better with respect to LPFC for prefix search.
For each pattern length taken in consideration LPFC($2x$)+TST beats LPFC(x) both in space occupancy and in prefix search time. The same can be said regarding the implementation of the Lookup operation, as we can notice from Table 6.6.
From Table 6.7 we can argue that our data structure does not clearly win over the centroid path decomposed trie, however, with a small overhead in space it is possible to obtain a data structure which is faster both in Lookup and Access.
In fact one of the best advantage of LPFC+TST is that it is a *parametric* algorithm which allows to "tune" the parameter x of LPFC(x) in order to get a good *tradeoff* between space occupancy and query time. In general with a small increase of the space occupancy we can get much better Lookup and Access time.
Access is very fast in LPFC for the same reason exposed in the previous section: the data structure used to find the exact location of the searched strings are very small and separated from the compressed string set.
If only the lexicographic path decomposed trie is taken into account, and we remind that

the comparison between LPFC+TST and lexicographic path decomposition is the most meaningful since both the data structures index the dictionary *lexicographically*, we see that with just approximately 0.9% of space overhead (with respect to the size of the not compressed URLs dictionary), LPFC(26)+TST is a bit faster in Lookup and approximately twice as fast in Access.

Chapter 7

Conclusion and future work

In this chapter we can finally draw the conclusions of this work. We started with the target of designing a compressed data structure able to store a dictionary a strings providing Lookup, Access and *prefix search* facilities. In the next section indeed we propose a brief summary of the work done in order devise such a data structure efficiently, in Section 7.2 we indicate some ideas that can inspire future works to improve our data structure, whilst in Section 7.3 I report some personal consideration about the drawing up of this Thesis.

7.1 Summing up

Here we sum up the arguments studied in this work, trying to synthetically analyse the most relevant ones and the achieved results.
In Chapter 3 we have studied some methods to compress and index any dictionary. More precisely, in Section 3.1 we analysed theoretically some of the most known and effective algorithms able to represent the strings belonging to a dictionary in a compressed fashion, finding that one of them, locality preserving front coding, ensures the *optimal* decoding of any strings of the dictionary and at the same time a clear theoretical bound on the occupied space not far from the best one achievable. Moreover, in the very same section, we studied an algorithm able to assure not only the optimal decoding of any string, but also the optimal decoding time of any *prefix* of any string belonging to the dictionary, yet guaranteeing space bounds very close to the ones of locality preserving front coding.
After that, in Section 3.2.1, we studied several ways to support the Access and the Retrieval of any string in the dictionary. We analysed the drawbacks of storing the indexes and the offsets of all the strings in plain arrays, hence we studied the advantages to store the very same quantities in succinct data structures. Then, we also analysed the possibility of storing only the positions and the indexes of the strings that are inserted *fully copied* in the compressed dictionary. We studied both the case in which such values are stored in succinct data structures and in plain arrays. At the end we came out with the fact that

the best space-time tradeoff is the latter. Indeed, storing the indexes and the positions of the uncompressed strings with two arrays gives the benefit that the size of both of them tends to decrease as much as the time to perform operations on them.
Lastly, in the same chapter, in Section 3.2.2, we also discussed how to store the encodings of the integers that must be inserted in the same memory area of the representation of the strings, ending up with the fact that good theoretical solutions are VBFast, more oriented towards decoding speed and GP, instead oriented towards the optimisation of space occupancy.

After the theoretical discussion about how to implement Access and Retrieval on any dictionary $\mathcal{D}$, in Chapter 4 we presented the results of the experiments performed on some actual dictionaries. In particular we analysed the application of our data structure to Google 1gram, a dictionary drawn from natural language, and to a subset of the ClueWebUrls dataset, a huge dictionary composed by Uniform Resource Locators (URLs). Before any experiment is performed, for both of these file we discussed what representation of the sets of positions and indexes we have to use in order to get good space-time tradeoff and what wold be the best integer encoder. All such considerations were enhanced by space and time measurements of the various supporting data structures and encoders. The outcome of the experiments has been that our data structure won against path decomposed tries both in space occupancy and in average Access time.

Continuing our dissertation, in Chapter 5 we started to study the possible approaches to solve the prefix search problem on the dictionary encoded with the techniques illustrated in Chapter 3. Initially, in Section 5.1 we described an algorithm that allows to prefix search a pattern P in $O(\text{Plog}\,|\mathcal{D}|)$ I/Os, and then, another one, exploiting a two level scheme, that ensure prefix search in $O(\text{Plog}\,n_c)$ I/Os, where n_c is the number of *uncompressed* strings in the dictionary. Both these techniques do not need the overhead of additional data structures other than the ones stored to have access to any string of the dictionary.
To further improve the prefix search time complexity, we presented in Section 5.2 a new data structure: the "tournament" PATRICIA ternary search trie, allowing to prefix search a pattern P in a time T_{tree} that is in the following range: $2H_{n_c} + |P| + O(1)$ I/Os $> T_{tree} \geq \lfloor \log n_c \rfloor + |P|$ I/Os, where $H_n = \sum_{i=1}^{n} \frac{1}{i}$, with the addition of $O(n_c)$ bits of space, hence negligible when the block of the encoded dictionary are sufficiently long.

Finally, in Chapter 6, we analysed the results of the experiments regarding the searches by prefix executed over the Google 1gram and ClueWebUrls dictionaries. We noticed that the use of the ternary search tree allowed to prefix search on those dictionaries more efficiently than binary search. We also compared our solution, now supporting Lookup and Access primitives, with the path decomposed tries, which are considered extremely efficient solutions capable of supporting both the aforementioned primitives. At the end we

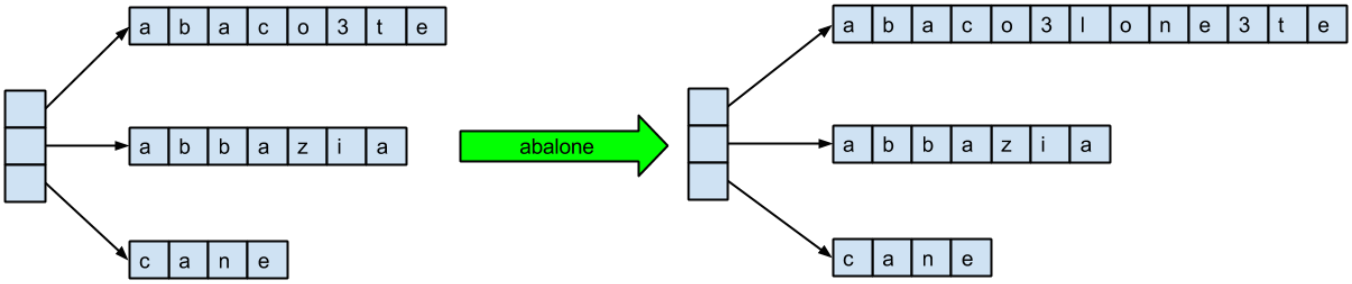
got that our data structure, basically a dictionary compressed by locality preserving front coding with two arrays storing the indexes and the positions of the uncompressed strings, plus a ternary search trie indexing them, won completely against path decomposed trie in Google 1gram, while it resulted *competitive* in ClueWebUrls. Indeed in both the cases, we overwhelmed path decomposed tries in Access time, while we obtained similar values for Lookup. The space occupied by path decomposed tries applied to ClueWebUrls was a little bit smaller than the one occupied by our data structure, anyway we pointed out the fact that our solution is *parametric*, while path decomposed tries are not. As a consequence, we can get much faster Lookup and Access time with small space growths, depending on the user's needs.

7.2 Ideas for future works

Now, after that we summed up all the work done in this Thesis, we may concentrate on what can be done in the future in order to improve our data structure. The following paragraphs show some proposals that can inspire future works.

Making a dynamic compressed dictionary Indeed the implementation of our dictionary is *static*. Given a dictionary $\mathcal{D}$, we build up a data structure capable of compressing $\mathcal{D}$ and providing prefix search and Access facilities. If we need to change, $\mathcal{D}$ by to inserting, removing, or substituting a single string, we have to build up the data structure again *from scratch*. An interesting problem would be to devise a front coded dictionary, like ours, capable of ensuring the same facilities plus the capability of inserting and removing strings from the encoded dictionary without the need to re-build the whole data structure. An idea can be to store the chunks of the front coded (or rear coded) dictionary in *different buffers*. Indeed, in our solution all the blocks are stored contiguously on memory and the insertion of a new string S in the compressed dictionary would need the shift of all the strings which are lexicographically greater than S, the update of the arrays and the reconstruction of the tree. In practice, we would have to rebuild (almost) the whole data structure from scratch.
If instead we store the compressed dictionary as a "vector" of blocks, each one starting at an arbitrary position in memory, the insertion and the deletion of the string S in the dictionary would cause only the rearrangement of a single block of the front coded dictionary, and the array of indexes should be rebuilt only in the case that the insertion of the new string breaks the rules of locality preserving front coding. At the same time, the ranges associated to the nodes of the ternary search trie must be recomputed only if the insertion or the removal cause the modification of the set of uncompressed strings.
Just to take an example of the fact that inserting a string may cause the modification of just a block, we start from the dictionary {`aback, abate, abbazia, cane`} and we

insert the string `abalone`. Once located the lexicographic position of such a string, if its insertion does not break the rule of LPFC, we have to shift only the strings which are in the same block and lexicographically greater. The following image illustrates such example.



Further improve the compression In our data structure, we exploited for compression only the *symbols* of the strings that are *shared* among several dictionary strings. We know that the best algorithm exploiting such a fact would need to store only the characteristic suffixes of the strings and separately, their longest common prefixes. Thanks to the results of [14], we also know that front coding is not far from the optimal algorithm exploiting only common prefixes.[1]

Despite this, we could achieve better compression exploiting common *sequence of symbols* which appear repeatedly in the suffixes of the compressed dictionary. For example, if there is a couple of symbols `ab` which appears frequently in the suffixes of the front coded dictionary, we can think to *augment* the the alphabet Σ_0 from which the strings are drawn, obtaining a new alphabet Σ_1, with a new symbol `A` such that $\Sigma_1 = \Sigma_0 \cup \{\texttt{A}\}$, substituting each occurrence of `ab`. Hence, we must store in the data structure also the information that `A`→`ab`.

We can repeat this procedure considering `A` as a normal symbol, that is, if we find that the sequence `Ac` appears frequently, we can define another new symbol `B` such that `B`→`Ac`.[2] We can iterate until every pair of symbols appears just *once*.

Indeed what we are doing is extracting a *grammar* $\mathfrak{G}$ generating the set of suffixes in the compressed dictionary. $\mathfrak{G}$ is such that its set of *terminal symbols* are the ones in the original alphabet Σ_0, its *metasymbols* are the ones with which we augmented Σ_0, while its set of *rules* is the one generated by the substitution of a couple of symbols belonging to Σ_{i-1} with one symbol of Σ_i.

A studied algorithm, devised by Larsson and Moffat [26] able to perform this kind of compression is called Re-Pair. There exists also an approximated version [8] that requires a tuning parameter allowing the user to trade between the compression ratio and the compression speed/memory usage.

[1] See Formula 3.2 in Section 3.1.1.

[2] Passing from Σ_1 to Σ_2.

7.3 What did I learn

At the end of work done for this Thesis I can say that I achieved a deeper understanding of how actually computer science research is. Without taking into account the precise field of this work, I touched with my hands what the study and the design of a data structure is. The mixture of theoretical and technical work constantly needed in order to achieve satisfactory results and the continuous comparisons with other devised solutions and ideas. I noticed that each phase is essential. It is essential to have clear in mind what must be done, the abstract data structure and the theoretical study of the space-time complexity. It is essential to code properly, accurately designing every detail of the algorithm and of the structures involved, not only for the performance of the data structure but also for the performance of the programmer.
I learned a lot of things about C++, most of all thanks to the code developed by more experienced scholars. I also learned a lot about performance engineering and its related technical tools like `perf`, needed to understand what is actually happening during the execution of the code.
I learned that the theoretical study of an algorithm is fundamental, but we need always to understand what is actually happening on our machine, which is the main source of performance degradation , i.e., the *bottleneck*, of our application and try to get rid of it.
I learned that it is essential the phase in which we *present* our work, not only to allow other people to know what has been done, but also to allow ourselves to deeper understand what have we really done and to tide our ideas. To give dignity to our work.

Bibliography

[1] Michael A. Bender and Martin Farach-Colton. The LCA problem revisited. In *Proceedings of the 4th Latin American Symposium on Theoretical Informatics, LATIN, Punta del Este, Uruguay*, pages 88–94, 2000.

[2] Michael A. Bender, Martin Farach-Colton, and Bradley C. Kuszmaul. Cache-oblivious string B-trees. In *Proceedings of the Twenty-Fifth ACM, SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, PODS, Chicago, Illinois, USA*, pages 233–242, 2006.

[3] Jon Louis Bentley and James B. Saxe. Algorithms on vector sets. *ACM SIGACT News*, 11(2):36–39, September 1979.

[4] Jon Louis Bentley and Robert Sedgewick. Fast algorithms for sorting and searching strings. In *Proceedings of the 8th Annual ACM-SIAM Symposium on Discrete Algorithms, SODA, New Orleans, Louisiana, USA*, pages 360–369, 1997.

[5] Rene De La Briandais. File searching using variable length keys. In *Proceedings of the Western Joint Computer Conference, San Francisco, California, USA*, IRE-AIEE-ACM '59 (Western), pages 295–298, 1959.

[6] Michael Burrows and David J. Wheeler. A block-sorting lossless data compression algorithm. Technical report, Digital Equipment Corporation, 1994.

[7] David Richard Clark. *Compact Pat Trees*. PhD thesis, Waterloo, Ontario, Canada, 1998. UMI Order No. GAXNQ-21335.

[8] Francisco Claude and Gonzalo Navarro. Fast and compact web graph representations. *ACM Transaction on the Web*, 4(4):16:1–16:31, September 2010.

[9] Erik D. Demaine, John Iacono, and Stefan Langerman. Worst-case optimal tree layout in a memory hierarchy. *CoRR*, cs.DS/0410048, 2004.

[10] Sheri Edwards. Thomas M. Cover and Joy A. Thomas, Elements of Information Theory (2nd ed.), john wiley & sons, inc. (2006). *IPM*, 44(1):400–401, 2008.

[11] Peter Elias. Efficient storage and retrieval by content and address of static files. *Journal of the ACM*, 21(2):246–260, 1974.

[12] Robert Mario Fano. *On the Number of Bits Required to Implement an Associative Memory*. Computation Structures Group Memo. MIT Project MAC Computer Structures Group, 1971.

[13] Paolo Ferragina and Roberto Grossi. The string B-tree: A new data structure for string search in external memory and its applications. *Journal of the ACM*, 46(2):236–280, 1999.

[14] Paolo Ferragina, Roberto Grossi, Ankur Gupta, Rahul Shah, and Jeffrey Scott Vitter. On searching compressed string collections cache-obliviously. In *Proceedings of the 27th ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, PODS, Vancouver, BC, Canada*, pages 181–190, 2008.

[15] Paolo Ferragina and Rossano Venturini. Compressed cache-oblivious string B-tree. In *Proceedings of the 21st Annual European Symposium on Algorithms, ESA, Sophia Antipolis, France*, pages 469–480, 2013.

[16] Johannes Fischer and Volker Heun. Space-efficient preprocessing schemes for range minimum queries on static arrays. *SIAM, Journal on Computing*, 40(2):465–492, 2011.

[17] William B. Frakes and Ricardo Baeza-Yates, editors. *Information Retrieval: Data Structures and Algorithms*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1992.

[18] Edward Fredkin. Trie memory. *Communications of the ACM*, 3(9):490–499, September 1960.

[19] Michael L. Fredman and Dan E. Willard. Trans-dichotomous algorithms for minimum spanning trees and shortest paths. *JCSS*, 48(3):533–551, 1994.

[20] Matteo Frigo, Charles E. Leiserson, Harald Prokop, and Sridhar Ramachandran. Cache-oblivious algorithms. *Journal of the ACM, Transactions on Algorithms*, 8(1):4, 2012.

[21] Roberto Grossi and Giuseppe Ottaviano. Design of practical succinct data structures for large data collections. In *Proceedings of the 12th International Symposium on Experimental Algorithms, SEA, Rome, Italy*, pages 5–17, 2013.

[22] Dan Gusfield. *Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology*. Cambridge University Press, New York, NY, USA, 1997.

[23] Bo-June Paul Hsu and Giuseppe Ottaviano. Space-efficient data structures for top-k completion. In *Proceedings of the 22nd International World Wide Web Conference, WWW, Rio de Janeiro, Brazil*, pages 583–594, 2013.

[24] D.A. Huffman. A method for the construction of minimum redundancy codes. *Proceedings of the IRE*, (10):1098–1101, 1952.

[25] Guy Jacobson. Space-efficient static trees and graphs. In *Proceedings of the 30th Annual Symposium on Foundations of Computer Science, FOCS, Research Triangle Park, North Carolina, USA*, pages 549–554, 1989.

[26] N. Jesper Larsson and Alistair Moffat. Offline dictionary-based compression. In *Proceedings of the IEEE Data Compression Conference, DCC*, pages 296–305, March 1999.

[27] Ming Li and Paul M. B. Vitányi. *An Introduction to Kolmogorov Complexity and Its Applications, Third Edition*. Texts in Computer Science. Springer, 2008.

[28] Giovanni Manzini. An analysis of the Burrows-Wheeler transform. *Journal of the ACM*, 48(3):407–430, 2001.

[29] Pat Morin. *Open data structures*. Web, 2014.

[30] Donald R. Morrison. PATRICIA - practical algorithm to retrieve information coded in alphanumeric. *Journal of the ACM*, 15(4):514–534, 1968.

[31] Ian Munro. Tables. In *Proceedings of the 16th Conference on Foundations of Software Technology and Theoretical Computer Science, FSTTCS, Hyderabad, India*, pages 37–42, 1996.

[32] Giuseppe Ottaviano. Path decomposed tries library. `https://github.com/ot/path_decomposed_tries`.

[33] Giuseppe Ottaviano. Succinct library. `http://github.com/ot/succinct`.

[34] Giuseppe Ottaviano and Roberto Grossi. Fast Compressed Tries through Path Decompositions. In *Proceedings of the 14th Meeting on Algorithm Engineering & Experiments, ALENEX. The Westin Miyako, Kyoto, Japan*, pages 65–74, 2012.

[35] Rajeev Raman, Venkatesh Raman, and Srinivasa Rao Satti. Succinct indexable dictionaries with applications to encoding k-ary trees, prefix sums and multisets. *Journal of the ACM, Transactions on Algorithms*, 3(4), 2007.

[36] Claude Elwood Shannon. A mathematical theory of communication. *Mobile Computing and Communications Review*, 5(1):3–55, 2001.

[37] Robert Endre Tarjan. Amortized computational complexity. *SIAM, Journal on Algebraic and Discrete Methods*, 6(2):306–318, 1985.

[38] Jeffrey Scott Vitter. Algorithms and data structures for external memory. *Foundations and Trends in Theoretical Computer Science, FTTCS*, 2(4):305–474, 2006.

[39] Ian H. Witten, Alistair Moffat, and Timothy Bell. *Managing Gigabytes: Compressing and Indexing Documents and Images, Second Edition.* Morgan Kaufmann, 1999.

[40] Jacob Ziv and Abraham Lempel. A universal algorithm for sequential data compression. *Journal of the IEEE, Transactions on Information Theory*, 23(3):337–343, 1977.