ISTITUTO DI MATEMATICA COMPUTAZIONALE

CNR, Area della Ricerca • I-56100 Pisa, Italy • Tel. +39 050 3152402 • Fax +39 050 3152333
E-mail: imc@imc.pi.cnr.it • URL: http://www.imc.pi.cnr.it

An experimental study of a compressed index

Paolo Ferragina* **Giovanni Manzini†**

B4-00-04

Abstract

The size of electronic data is currently growing at a faster rate than computer memory and disk storage capacities. For this reason *compression* appears always as an attractive choice, if not mandatory. However space overhead is not the only resource to be optimized when managing large data collections; in fact data turn out to be useful only when properly *indexed* to support search operations that efficiently extract the user-requested information.

Approaches to combine compression and indexing techniques are nowadays receiving more and more attention. A first step towards the design of a compressed full-text index achieving guaranteed performance in the worst case has been recently done in [10]. The novelty of that index resides in the careful combination of the compression algorithm proposed by Burrows and Wheeler [6] with the suffix array data structure [16]. The index is *opportunistic* in that, although no assumption on a particular fixed distribution is made, it takes advantage of the compressibility of the input data by decreasing the space occupancy at no significant asymptotic slowdown in the query performance.

In this paper we present an implementation of this index and perform an extensive set of experiments on various text collections. These experiments allow us to highlight properties and drawbacks of the proposed solution, as well as identify some interesting scenarios where this novel index may find effective application.

*Dipartimento di Informatica, Università di Pisa, Italy. E-mail: `ferragin@di.unipi.it`. Supported in part by Italian MURST project "Algorithms for Large Data Sets: Science and Engineering" and by UNESCO grant UVO-ROSTE 875.631.9.

†Dipartimento di Scienze e Tecnologie Avanzate, Università del Piemonte Orientale, Alessandria, Italy and IMC-CNR, Pisa, Italy. E-mail: `manzini@mfn.unipmn.it`. Supported in part by MURST 60% funds.

1 Introduction

The study, the design and the experimentation of methods for searching and updating text collections have attracted the attention of the algorithmic and data structural community during the last five decades. Precious ideas have been presented in the main literature—inverted lists, Patricia trees, tries, ternary search trees, suffix trees, suffix arrays, just to cite a few—and they constitute the heart of several software tools currently used for processing textual data. The research in this area has been recently re-vitalized by new interesting applications as digital libraries, office automation systems, SGML/XML tagged text collections, document and genome databases, web search engines. In most cases the text collections are so large that scan-based (i.e. grep-like) approaches are not appropriate, and data structures supporting effective and powerful search operations become mandatory.

The main idea underlying data structures for text searching is to build an *index* that allows to focus the search for a given pattern only on a small portion of the input text. The improvement in the query performance is paid by the additional space necessary to store the index. Most of the research in this field has been therefore directed to design data structures which offer a good trade-off between query and update time versus space usage. In this context compression appears always as an attractive choice, especially in the light of the significant increase in CPU speed that makes more economical to store data in compressed form than uncompressed. It goes without saying that compression may also introduce some improvements which are surprisingly not confined to the space occupancy: "*space optimization is closely related to time optimization in a disk memory*" [14].

Starting from these promising considerations, many researchers have recently tried to combine text compression with indexing techniques and searching algorithms. They have mainly investigated the *compressed matching problem* under various compression schemes: for example LZ77 [8], LZ78 [1], Huffman [19], Antidictionaries [7]. Although these algorithms result asymptotically faster than the classical scan-based methods, their overall time requirement may be yet too high since they rely on a full scan of the compressed text.

Some authors have tried to plug classical indexing tools—like inverted lists [22] or suffix arrays [18]—upon compressed texts and achieved experimental trade-offs between space occupancy and query performance (see e.g. Glimpse [17]). Other authors [12, 15, 20] have instead proposed techniques to represent succinctly the index itself and still support effective search operations; however, the space occupancy of their data structures grows linearly with the size of the indexed text.

The first step towards the design of a *compressed index* ensuring effective search performance in the worst case has been recently pursued in [10]. The novelty of the approach in [10] resides in the careful combination of the Burrows-Wheeler compression algorithm [6] with the suffix array data structure [16] to obtain a sort of *compressed suffix array* (see Section 2). The resulting index is *opportunistic* in that, although no assumption on a particular fixed distribution is made, it takes advantage of the compressibility of the input data by decreasing the space occupancy at no significant asymptotic slowdown in the query performance. More precisely in [10] it is proven that the space required to index a text T is $O(H_k(T))+o(1)$ bits per text character, where $H_k(T)$ is the k-th order empirical entropy of T (the bound holds for any fixed $k \geq 0$). We point out that this index also includes the compressed text. The index allows to count the number of occurrences of an *arbitrary pattern* $P[1,p]$ in $O(p)$ time, and list them in $O(\log^{\epsilon} u)$ time per occurrence, where $\epsilon > 0$ is an arbitrarily fixed constant. Since this is a *Full-text* index and occupies *Minute* space, in the following it will be shortly called *FM-index*.

Notice that there exists in the literature another family of indices, called *word-based* indices, which includes for example inverted lists and signature files [22]. Although much compact in space, these indices support only *word-based* queries so that their effective application is limited to linguistic texts. Full-text indices are more flexible. For example, they allow to search for arbitrary substrings in text collections—like DNA sequences or oriental languages—where *word delimiters* are not so clear.

Given the appealing asymptotical performance and structural properties of the FM-index, it is interesting to investigate its behavior in an experimental setting. In this paper we describe an implementation of this index and perform an extensive set of experiments on various kinds of texts: plain text, DNA

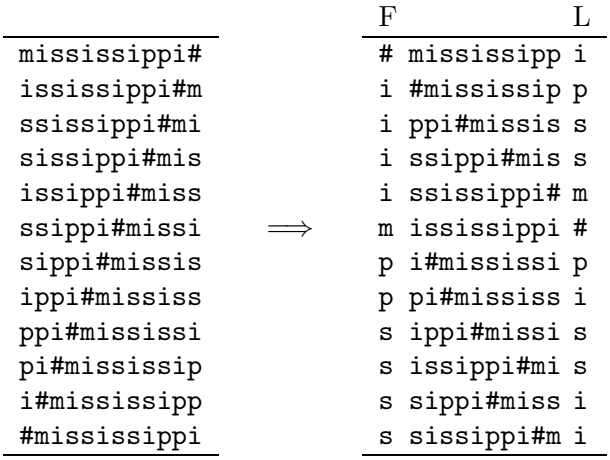

Figure 1: Example of Burrows-Wheeler transform for the string $T =$ `mississippi`. The matrix on the right has the rows sorted in lexicographic order. The output of the BWT is column L; in this example the string `ipssm#pissii`.

sequence, SGML-tagged file, `.html` and `.java` source. These experiments show that the FM-index is compact (its space occupancy is close to the one achieved by the best known compressors), it is fast in counting the number of pattern occurrences, and the cost of their retrieval is reasonable when they are few (i.e. in case of a selective query). In addition, our experiments show that the FM-index is flexible in that it is possible to trade space occupancy for search time by choosing the amount of auxiliary information stored into it.

The paper is organized as follows. In the next section we introduce some basic notation and definitions as well as the basic search operations on the FM-index. In Section 3 we propose an implementation of the FM-index which we test in Section 4. Concluding remarks are given in Section 5 together with a few examples of applications of our index.

2 Background

Let $T[1, u]$ denote a text over the alphabet Σ. The Burrows-Wheeler compression algorithm is based on a reversible transformation, called *BW-Transform* (BWT from now on) which transforms the input text T into a new string which contains the same characters but it is usually easier to compress. The BWT consists of three basic steps (see Fig. 1): (1) append to the end of T a special character # smaller than any other text character; (2) form a *conceptual* matrix $\mathcal{M}$ whose rows are the cyclic shifts of the string $T\#$ sorted in lexicographic order; (3) construct the transformed text L by taking the last column of $\mathcal{M}$. Notice that every column of $\mathcal{M}$, hence also the transformed text L, is a permutation of $T\#$. In particular the first column of $\mathcal{M}$, call it F, is obtained by lexicographically sorting the characters of $T\#$ (or, equally, the characters of L). The transformed string L usually contains long runs of identical symbols and therefore can be efficiently compressed using move-to-front coding [4], in combination with statistical coders (see for example [6, 9]).

Note that when we sort the rows of $\mathcal{M}$ we are essentially sorting the suffixes of T. Hence, there is a strong relation between the matrix $\mathcal{M}$ and the suffix array of T. This relationship is a central concept in the design of the FM-index. The matrix $\mathcal{M}$ has also other remarkable properties; to illustrate them we introduce the following notation:

- for $c \in \Sigma$ let $C[c]$ denote the total number of occurrences in T of the characters which are alphabetically smaller than c.
- for $c \in \Sigma$ let $\mathsf{Occ}(c, k)$ denote the number of occurrences of c in the prefix $L[1, k]$ of the transformed text L.

As an example, in Fig. 1 we have $C[\mathtt{s}] = 8$ and $\mathsf{Occ}(\mathtt{s}, 10) = 4$. The following properties of $\mathcal{M}$ have been proven in [6]:

Algorithm $\mathsf{count}(P[1,p])$

1. $i = p$, $c = P[p]$, $sp = C[c] + 1$, $ep = C[c+1]$;
2. **while** $((sp \leq ep)$ **and** $(i \geq 2))$ **do**
3. $\quad c = P[i-1]$;
4. $\quad sp = C[c] + \mathsf{Occ}(c, sp-1) + 1$;
5. $\quad ep = C[c] + \mathsf{Occ}(c, ep)$;
6. $\quad i = i - 1$;
7. **if** $(ep < sp)$ **then return** "not found" **else return** "found $(ep - sp + 1)$ occurrences".

Figure 2: Algorithm count for computing the number of occurrences of $P[1,p]$ in $T[1,u]$.

a. Given the ith row of $\mathcal{M}$, its last character $L[i]$ precedes its first character $F[i]$ in the original text T.

b. Let $LF(i) = C[L[i]] + \mathsf{Occ}(L[i], i)$. The character in the first column F corresponding to $L[i]$ is located in position $LF(i)$. For example, in Fig. 1 we have $LF(10) = C[\mathtt{s}] + \mathsf{Occ}(\mathtt{s}, 10) = 12$. Indeed, both $L[10]$ and $F[12]$ correspond to the first `s` in `mississippi`. We call $LF(\cdot)$ the *LF-mapping* (Last-to-First column mapping).

c. If $T[k]$ is the ith character of L then $T[k-1] = L[LF(i)]$. For example, in Fig. 1 $T[3]$ is the 10th character of L and we correctly have $T[2] = L[LF(10)] = L[12] = \mathtt{i}$.

The FM-index consists of a compressed representation of the transformed string L together with some auxiliary information. We point out that from the compressed representation of L, it is possible to get back the original text T by exploiting repeatedly Property **c**. In the next section we describe a practical implementation of the FM-index. To help the reader in following the description we now give a high level overview of the two basic search procedures supported by the FM-index: count and locate.

Procedure count takes as an input a pattern $P[1,p]$ and returns the number of occurrences of P in T. count exploits two nice structural properties of the matrix $\mathcal{M}$: (i) all the suffixes of the text $T[1,u]$ prefixed by a pattern $P[1,p]$ occupy a contiguous set of rows of $\mathcal{M}$; (ii) this set of rows has starting position sp and ending position ep, where sp is the *lexicographic position* of the string P among the ordered rows of $\mathcal{M}$. count determines the positions sp and ep via p phases, each one preserving the following invariant: *At the i-th phase, the parameter sp points to the first row of $\mathcal{M}$ prefixed by $P[i,p]$ and the parameter ep points to the last row of $\mathcal{M}$ prefixed by $P[i,p]$* (see the pseudo-code in Fig. 2). After the final phase, sp and ep will delimit the portion of $\mathcal{M}$ containing all the text suffixes prefixed by P. The integer $(ep - sp + 1)$ will therefore account for the total number of occurrences of P in T. For example, in Fig. 1 for the pattern $P = \mathtt{si}$ we have $sp = 9$ and $ep = 10$ for a total of two occurrences. In [10] it is shown how to compute $\mathsf{Occ}(c,k)$ in constant time, so computing $\mathsf{count}(P[1,p])$ takes $O(p)$ time in the worst case.

Procedure locate takes as an input the index i of a row of the matrix $\mathcal{M}$ and returns the starting position in T of the suffix corresponding to $\mathcal{M}[i]$ (in the following we write $pos(i)$ to denote such a position). For example in Fig. 1 we have $pos(3) = 8$ since $\mathcal{M}[3] = \mathtt{ippi\#mississ}$ and $T[8,11] = \mathtt{ippi}$.

If one wants to compute the positions of all occurrences of a pattern $P[1,p]$ it suffices to call $\mathsf{locate}(i)$ for $i = sp, \ldots, ep$ where sp, ep are the row indexes computed by count. The basic idea described in [10] for computing $\mathsf{locate}(i)$ is the following. We *logically mark* a suitable subset of the rows of $\mathcal{M}$. For these marked rows we keep explicitly their positions in T. Therefore, if i is a marked row $pos(i)$ is directly available. If i is not marked, the procedure locate uses the LF-mapping and Property **c** above to find the row i_1 corresponding to the suffix $T[pos(i) - 1, u]$. This procedure is iterated v times until we reach a marked row i_v for which $pos(i_v)$ is available; then we set $pos(i) = pos(i_v) + v$. Because of Property **b** each LF-mapping computation requires a call to the Occ procedure and a table lookup. Hence an effective implementation of Occ and a proper marking strategy are the key ingredients for a

fast locate. In [10] two different marking strategies are described. The first one is simpler and yields a $O(\log^2 u)$ time implementation for locate. The second strategy is more complex but significantly faster: it yields a $O(\log^\epsilon u)$ time implementation for locate for any fixed $\epsilon > 0$.

3 An implementation of the FM-index

In this section we describe an implementation of the FM-index. Our implementation is based on ideas introduced in [10], but in some points we use techniques which work well in practice rather than more cumbersome techniques with guaranteed good asymptotic worst case behavior. The implementation described here will be extensively tested in Section 4.

We have seen in the previous section that for an efficient implementation of the count and locate procedures it is important to be able to efficiently compute the value $\mathsf{Occ}(c, k)$, that is, to count the number of occurrences of the character c in the prefix $L[1, k]$. To this end, we partition the string L into *superbuckets* of size ℓ_{sb}. Each superbucket is in turn partitioned into buckets of size ℓ_b (clearly ℓ_b divides ℓ_{sb}). For each superbucket we store a table containing for each character $c \in \Sigma$ the number of its occurrences since the beginning of the string L. In other words, for the superbucket S_i we store the number of occurrences of c in $S_1, S_2, \ldots, S_{i-1}$. Similarly, for each bucket we store the number of occurrences of every character since the beginning of its superbucket. Using this auxiliary information we can easily compute the number of occurrences of any given character from the beginning of L up to the beginning of a bucket. In order to efficiently compute the number of occurrences *inside* a bucket, instead of compressing the string L as a single unit, we compress each bucket separately.

Summing up, the overall structure of the FM-index is the following:

- The *superbuckets section* which contains for each superbucket the number of occurrences of every character in the previous superbuckets.
- The *bucket directory* which contains the starting position of each compressed bucket in the body of the FM-index.
- The *body* of the FM-index which contains the compressed image of each bucket. The compressed image of each bucket includes an *header* containing the number of occurrences of each character since the beginning of the superbucket.

Given the above structure, in order to compute $\mathsf{Occ}(c, k)$ we first locate (using the bucket directory) the starting position of the bucket B_i containing $L[k]$. Then, we decompress B_i and count the number of occurrences of c from the beginning of the bucket up to $L[k]$. Then, from the bucket header we get the number of occurrences of c since the beginning of the superbucket. Finally, from the superbucket section we get the number of occurrences of c since the beginning of L.

In the above description we did not mention the actual size of buckets and superbuckets, neither the algorithm used to compress the single buckets. In [10] these parameters have been set to achieve effective worst case bounds. Here the choice has been made on an experimental basis and is therefore discussed in Section 4.

We have already observed that the string L is usually locally homogeneous, that is, if we look at a small portion of it we will likely see only a few distinct characters. We have taken advantage of this property as follows. For each (super)bucket we store a bitmap of the characters occurring in it. These bitmaps make it possible to quickly detect if a given character occurs in a certain (super)bucket thus possibly speeding up the computation of $\mathsf{Occ}(c, k)$. Additionally, these bitmaps allow to reduce the cost of storing the auxiliary information described above; for example the header of a bucket is restricted to the characters occurring in its superbucket.

The final point to be discussed is how we select and mark a subset of the rows of $\mathcal{M}$ as required by the procedure locate. Our first design decision on this issue has been to let the user choose the fraction f of the rows to be marked. Our second decision has been to use a marking strategy different from the one described in [10]. Since we are mainly interested in indexing text collections, we decided to take advantage

Name	Size	Content	Name	Size	Content
bible	4,047,392	King James Bible	*cantrbry*	2,821,120	Canterbury corpus
e.coli	4,638,690	DNA sequence	*ap90*	67,108,864	SGML-tagged text
world192	2,473,400	1992 CIA world fact book	*jdk13*	69,872,170	`html` and `java` sources

Table 1: Files used in our experiments. The files *bible*, *e.coli*, *world192* are from the Canterbury Corpus [2]. The file *cantrbry* is a tar archive containing the small files of the Canterbury Corpus. It includes both binary files and text files in various formats (plain text, html, c, and lisp code). The file *ap90* consists of the first 64Mb of the concatenation of the files `ap90`*MMDD*`.txt` from the TREC collection [13]. The file *jdk13* consists of the concatenation of the `.java` and `.html` files from the Java Jdk 1.3 documentation.

of the fact that the occurrences of each character in these texts are roughly equally spaced. Therefore, when we construct the index we select *one single character*, say c, which occurs with frequency close to f. Then, all the rows in $\mathcal{M}$ whose last character is c are logically marked and the starting positions in T of their corresponding suffixes are stored in an array $\mathcal{P}$ in the order they occur in $\mathcal{M}$. In particular if row j ends with c, then the position $pos(j)$ of the corresponding suffix will be stored in the entry $\mathsf{Occ}(c, j)$ of $\mathcal{P}$. At search time the locate procedure computes $pos(i)$ as follows. If $L[i] = c$ then $pos(i) = \mathcal{P}[\mathsf{Occ}(c, i)]$. Otherwise (i.e. if $L[i] \neq c$) the locate procedure iterates the LF-mapping v times until it reaches a row i_v whose last character is c (i.e. $L[i_v] = c$). Then locate returns $pos(i) = \mathcal{P}[\mathsf{Occ}(c, i_v)] + v$.

It goes without saying that this marking strategy heavily relies on the structure of the text and does not ensure good performances in the worst case, as instead guaranteed by the theoretical approach of [10]. Nonetheless the simplicity of this marking strategy, its reduced space overhead, and the *expected* regular structure of the indexed texts, drive us to think favorably of this scheme; its actual performance will be investigated and commented in the next section.

4 Experimenting the FM-index

The implementation of the FM-index described in the previous section contains several parameters: the size of buckets and superbuckets, the algorithm used for the compression of the buckets, the frequency of the marked characters, *etc.*. In this section we describe the results of an extensive set of experiments aimed at investigating the role played by each one of these parameters. We also compare the performance of the FM-index with those of other compressors and of the suffix array. We ran all the experiments on a machine equipped with a 600Mhz Pentium III processor with 512Kb L2 cache, 1 Gb RAM, and a 9.1 Gb SCSI hard disk. The operating system was Gnu/Linux Debian 2.2.

We point out that searching in the FM-index requires at run time a very small amount of internal memory. In fact, we access one bucket at a time (via the `fseek/fread` procedures) and therefore we need a constant amount of internal memory independent of the size of the indexed text.

Table 1 reports the files used in our experiments. We also make use of truncated versions of the file *ap90*: we write *ap90-N* to denote the file consisting of the first N Megabytes of *ap90*.

Compression of buckets. We have tested four different algorithms for the compression of single buckets. From the simplest to the most complex they are: Unary coding (see [9, Sect. 7.1]), Hierarchical 3-level coding (see [9, Sect. 7.2]), Arithmetic coding [23], and Huffman coding with multiple tables (following the implementation of [21]). After a few preliminary tests we discarded Unary coding and Arithmetic coding which turned out to be slower and less efficient in compression than, respectively, Hierarchical coding and Multiple Tables Huffman coding (MTH coding from now on). For this reason in the following we report the results only for Hierarchical coding, which is the fastest algorithm, and MTH coding, which compresses better.

Size of superbuckets. It should be clear from the description in Section 3 that the purpose of superbuckets is to reduce the amount of auxiliary information stored in each bucket (in each bucket we store, for each character, its number of occurrences from the beginning of the superbucket, rather than from the beginning of the file). Since each superbucket introduces some overhead of its own, in our first test we have tried to determine which is the optimal ratio between the size of buckets and superbuckets. Table 2

Superbucket size	2Kb	4Kb	8Kb	16Kb	32Kb	64Kb	128K	256Kb	512Kb	1024Kb	4096Kb
Compression ratio	41.04	35.45	33.05	32.28	32.34	32.89	33.64	34.68	36.06	37.47	41.16
Ave. count time	1.3	0.9	1.2	1.1	1.1	1.4	1.0	1.0	1.0	1.4	1.2
Ave. locate time	8.6	8.7	8.7	8.7	8.8	8.8	8.9	9.0	8.9	8.7	7.5

Table 2: Compression ratio (percentage), and average time (milliseconds) for the count and locate operations as a function of the superbucket size. The FM-index was built for the file *bible* using MTH coding, marking 2% of the input characters and adopting buckets of 1Kb. In each test, we searched 100 randomly chosen English words of length between 4 and 8, for a total of 100 count operations and 1,614 locate operations. We have repeated the same set of experiments using the file *ap90-8* obtaining a similar behavior.

MTH coding	*bible*				*ap90-8*			
Bucket size	1Kb	2Kb	4Kb	8Kb	1Kb	2Kb	4Kb	8Kb
Compression ratio	32.28	29.35	27.63	26.57	38.67	34.77	32.44	30.98
Ave. count time	1.0	1.6	2.5	4.1	1.7	2.4	3.5	6.1
Ave. locate time	7.5	10.8	17.2	29.6	5.4	7.7	12.0	20.5

Hierarchical coding	*bible*				*ap90-8*			
Bucket size	1Kb	2Kb	4Kb	8Kb	1Kb	2Kb	4Kb	8Kb
Compression ratio	40.08	37.64	36.27	35.49	48.00	44.71	42.84	41.79
Ave. count time	0.9	1.1	1.5	2.5	1.2	1.7	2.3	4.1
Ave. locate time	5.8	7.9	12.3	20.7	4.3	5.8	8.8	14.8

Table 3: Compression ratio (percentage) and average time (milliseconds) for the count and locate operations as a function of the bucket size. The first table refers to the FM-index built using MTH coding, the second one refers to Hierarchical coding. Each test consisted in searching 1,000 randomly chosen English words of length between 4 and 8. The total number of locate operations in each test was 24,434 for *bible* and 55,501 for *ap90-8*. In all tests the FM-index was built marking 2% of the input characters.

reports the results of our experiments. We see that this ratio does not significantly influence the average time of count and locate operations. We also see that very large or very small ratios yield a poor overall compression. However, there is a large range of ratios which yield a compression close to 33% (that is, the size of the FM-index is roughly one third of the original text size). In view of these results, in the rest of our experiments we will choose the sizes of buckets and superbuckets so that their ratio is 1:16.

Size of buckets. Intuitively, the finer is the bucket decomposition, the faster is the decompression of a single bucket and the worse should be the overall compression because of the larger auxiliary information kept at bucket level. Since the count and locate operations need to decompress several buckets (one bucket is decompressed at each call of the subroutine Occ, see Section 3), we would like to set the bucket size as small as possible; on the other side, in order to improve the compression ratio we would like to increase the bucket size to reduce the number of buckets and hence minimize the overall auxiliary information kept for them. Table 3 shows that there is a trade-off between compression ratio and search speed so that the choice of the appropriate bucket size clearly depends on the resource the end-user wishes to minimize.

From the results in Table 3 we see that for a fixed bucket size Hierarchical coding induces smaller average times for count and locate operations, whereas MTH coding compresses better. However, for a fair comparison it is natural to ask which is the fastest when the resulting FM-index requires a similar space occupancy. In this setting the MTH coding strategy appears to be superior. In fact the FM-index based upon Hierarchical coding needs to use buckets of size 8Kb in order to achieve a compression similar to the one obtained by the index based upon MTH coding with 1Kb buckets. But MTH coding with 1Kb buckets is faster in both the count and locate operations (in both cases by a factor roughly 2.5). In other words, the space saved by MTH coding in the compression of the single buckets makes it possible to use smaller buckets (i.e. to increase the amount of auxiliary information), and this more than compensate the slower decompression speed of MTH coding. The net result is thus a faster searching algorithm. For this reason in the following we focus only on the FM-index built using MTH coding.

Percentage of marked characters. This parameter clearly introduces a trade-off between compression and searching speed: the larger is the number of marked characters, the bigger is the space required for

		bible				*ap90-8*			
	Bucket size	1Kb	2Kb	4Kb	8Kb	1Kb	2Kb	4Kb	8Kb
1%	Compression ratio	29.54	26.61	24.89	23.89	35.71	31.81	29.48	28.02
	Ave. locate time	15.0	21.7	34.7	59.5	15.6	22.1	34.7	58.8
2%	Compression ratio	32.28	29.35	27.63	26.57	38.67	34.77	32.44	30.98
	Ave. locate time	7.5	10.8	17.2	29.6	5.4	7.7	12.0	20.5
5%	Compression ratio	40.25	37.32	35.60	34.54	46.67	42.77	40.43	38.98
	Ave. locate time	2.9	4.1	6.6	11.3	3.6	5.1	8.0	13.7
10%	Compression ratio	53.70	50.77	49.06	47.99	58.25	54.35	52.02	50.56
	Ave. locate time	1.3	1.9	3.0	5.1	1.9	2.7	4.3	7.3

Table 4: Compression ratio (percentage) and average time (milliseconds) for a locate operation as a function of bucket size and percentage of marked characters. Each test consisted in searching 1,000 randomly chosen English words of length between 4 and 8. The total number of locate operations in each test was 24,434 for *bible* and 55,501 for *ap90-8*. In all test the FM-index was built using MTH coding. The percentage of marked characters is given in the first column.

	File	*bible*	*e.coli*	*world*	*cantrbry*	*jdk13*	*ap90-8*	*ap90-16*	*ap90-32*	*ap90*
FM	Compr. ratio	32.28	33.61	33.23	46.10	21.41	38.69	37.43	36.36	35.49
index	Ave. count time	1.0	2.3	1.5	2.7	2.6	1.7	1.7	1.7	1.6
	Ave. locate time	7.5	7.6	9.4	7.1	32.8	5.4	5.3	5.5	5.3
suffix	Compr. ratio	375.00	387.50	375.00	375.00	437.50	400.00	412.50	425.00	437.50
array	Ave. search time	0.5	0.6	0.5	0.5	1.0	0.7	0.8	1.1	1.6

Table 5: Compression ratio (percentage) and average search time (milliseconds) for the FM-index and the suffix array on different types of input files. Each test consisted in searching 1,000 randomly chosen English words of length between 4 and 8 (for *e.coli* we used random DNA sequences of length between 8 and 15). In all tests the FM-index was built using 1Kb buckets compressed with MTH coding and marking 2% of the input characters. The search time for the suffix array accounts for the cost of retrieving all pattern occurrences.

storing their positions, the smaller is the number of LF-steps required for a locate operation. Of course count is not affected by the value assigned to this parameter since it does not use marked characters.

Table 4 reports the performance of the FM-index built using MTH coding and different bucket sizes and percentages of marked characters. Although we do not have sufficient data to draw a general rule, our results suggest that it is preferable to use small buckets. For example, the index built upon 1Kb buckets and 2% marked characters is more compact and supports faster searches than the index with 8Kb buckets and 5% of marked characters.

Robustness of the FM-index. Our next set of experiments are designed to test the performances of the FM-index on different types of input files. We have built the FM-index for all files in Table 1 using MTH coding, setting the bucket size to 1Kb, and marking 2% of the input characters. We emphasize that these were arbitrary choices (even if reasonable ones) since our previous experiments show that, depending on the application, other parameters could be more appropriate.

Table 5 summarizes the results of our tests. The first point to be noted is that the data for the files *ap90-N* show that the searching time is not significantly influenced by the size of the input file. In addition, for most of the files the average count and locate time have similar values, the only exception being the average locate time in *jdk13*. We believe that the reason for this exception resides in the simple marking technique described in Section 3 which does not work well for this file. The assumption that the occurrences of a given character are evenly distributed in the input file is probably not valid for *jdk13* which consists of `.html` and `.java` files. Table 5 also shows the performance of the suffix array highlighting that it is very fast in searching but requires a large space occupancy, a factor in the range $[8, 13]$ more than the FM-index.

FM-index vs gzip/bzip. We have compared the compression ratio and the (de)compression speed of the FM-index with those of gzip (the standard Unix compressor) and bzip2 (the best known compressor based on the BWT, see [21]). We have considered two versions of the FM-index: a "fat" index with 1Kb buckets and 2% marked characters (its performance is the one reported in Table 5) and a "tiny" index with 8Kb buckets and no marked characters (the tiny index supports only the count operation which

	File	*bible*	*e.coli*	*world*	*cantrbry*	*jdk13*	*ap90-8*	*ap90-16*	*ap90-32*	*ap90*
FM-index (tiny)	Compression ratio	21.09	26.92	19.62	24.02	6.94	25.06	23.93	22.96	22.14
	Construction time	2.24	2.19	2.26	2.21	3.48	2.49	2.64	2.81	3.04
	Decompression time	0.45	0.49	0.44	0.38	0.42	0.48	0.50	0.52	0.57
FM-index (fat)	Compression ratio	32.28	33.61	33.23	46.10	21.41	38.69	37.43	36.36	35.49
	Construction time	2.28	2.17	2.33	2.39	3.51	2.59	2.74	2.88	3.10
	Decompression time	0.46	0.51	0.46	0.41	0.44	0.51	0.52	0.55	0.59
bzip2	Compression ratio	20.90	26.97	19.79	20.24	7.23	27.38	27.33	27.32	27.36
	Compression time	1.16	1.28	1.17	0.89	1.52	1.16	1.17	1.16	1.16
	Decompression time	0.39	0.48	0.39	0.31	0.28	0.43	0.43	0.43	0.43
gzip	Compression ratio	29.07	28.00	29.17	26.10	10.97	37.21	37.28	37.31	37.35
	Compression time	1.74	10.48	0.87	5.04	0.39	0.96	0.97	0.97	0.97
	Decompression time	0.07	0.07	0.06	0.06	0.04	0.07	0.07	0.07	0.07

Table 6: Compression ratio (percentage) and (de)compression speed (microseconds per input byte) of the FM-index compared with those of gzip (with option -9 for maximum compression) and bzip2 (version 1.0.1). The "fat" FM-index uses 1Kb buckets and 2% marked characters, whereas the "tiny" FM-index uses 8Kb buckets and no marked characters.

takes about 6 milliseconds on average).

From the results in Table 6 we see that the tiny FM-index takes significantly less space than the corresponding gzip compressed file. In addition, for all files except *bible* and *cantrbry*, the tiny FM-index compresses better than bzip2. This may appear surprising since bzip2 is also based on the BWT and MTH coding. The explanation is that the FM-index computes the BWT for the entire file whereas bzip2 splits the input in 900Kb blocks. This compression improvement is payed in terms of speed; the FM-index is slightly slower than bzip2 in both compression and decompression, the difference being more noticeable for larger files.

We have already observed that the FM-index includes the compressed text as well as some auxiliary information for supporting the count and locate operations. The comparison of the "tiny" FM-index with bzip2 shows that the auxiliary information used by count is negligible for proper bucket sizes. On the other side, the "fat" FM-index performs slightly worse than gzip but supports both count and locate.

5 Concluding remarks

The bottom line of our extensive set of experiments is that the FM-index is compact (its space occupancy is close to the one achieved by the best known compressors), it is fast in counting the pattern occurrences, and the cost of their retrieval is reasonable when they are few (i.e. in case of a selective query). These algorithmic features make the FM-index interesting in various contexts. We briefly mention three of them.

- In CD-ROM production the space issue is the primary concern, hence the compactness of the FM-index may result useful in squeezing in small space both the text and a complete index for it. This can be done in a flexible way since the FM-index allows to trade space occupancy for search time.
- In spell checkers and virus detectors we are only interested in retrieving few occurrences, possibly a single one, from a dictionary of words. The literature about dictionary implementations is huge, and basic tools are tries, hash-tables and ternary search trees (TST) [5]. None of them, however, offer any kind of compression. Some preliminary experiments comparing the TST and the FM-index have shown that the former is up to 200 times faster in searching but occupies about 15 times more space.
- The FM-index can be used as a basic block in more complex indexing tools. In [10] it is shown how to plug the FM-index into the Glimpse tool [17]. The resulting index allows to achieve both sublinear space occupancy and sublinear search-time complexity in the worst case. Conversely, inverted lists allow to achieve only the second goal, whereas the classical Glimpse achieves both goals but under some restrictive conditions (see [3]).

As a future research we plan to investigate and implement new marking strategies which are simple and whose performance do not degenerate with biased distributions of the characters in the indexed text. This would make the locate procedure less sensitive to the text structure. Another interesting issue is the extension of the search procedure to more complex queries, like approximate matches and regular expressions searching. We believe that the structural properties of the FM-index may allow to easily adapt the algorithms known for the suffix array [11]. Finally, we are currently implementing a text retrieval system based on the combination of Glimpse and the FM-index following the ideas detailed in [10].

References

[1] A. Amir, G. Benson, and M. Farach. Let sleeping files lie: Pattern matching in Z-compressed files. *Journal of Computer and System Sciences*, 52(2):299–307, 1996.

[2] R. Arnold and T. Bell. The Canterbury corpus home page. `http://corpus.canterbury.ac.nz`.

[3] R. Baeza-Yates and G. Navarro. Block addressing indices for approximate text retrieval. *Journal of the American Society for Information Science*, 51(1):69–82, 2000.

[4] J. Bentley, D. Sleator, R. Tarjan, and V. Wei. A locally adaptive compression scheme. *Communication of the ACM*, 29(4):320–330, 1986.

[5] J. L. Bentley and R. Sedgewick. Fast algorithms for sorting and searching strings. In *Proceedings of the 8th ACM-SIAM Symposium on Discrete Algorithms*, pages 360–369, 1997.

[6] M. Burrows and D. Wheeler. A block sorting lossless data compression algorithm. Technical Report 124, Digital Equipment Corporation, 1994.

[7] M. Crochemore, F. Mignosi, A. Restivo, and S. Salemi. Text compression using antidictionaries. In *Proc. of International Colloquium on Automata and Languages (ICALP '99)*, pages 261–270. Springer Verlag LNCS n. 1644, 1999.

[8] M. Farach and M. Thorup. String matching in Lempel-Ziv compressed strings. *Algorithmica*, 20(4):388–404, 1998.

[9] P. Fenwick. The Burrows-Wheeler transform for block sorting text compression: principles and improvements. *The Computer Journal*, 39(9):731–740, 1996.

[10] P. Ferragina and G. Manzini. Opportunistic data structures with applications. In *Proceedings of the 41st IEEE Symposium on Foundations of Computer Science*, 2000.

[11] G. H. Gonnet, R. A. Baeza-Yates, and T. Snider. *Information Retrieval: Data Structures and Algorithms*, chapter 5, pages 66–82. Prentice-Hall, 1992.

[12] R. Grossi and J. Vitter. Compressed suffix arrays and suffix trees with applications to text indexing and string matching. In *Proceedings of the 32nd ACM Symposium on Theory of Computing*, 2000.

[13] D. K. Harman, editor. *Proc. TREC Text Retrieval Conference*. National Institute of Standards, 1992. Special Pubblication 500-207.

[14] D. E. Knuth. *Sorting and Searching*, volume 3 of *The Art of Computer Programming*. Addison-Wesley, Reading, MA, USA, second edition, 1998.

[15] V. Makinen. Compact suffix array. In *Proceedings of the 11th Symposium on Combinatorial Pattern Matching*, pages 305–319. Springer-Verlag LNCS n. 1848, 2000.

[16] U. Manber and G. Myers. Suffix arrays: a new method for on-line string searches. *SIAM Journal on Computing*, 22(5):935–948, 1993.

[17] U. Manber and S. Wu. GLIMPSE: A tool to search through entire file systems. In *Proceedings of the USENIX Winter 1994 Technical Conference*, pages 23–32, 1994.

[18] E. Moura, G. Navarro, and N. Ziviani. Indexing compressed text. In N. Ziviani, R. Baeza-Yates, and K. Guimarães, editors, *Proceedings of the 4th South American Workshop on String Processing*. Carleton University Press, 1997.

[19] E. Moura, G. Navarro, N. Ziviani, and R. Baeza-Yates. Fast searching on compressed text allowing errors. In *Proceedings of the 21st International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 298–306, 1998.

[20] K. Sadakane. Compressed text databases with efficient query algorithms based on the compressed suffix array. In *Proceeding of the 11th International Symposium on Algorithms and Computation (ISAAC '00)*. Springer-Verlag LNCS, 2000.

[21] J. Seward. The BZIP2 home page, 1997. `http://sourceware.cygnus.com/bzip2/index.html`.

[22] I. H. Witten, A. Moffat, and T. C. Bell. *Managing Gigabytes: Compressing and Indexing Documents and Images*. Morgan Kaufmann Publishers, Los Altos, CA 94022, USA, second edition, 1999.

[23] I. H. Witten, R. M. Neal, and J. G. Cleary. Arithmetic coding for data compression. *Communications of the ACM*, 30(6):520–540, June 1987.