to appear in *ACM Trans. on Information Systems* **7**, (July 1989)

Storing Text Retrieval Systems on CD-ROM: Compression and Encryption Considerations

Shmuel T. Klein, Abraham Bookstein, Scott Deerwester

Committee on Information Studies
University of Chicago
1100 E 57 St, Chicago, IL 60637

The first and third authors were partially supported by a fellowship of the Ameritech Foundation
The authors are members of the Textual Information Retrieval and Analysis (TIRA) research group

Abstract: The emergence of the CD-ROM as a storage medium for full-text databases raises the question of the maximum size database that can be contained by this medium. As an example, the problem of storing the Trésor de la Langue Française on a CD-ROM is examined in this paper. The text alone of this database is 700 MB long, more than a CD-ROM can hold. But in addition the dictionary and concordance needed to access this data must be stored. A further constraint is that some of the material is copyrighted, and it is desirable that such material be difficult to decode except through software provided by the system. Pertinent approaches to compression of the various files are reviewed and the compression of the text is related to the problem of data encryption: specifically, it is shown that, under simple models of text generation, Huffman encoding produces a bit-string indistinguishible from a representation of coin flips.

Categories and Subject Descriptors: E.3 E.4 H.3.2 J.5
General terms: Algorithms, Security
Additional Key Words and Phrases: Full-text storage, Huffman coding, CD-ROM, bit-maps

Authors' e-mail addresses:
Klein: tomi@cerberus.uchicago.edu
Bookstein: bkst@cerberus.uchicago.edu
Deerwester: scott@cerberus.uchicago.edu

1. Motivation and Introduction

Until a few years ago, large full-text information retrieval systems could only be operated on powerful mainframes. Then the personal microcomputer became more popular and information retrieval software was adapted to this smaller device, which, however, had only a relatively small memory and therefore could not be used for very large systems. Recently, the CD-ROM (compact disc – read only memory) optical disc medium has become widespread, permitting access by a PC to very large amounts of storage at very low cost. It is thus possible to transfer large databases from the mainframes, where they used to be kept, to the cheaper PC's. This creates new challenges of efficient data handling. On one hand, retrieval algorithms have to be improved since computation power is usually reduced; on the other hand, even though the storage capacity of a CD is huge (550 to 725 MB), it is still limited and cannot be expanded. Therefore sophisticated compression methods are now needed, perhaps more than ever. In certain cases, the additional savings of a few percent in storage space, which before may not have been considered to be worth the effort, can be critical to the task of transferring a large system to a single CD-ROM. For if, when transferring the database, there is even a very small overflow, two discs will be necessary, requiring either the addition of a new disc drive or that discs are changed physically; both solutions are inconvenient.

CD-ROMs differ from magnetic media, and the differences must be taken into account when designing retrieval algorithms. Access time is much higher and the data transfer rate is lower, suggesting that we try to minimize the number of seeks and to increase the amount of data transferred with each read operation (see Cichocki & Ziemer [10] and Christodoulakis & Ford [9]). In this paper, however, we are not concerned with creating new methods specially adapted to the new technology; rather we consider various compression techniques for the different files which typically appear in a full text information retrieval system containing a large natural language database. The obvious advantage of data compression is to reduce the size of a file. There is however also a gain in processing speed since the ef-

fect of compressing the text is to be able to transfer an increased amount of data with every read command. This is particularly important for CD-ROMs, with their slow read-operations. The time spent on decompression (which is done in the fast memory) is usually small compared to the savings in I/O operations.

The subject of this paper was directly motivated by our work with the *Trésor de la Langue Française* (TLF). The University of Chicago, by means of the project for *American and French Research on the TLF* (ARTFL), has been designated the U.S. depository of this large French database, covering the literature, history and science of France from the eighteenth century to today, though there are also some medieval texts in ancient French. The database consists of roughly 2600 texts with a total of about 112 million words. Recently, the French government, which owns the database, has decided to mount the TLF on CD-ROM — *a priori* an almost impossible task, as the text alone, without the necessary indices, now spans some 700 MB. In this paper we show that, perhaps with certain restrictions, it *can* be done, for the TLF and for other systems with similar characteristics and the same order of magnitude.

In the next section we investigate compression techniques for the most important file of any full text retrieval system: the **text**-file. Also important for us is the issue of the cryptographic security of storing the text in compressed form, as might be required for copyrighted material, and propose a new heuristic which yields both high compression and security. In Section 3 we review compression methods for other files which are typically found in a retrieval system. In the usual approach to full text retrieval, the processing of queries does not directly involve the original text files (in which key words may be located using some pattern matching technique), but rather the auxiliary **dictionary** and **concordance** files. The dictionary is the list of all the different words appearing in the text and is usually ordered alphabetically. The concordance contains, for every word of the dictionary, the lexicographically ordered list of references to all its occurrences in the text; it is accessed via the dictionary, which contains for every word a pointer to the corresponding list in the

concordance.

Finally, we also consider the compression of auxiliary files, like files of large sparse **bit-maps**. All the above methods are in process of being applied to the TLF database.

2. Storing the text

2.1 Compression and encryption

The most important file of any full text retrieval system is the text-file itself, which, because of its lack of structure, is also the most difficult to compress. There are several problems in storing the text on a CD. First, for a large full-text retrieval system, the file may be too big. The 700 MB of the text of TLF wouldn't fit on a CD, even before the concordance and the other files are added. The second problem is related to copyrighted material. Since the database is supposed to be widely distributed, one should try to prevent illegal use. For instance, the French government, which stands behind the program of mounting TLF on a CD-ROM, also wishes to restrict the printout of retrieved locations to at most 300 characters. There is therefore a need to encrypt the text such that readable text can be produced only by means of the supplied software.

One way of dealing with both of these problems is by using advanced methods for data compression. Beside the obvious advantage of reducing the size of the file, data compression also provides *de facto* data encryption. A text in natural language can be compressed by removing or at least reducing its redundancies. For example, in English, the letter `q` is almost always followed by `u`, in German `sc` is almost always followed by `h`, and in French `yi` is almost always followed either by `ons` or by `ez`. The knowledge of these redundancies is of great help when trying to decipher an encrypted text known to be in a given natural language. If the ciphertext looks like a random binary string, its decryption will be very hard without knowing the code. We are therefore looking for a compression technique, the output of which

can be considered as a good approximation to a random string.

Encryption and compression are in fact intimately related, in that it is redundancy in the text that permits decryption. More formally, suppose a string S becomes, upon compression, a string s. The source generating S has entropy H_S. If s has all the information of S, it too must have information content H_S. If s is a bit-string of length ℓ, with each bit occurring independently with probability $\frac{1}{2}$, its entropy will be ℓ bits, the maximum that a string of ℓ bits could obtain. Thus any string of less than ℓ bits will lose information, and a technique which, for a given source, produces a random bit-string also achieves maximum compression. But a random bit-string is, without further information, impossible to decrypt. Thus the objective of making the encoded text impossible to decipher and the task of producing maximum compression are equivalent. Below we shall further discuss compression and its interrelation with encryption.

2.2 Some known compression techniques

One of the best known compression techniques is due to Huffman [21], which for a given probability distribution is optimal in that it achieves a minimum redundancy code. There is however a problem in choosing the elements to be encoded. If we choose the characters, we could get at most 48% compression for English text. This is based on the assumption that in the uncompressed form we would use one byte, i.e. 8 bits, per character, and on the distribution for the 26 characters given by Heaps [20], which yields a code of 4.185 bits per character on the average; if we include spaces and punctuation signs, compression effectiveness would deteriorate. We could get 52.5% compression (3.804 bits per character) if we encode character pairs instead of isolated characters, but at the cost of a much larger table. To encode character triplets, we would need more than 17000 entries in the decoding table! Most implementations of Huffman compression, for example the scheme used in the UNIX `pack` command, are based on encoding the individual characters. Below we shall refer to such a scheme as *simple* Huffman coding.

Another popular compression technique is the Ziv & Lempel [32] method and its variants (Welch [31], Jakobsson [22]), in which a string S is compressed by replacing some of its sub-strings T by pointers to previous occurrences of T in S. The method is adaptive and thus needs only one pass over the string: the encode and decode routines dynamically construct a dictionary of sub-strings which is accessed using some hashing strategy. The UNIX `compress` command applies Ziv & Lempel compression to its argument. Since blocks of variable length are encoded, these methods often are superior to Huffman coding based on fixed length sub-strings. For example, using the Hebrew text of the Pentateuch (one byte per character), Huffman coding achieves 47.6% compression, while we get 57.6% compression with the Ziv & Lempel method.

Nevertheless, while adaptive methods are preferred in some real-time applications and for communication, they are not suitable for storing a large body of static text. In large information retrieval systems, the text is usually not decrypted sequentially from the beginning of the file, but rather short passages are decrypted at various points arrived at by means of preceding pointer manipulation. This cannot be done using the Ziv & Lempel method for two reasons: (1) the dictionary by means of which the text is decoded is not permanently stored but constructed by a sequence of operations starting with the beginning of the text; (2) identical parts of the original text do not always have identical counterparts in the compressed file. Therefore when one wishes to locate a string S in a compressed text T using some pattern matching technique, one needs first to decompress T, whereas if the compression method, like simple Huffman coding, always encodes each item in the same way, one could instead compress S and search for it in the compressed text. We thus need for our application a code which is fixed for the entire text.

Huffman codes can be adapted to improve compression if we realize that we are not bound to use fixed length blocks as basic units to be encoded. One could for example also choose some of the most frequent words (`the`, `of`, `and`, ...) or even word sequences (`of the`, `once upon a time`, ...) as a single unit, as well

as frequent word fragments (`ing`, `tion`, ...). Moreover, we can encode strings that are themselves the result of previous compression operations. For example, one could add repetition factors for long strings of blanks or zeros, indicators for capital letters (which rarely appear in the middle of a word, unless it is written only in upper case), etc. Once the set of elements to be encoded is chosen, the decomposition of the text into those elements is not always trivial because of the possibility of overlaps, but good algorithms exist (see Wagner [29], or Storer [27, Chapter 5.4]). Because of the possibility of overlaps, the problem of optimally choosing the sub-strings or other elements to encode seems not to be tractable: even if we restrict ourselves to the prefixes and suffixes of words appearing in the text, and if these are to be encoded by fixed length pointers to a table, the problem of choosing an optimal set has been shown by Fraenkel, Mor & Perl [19] to be NP-complete. With Huffman coding we have the additional complication of having a variable length encoding: for fixed length encodings, the storage savings due to the encoding of each element, which is what we are trying to maximize, is simply equal to (frequency $\times$ (length of element $-$ size of pointer)). However, for Huffman coding, the length of the codeword, which plays a role analoguous to that of the pointer for fixed length encoding, is itself related to the probability of occurrence of the encoded element; this most likely will increase the complexity of the procedure. We are thus justified in looking for heuristics which yield good compression in practical applications, as for example in Rubin [26]. Recall that our application is to a static, large full-text information retrieval system, for which the compression process is performed only once. We therefore can take as much time as needed on the fine tuning of the parameters of the compression procedures.

In spite of their optimality, Huffman codes are not very popular with programmers, for two reasons: first, the required bit-manipulations are not suitable for smooth programming in most high level languages; and second, Huffman codes are extremely error sensitive: a single wrong bit may propagate and render the bit stream after the error useless. A method is presented in [8] which overcomes the

first objection by using decoding tables which are prepared in a preprocessing stage. These tables can also be adapted to our case, where variable-length input blocks are encoded, and enable efficient high-level language implementation. As to the sensitivity of Huffman codes to errors, the physical encoding algorithm on CD-ROM takes care of it and provides error correction even if the errors occur in bursts (see Davies [11]).

2.3 Text compression yielding good encryption

The heuristic we are proposing is related to the problem mentioned above of using compression also as a data encryption method. We first need some definitions. A *dyadic* probability distribution is a distribution where each probability is an integral power of 2^{-1}. For example, the probability distribution $(2^{-2}, 2^{-2}, 2^{-2}, 2^{-3}, 2^{-4}, 2^{-4})$ is dyadic. Though a real life distribution S is rarely dyadic, we will use as an approximation to S the dyadic distribution which yields the same Huffman tree as S. In Longo & Galasso [25], the set of probability distributions over a finite alphabet is given a "pseudometric", and an upper bound is derived for the distance from any probability distribution to the dyadic distribution giving the same Huffman tree.

For n encodable objects, the Huffman algorithm assigns to every probability distribution $(p_1, \ldots, p_n)$ an integer vector of codeword lengths $(l_1, \ldots, l_n)$ such that $\sum_{i=1}^{n} p_i l_i$ is minimized over the set of l_i's for potential encodings; the l_i also satisfy the condition that $\sum_{i=1}^{n} 2^{-l_i} = 1$, that is, the code is *complete* (see Knuth [23, Exercise 2.3.4.5–3]). One can thus define the class of all probability distributions yielding the same lengths vector, and this class can be identified by the string $\langle n_1, n_2, \ldots, n_\ell \rangle$, where n_i is the number of codewords of length i. Such a string is called a *quantized source* in Ferguson & Rabinowitz [14], or simply *source* in the sequel. The following theorem brings sufficient conditions for getting a nearly random string as output from the Huffman algorithm; below we assume a source with n characters, where characters are generated independently and the probability

of the r-th character is p_r, for $\{p_r\}$ a dyadic distribution.

Lemma. *In a dyadic distribution, the elements combined at any stage in the Huffman tree construction have equal probability.*

Proof: If $n = 2$, the distribution must be $(\frac{1}{2}, \frac{1}{2})$, so the lemma is true. Suppose it is true for $n-1$, and let $(p_1, \ldots, p_n)$ be a dyadic distribution with $p_{n-1} = 2^{-j} \geq p_n = 2^{-i}$ being the two smallest probabilities. Suppose $p_{n-1} > p_n$, or equivalently $j < i$; then only in the binary representation of p_n is there a 1 in the i-th position to the right of the "binary point", so that $\sum_{t=1}^{n} p_t$ cannot possibly sum up to 1; thus $p_{n-1} = p_n$ and $p_{n-1} + p_n = 2^{-i+1}$, which is also a power of 2. Therefore the probability distribution $\{p_1, \ldots, p_{n-2}, (p_{n-1} + p_n)\}$, which is used in the next iteration of the Huffman process, is a dyadic distribution for $n-1$ elements, and the inductive hypothesis applies. ∎

Theorem. *Let Σ be an alphabet with a dyadic probability distribution such that the appearance of the elements of Σ in a text T is mutually independent. Then the Huffman encoding process on T produces a string which is indistinguishible from a random binary string with probability of a 1-bit being equal to $\frac{1}{2}$.*

Proof: We show that the digits of the Huffman encoded string can be considered as the outcome of a sequence of Bernoulli trials with probability $\frac{1}{2}$, that is $\Pr(x_i = 0 \mid x_1 \cdots x_{i-1}) = \frac{1}{2}$. Consider the i-th digit x_i of the string. The probability that this digit is 1 or 0 depends on our position p in the decoding tree after the $i-1$ first digits. If we are at the root, x_i is the first digit of the next codeword. From the lemma we know that the probabilities of all the codewords with a leading 0-digit add up to $\frac{1}{2}$; since we are assuming that the appearance of codewords is mutually independent we get $\Pr(x_i = 0 \mid S, p \text{ is the root position}) = \frac{1}{2}$, where S denotes the string of preceding characters. Suppose then that the position p within the tree after the decoding of the $i-1$ first digits is some internal node v on level ℓ of the Huffman tree, with $\ell > 0$, i.e. v is not the root. We first note that

$\Pr(x_i = 0 \mid x_1 \cdots x_{i-1}) = \Pr(x_i = 0 \mid S, x_{i-\ell} \cdots x_{i-1}) = \Pr(x_i = 0 \mid x_{i-\ell} \cdots x_{i-1})$ by the independence assumption. But our lemma asserts the last probability is $\frac{1}{2}$. Thus in fact x_i is 0 or 1 with probability $\frac{1}{2}$, independently of the current position, i.e. independently of $x_1, \ldots, x_{i-1}$. ∎

This result is quite surprising if we note that we are effectively producing a random string by concatenating variable length codewords which are very closely related: their set is finite and they form a complete prefix code. Moreover, *every* Huffman code has this property, even though the number of different Huffman codes for a given distribution may be huge. In fact, the number of complete prefix codes for a given source $\langle n_1, \ldots, n_\ell \rangle$ is shown in [17] to be

$$\prod_{i=1}^{\ell} \binom{2^i - \sum_{j=1}^{i-1} 2^{i-j} n_j}{n_i}.$$

For example, the source of the distribution of the English alphabet, as given in Heaps [20], is $\langle 0, 0, 2, 7, 7, 5, 1, 1, 1, 2 \rangle$, and there are 127,733,760 different codes for this source, all of which would yield the above randomness result if the conditions of independence and dyadicity were met.

The restriction to a dyadic distribution was needed for the proof of the theorem, but is not critical in actual applications. For an adequate approximation, it suffices to choose a large enough "alphabet", Σ, of elements which are to be encoded, and to avoid bias during the Huffman tree construction. That is, one should not, for example, systematically assign the digit 0 to the branch with lower probability and the digit 1 to the other branch at each step of the algorithm, but rather use some randomizing function for this choice, which does not affect the optimality of the Huffman code.

It is more difficult to choose Σ such that the independence condition of the theorem is fulfilled. If Σ_0 denotes the set of characters in a natural language text (including space and punctuation), its elements are strongly correlated. The main idea of our heuristic is to construct a set Σ, starting from Σ_0, by adding strings of

elements of Σ_0 which are positively correlated, i.e. a string $\sigma_1 \cdots \sigma_m \in \Sigma_0^*$ should be adjoined as element of Σ if and only if

$$\Pr(\sigma_1 \cdots \sigma_m) > \prod_{i=1}^{m} \Pr(\sigma_i). \tag{1}$$

Thus the strongest dependencies are incorporated into the alphabet itself. Extending the alphabet in this manner at once reduces dependencies between successive characters and could improve compaction effectiveness.

In practice, we would impose some small lower bound ϵ_1 on the difference of the terms in (1) in order to qualify a string as being really positively correlated. To improve compression, we also would restrict ourselves to strings $\sigma_1 \cdots \sigma_m$ the probability of which exceeds some small threshold ϵ_2. The parameters ϵ_i are a function of the total available space for the Huffman tree.

While including positively correlated strings in Σ might bring us closer to the ideal of having independent elements, it is not obvious that it also improves compression. If $\Pr(xy) > \Pr(x)\Pr(y)$ for $x, y \in \Sigma_0$, it is true that the Huffman algorithm could assign a shorter codeword to the string xy than one gets from concatenating the strings corresponding to x and y. On the other hand, we now must update the individual probabilities, which can result in longer codewords for x and y than before the update, though these now occur less frequently. Moreover, the number of elements in Σ has increased, which also may have a negative effect on compression. We thus should only include those strings in Σ that actually improve compression at least by some constant ϵ_3, so that the independence between the elements of Σ will only be approached. For a similar reason we didn't include *negatively* correlated strings, as we should if we are seeking real independence. But while the positively correlated strings will include the most frequent bigrams, trigrams, words, etc., the negatively correlated strings correspond to the rather rare character combinations. It is thus most likely not worth adding such strings, which anyway have a low probability of occurrence and therefore almost no influence on the compression efficiency.

The following algorithm is used for adjoining bigrams to Σ. The compression

obtained from Huffman coding with Σ_0 is denoted C. In the following pseudocode, we use **fi** and **od** to close **if** and **do** clauses respectively.

$\Sigma \leftarrow \Sigma_0$
for each bigram $xy \in \Sigma^2$ **do**
  if $\Pr(xy) > \max(\epsilon_2, \Pr(x)\Pr(y) + \epsilon_1)$ **then**
    $\Sigma' \leftarrow \Sigma \cup \{xy\}$
    update probabilities of x and y as elements of Σ'
    compute Huffman tree for Σ' and new compression C'
    if $C' > C + \epsilon_3$ **then**
      $\Sigma \leftarrow \Sigma'$
      $C \leftarrow C'$
    fi
  fi
od

When continuing in a similar manner to consider the trigrams, there is no need to consider all the $|\Sigma_0|^3$ possible strings. Although conceivably a string xyz could exist with both xy and yz negatively correlated but with $\Pr(xyz) > \Pr(x)\Pr(y)\Pr(z)$, it is not probable that we will find such a string with significant frequency in a natural language text. We thus can restrict ourselves to extending the bigrams chosen in the first iteration. We then pass similarly to 4-grams, 5-grams, etc. Although the potential number of n-grams is an exponentially growing function of n, the actual number of n-grams added to Σ will rapidly decrease with n, because the probabilities of the individual n-grams become smaller, and ϵ_2 will filter out most of the candidates. The heuristic is therefore certain to stop.

An alternative to this bottom-up approach would be to start top-down by including in Σ some of the most frequent word combinations, then the most frequent words, etc. This could speed up the process. In TLF for example, the word `toujours` is one of the hundred most frequent in the database, but would have been added, if at

all, only in the iteration for the 8-grams if the preceding algorithm is used. However, the most frequent word combinations are not necessarily the most correlated ones. A method for automatically generating genuine idioms and other strongly correlated expressions, even if their frequency is small, is described in [7]. Finally, the most promising method is a hybrid one, including both described above. Starting with Σ which includes the individual characters Σ_0 and some frequent words and phrases, we then progress bottom-up. Special care is then required as we update the frequencies after each new element is adjoined to Σ.

If we really were able to construct an alphabet Σ with a dyadic probability distribution and with independent elements, the encoded string would be perfectly random and its decryption would be impossible without some knowledge of the code. But even with real data, where the conditions are only approached, the compressed text will be very similar to a random string and its decryption extremely hard. In order to get a feeling of what a Huffman encoded string looks like, we have applied simple Huffman coding to the text used as input file to Knuth's TEX typesetting system for producing the final copy of this paper; the text (including the current sentence) consisted of 57537 characters, and the Huffman code was generated for 94 different characters, yielding an average codeword length of 4.863 bits. The encoded string appears in Figure 1, where the digit 1 is represented by a black dot and the digit 0 by a space.

Figure 1: *Simple Huffman encoding of English text*

2.4 The cryptographic security of Huffman codes

Even if a closer examination of the encoded string reveals some regularities, an immense effort seems necessary to conduct a cryptographic attack. We should bear in mind that we are not seeking absolute secrecy as is required, for example, in some military applications. In our case one needs only to make the cryptanalysis difficult enough, so that the cost of required manpower and computation hours exceeds the potential profit of breaking the code (see Konheim [24]). It seems that even simple Huffman coding comes close to that goal. However we also have to consider Huffman coding as a cryptographic system in which the "opponent" has access to some ciphertext and corresponding plaintext. Since the plaintexts are literary works which are available to everyone, it is possible to get the decryption of short parts of the compressed data, which might be used to guess the code.

Suppose, then, that we are given a binary string $X = x_1 x_2 \cdots x_n$ of which we know that it is the Huffman encoding of some string of characters $C = c_1 \cdots c_m$. Suppose first that simple Huffman coding was used, i.e., there is a codeword for each c_i. As any binary string can be obtained with equal likelihood as any other from *any* Huffman code, we have *a priori* no clues as to the boundaries of the codewords in X. An exhaustive search over all the partitions of X into m non-empty codewords has to consider $\binom{n-1}{m}$ cases (see, for example, Feller [13, Section II.5]). Each of these partitions defines a sequence of m binary codewords $(d_1, \ldots, d_m)$ which has to be checked to satisfy the following two conditions:

(1) the set $\{d_1, \ldots, d_m\}$ of the different codewords in the sequence is a prefix set, i.e., no d_i is the prefix of any other (the $\{d_i\}$ are not necessarily a complete code, as not all the characters of the alphabet need appear in C);

(2) the encoding defined by the sequence is consistent, that is, $c_i = c_j$ if and only if $d_i = d_j$, for all $1 \leq i, j \leq m$.

These conditions are easily checked, but the exponential number of partitions (m must be fairly large to allow decryption) renders exhaustive search impossible. Fur-

thermore, in a realistic context, the value m may also be unknown.

The number of partitions can be reduced from $\binom{n-1}{m} = O(n^m)$ to $O(k^m)$ if one guesses an upper bound k to the length of a codeword. Normally, k will be a small integer. Indeed, suppose we have an enormous corpus like TLF, with 700 million characters; then even if we encode a character which appears there only once, the length of that character's encoding will be about $\log_2(7 \times 10^8) < 30$. In actual applications the maximal depth of the Huffman code rarely exceeds 20, even for large alphabets of hundreds of characters. The longest codeword of the Huffman code generated for the 378 bigrams of English text given in [20] has 13 bits. Nonetheless, an attack with complexity $O(k^m)$ is still formidable.

A more promising attack would be to use the fact that we have fairly good knowledge of the distribution of characters in natural language text. The probabilities can be estimated from analyzing even a small text. Let p_i be the probability for letter i in the sample; we can assume that the actual probability in the full text is not much different. In any case, the length of the Huffman code for this letter has to be close to $\ell = \left[\log_2 \frac{1}{p_i}\right]$, where $[x]$ denotes the integer closest to x; thus a reasonable guess would be that the i-th letter is encoded by a string of length $\ell - 1$ or ℓ or $\ell + 1$. This reduces the number of cases to be checked to $3^{|\Sigma_0|}$, which could already be feasible, especially when the given plaintext does not include all the $|\Sigma_0|$ different letters.

In order to prevent such decryption attempts, which are all based on the assumption of simple Huffman coding, variable length input blocks should be used, as suggested in the above heuristic. This puts the additional burden on the opponent of guessing not only the partition of the ciphertext X, but also that of the plaintext C. Exhaustive search is now again ruled out, so the opponent needs a different approach. He could search in X and in C for reoccurring patterns, for example by using Weiner's [30] position tree algorithm (see also Aho, Hopcroft & Ullman [1, Section 9.5]), and then try to match the strings in X with strings of similar frequency and corresponding relative position in C. The algorithm is generally linear

in the length n of the input string (actually, one needs a quite artificial example of an input text to get an $O(n^2)$ complexity, see [1, Exercise 9.26]). Once the tree is constructed, one can easily respond to questions like: what is the longest, second longest, most frequent, etc. reoccurring substring. But this still requires some work, since a reoccurring substring is not necessarily a codeword. The first and last few bits may be respectively the suffix and the prefix of different other codewords. Another complication is the fact that the longer the substring, the smaller the corresponding probability, thus long reoccurring strings will be rather rare.

As a countermeasure to the possible analysis of the ciphertext, we suggest encrypting the Huffman code itself using some very simple method. For example, choose a secret integer constant k and replace every k-th bit of the Huffman code by its logical complement. This change cannot be detected as it does not change the distribution of 0's and 1's in the "random" Huffman encoded string; there is no space overhead and the increase in decoding time is negligible. On the other hand the complemented bits will break up regularities (and create some fake ones) rendering the analysis of the text impossible. If the opponent knows about this additional encoding scheme, however, he can guess k which must be small for the method to be effective. Thus, instead of a single constant k, we choose a vector of ℓ constants $k_1, \ldots, k_\ell$, all relatively small; the bits to be complemented are now chosen such that the lengths of the intervals between them form the periodic sequence $k_1, \ldots, k_\ell, k_1, k_2, \ldots$. For example if $\ell = 10$ and $k_i \leq 15$, this multiplies the complexity of a cryptographic attack by 10^{15}. The compressed file cannot be accessed at every bit, but only at certain points like the beginning of a sentence. Therefore the process of complementing the bits should start over again at every such entry point. An alternative, more general, approach is to choose a key of k bits and XOR successive blocks of k bits by this key. This does increase the cost of decryption, especially if the length of the key is not public.

3. Storing the concordance, dictionary and bit-maps

An efficient way of storing the concordance of a large full-text system seems to be the following. Enumerate the words sequentially from 1 to the number of words in the text, and use the index of a word W in the text as reference to its location; such references are called *coordinates* of W. This would mean for the TLF that the coordinate of a word is some number between 1 and 112 million, so that 27 bits are necessary to represent any coordinate. The expected size of the concordance of TLF would then be about 360 MB. There are however two serious objections to such an encoding. First, with a sequential numbering, there is no information as to the sentence, paragraph, chapter and even book boundaries. It would thus not be possible to process queries of the type: "retrieve all the occurrences of A and B in the same sentence", unless we have some additional information. Even queries imposing some upper bound on the number of words between A and B, which often occur when searching for an expression, may generate non-relevant results, as we are certainly not interested in locations where A appears towards the end of a paragraph and B near the beginning of the following one. It is thus preferable to describe a location by referring to the hierarchical structure of the text. Every occurrence of every word in the database can be uniquely characterized by a sequence of numbers that give its exact position in the text. In the TLF, for example, the sequence consists of the collection number c, the author number a (within the collection), the document number d (for each author), the part number p (in the document), the sentence number s (in the part) and the word number w (in the sentence). Thus the coordinate of the occurrence is the hexatuple (c, a, d, p, s, w). At first sight, such a hierarchical description seems wasteful, because each subfield of the coordinate must be large enough to accommodate the maximal values to be stored; however, a concordance of this form can be efficiently compressed.

The second objection to sequential numbering is that, perhaps surprisingly, it is not always the best way to compress the concordance. The way to retrieve information from the concordance is by reading blocks from the disc, which are

then scanned sequentially. We can thus use the sequential character of the access to refer, for certain coordinates, to the coordinate just preceding it. For instance, words tend to appear in clusters, like the word `coordinate`, various forms of which appeared twice in the preceding sentence and four times in this paragraph. Thus consecutive coordinates would often share some common prefix in a hierarchical scheme (same collection, author, document, part, etc.) and these prefixes need only be stored once. This is the prefix-omission method (POM) described in [6], where it is shown that its application to large concordances may yield storage savings that sometimes exceed those of sequential numbering.

The problem of compressing the concordance has been studied in [6]. The methods proposed are extensions of POM, which is usually applied to large dictionaries. They are based on encoding the different values in the coordinate in variable length fields, because most of these values are small. For TLF there are 20 collections; the maximal values for the other fields are: 41 for author, 52 for document, 455 for part, 31216 for sentence and 2897 for word. We would thus need $\lceil \log_2 2897 \rceil = 12$ bits to represent the largest possible value in a word-field, but only about 3% of the sentences are longer than 63 words; 99.7% of the sentences have less than 128 words, so that 7 bits instead of 12 bits are sufficient for the word-field of almost all the coordinates.

The existence of "sentences" with thousands of words is due to the fact that almost no structural information is available in the TLF. The text has just been typed in sequentially, and even the ends of paragraphs have not been marked. In the current implementation the end of sentences are actually guessed by the presence of a period, exclamation mark, etc. followed by a space. This is clearly not very accurate because of the existence of abbreviations and the French way of dealing with direct speech. "⟪`Parbleu!`⟫ `s'écria Mr. Dupont.`" will be parsed as three sentences! The fact that direct speech is usually not followed by a capitalized letter is of no help, because the TLF was originally typed in upper case characters only. This sentence defining heuristic works for the large majority of the text, but it

produces as a side-effect these super-long sentences; this effect is most pronounced for some modern poets, who use punctuation signs scarcely, if at all.

Similarly, text components with tens of thousands of sentences are usually entire documents lacking any sub-division. Only about 2% of the parts contain more than 4000 sentences, so introducing artificial part divisions for some of the longest documents might drastically reduce the maximum values that need to be stored.

The method in [6] yielding the highest compression starts as follows: for a given coordinate C, let $\ell_w, \ell_s, \ldots$ denote the length in bits of the binary representation, without leading zeros, of the values stored in the word-field, the sentence-field, ... of C. This constitutes a "length-tuple". The set of possible tuples $(\ell_c, \ell_a, \ell_d, \ell_p, \ell_s, \ell_w)$ is then sorted by decreasing frequency of occurrence of that tuple. It turns out that a relatively small number of the length-tuples, say 255 or 511, describe almost all the length combinations of coordinates in the entire concordance. Therefore every coordinate is prefixed by a fixed length header giving the index of the corresponding length-tuple in the table. If the coordinate is one of those with a rare lengths combination, it is either stored uncompressed, or processed in another way. If it is in the table, only the significant digits of the coordinate components need be stored, as the length-tuple indicates the subfield boundaries. If, for the ease of computer manipulation, we restrict ourselves in the uncompressed coordinate to fields the lengths of which are multiples of half-bytes, 8 bytes will be needed for a coordinate of the TLF. The above method yields compression close to 50% (see [6] for details), so we can expect the size of a compressed coordinate to be about 4 bytes on the average, or about 430 MB for the compressed concordance. This might still be too much, leaving only about 120 MB on the CD-ROM for the text and the other files.

A complete concordance includes every word in the text. One can achieve better savings by removing the most frequent words. These include articles, prepositions, etc., which are likely to appear almost anywhere and have little information content. For TLF, the coordinates of the hundred most frequent words (of a dictionary of

roughly 360,000 words) constitute 54% of the concordance. This does not mean that the concordance from which the coordinates of these so-called *stop-words* are removed will shrink to 46% of its previous size, because it is for these high frequency words that POM yields the best results. We can however assume that 5 bytes will be sufficient for the average coordinate of the 51.5 million non stop-words, resulting in a concordance of about 250 MB in size. It should however be noted that the removal of stop-words, while appropriate for languages like English, cannot easily be applied to languages like Hebrew, in which most of the words are homographic. For example, at the Responsa Retrieval Project (RRP), which has a database of about 60 million words of text written mainly in Hebrew and Aramaic (see for example [15] or [3]), no stop-words are defined and *all* the words are searchable, including some with hundreds of thousands of occurrences. For the French database of TLF, a request was received for locating all the occurrences of `un de ces` (the phrase "one of these") in the works of André Gide; all three keywords in this query are stop-words!

But even if we had enough space on the disc, it is not evident that in a CD-ROM application, it would be desirable to include the stop-words in the concordance. Recall that access times and transfer rates for CD-ROMs are worse than for magnetic media; the coordinates of a very frequent word may span hundreds of blocks, and having to read these blocks can seriously deteriorate the response time. It might thus be preferable for our application to process a query first by ignoring the stop-words; this typically yields up to a few hundred locations, each of which could then be scanned for occurrences of the stop-words of the query. For most queries, this approach will speed up the processing. However, for certain queries, like `un de ces`, this procedure may require a complete scan of the full text. As queries comprising only stop-words seem to be rather exceptional, this method of using pattern matching for the stop-words is a practical alternative to a method relying on the concordance alone.

The dictionary of a full text retrieval system is much smaller than the text or

the concordance, usually only a few megabytes in size. Since consecutive entries usually share some leading letters, POM yields good results here (about 40% on the dictionary of RRP). A combinatorial compression method for dictionaries has been suggested in Fraenkel & Mor [18], which achieves up to 48% savings on an English dictionary. Bratley & Choueka [2] suggest that the regular dictionary of a full text system be replaced by a so-called permuted dictionary, which is larger than the conventional one, but which enables the use of variable length "don't-care" characters in the formulation of a query. Thus prefix, suffix and infix truncated terms can be efficiently processed. The compression technique proposed in [2] for the permuted dictionary is again POM, and the compressed size of this file for TLF is about 18 MB.

For certain systems, a different approach to query processing is possible. The idea is to replace the concordance of a system having ℓ documents by a set of *bit-maps* of fixed length ℓ. Given some fixed ordering of the documents, a bit-map $B(W)$ is constructed for every distinct word W of the database, where the i-th bit of $B(W)$ is 1 if W occurs in the i-th document, and is 0 otherwise. Processing queries then reduces to performing logical OR/AND operations on binary sequences, which is efficiently done on most machines, instead of merge/collate operations on more general sequences. If we allow more complex queries that refer to the exact position of the keywords relative to each other, the concordance is still needed; however, bit-maps may be useful in this case too. The way in which bit-maps, used together with a concordance, can enhance the retrieval process for large full text systems has been studied in [5], where the maps are used to eliminate *a priori* parts of the text that cannot possibly contain a solution to the given query. Since these bit-maps are extremely sparse, they can be compressed very efficiently. For example, a hierarchical compression method for sparse bit-vectors is proposed in Vallarino [28]. The initial vector v_0 is partitioned into blocks of equal size and a new vector v_1 is constructed, with one bit for each block of v_0. A bit in v_1 is set to 0 only if the corresponding block in v_0 consists only of zeros. Then the process is repeated for

v_1, forming v_2, and so on. At each stage, when storing v_i, all blocks corresponding to 0's in v_{i+1} are dropped. The method is improved in [4] by pruning as well some of the branches of the hierarchy which ultimately point to very few 1-bits. A different method suggested in [16] combines Huffman coding with run-length coding for blocks of zeros. The methods in [4] and [16] yield compression of up to 94% on a set of bitmaps constructed at RRP.

A different kind of bit-map file is a so-called *signature*-file (see for example Faloutsos & Christodoulakis [12]). Here the text is partitioned into relatively small parts P, each of which is assigned a signature, which is a function of the words in P. Similarly, the signature of the keywords of the query is computed and compared with the elements of the signature file; this matching procedure allows us to discard a large number of non-qualifying parts. In order to minimize the probability of a false drop, the probability of 1-bits in the signature should be $\frac{1}{2}$, so the file can hardly be compressed.

The difference between the bit-map or signature file and the others is that they are not absolutely necessary to the retrieval system, but can improve processing time. Also their size is flexible; the larger we choose to build them, the better discrimination they allow and the faster the algorithms will be. The general policy for our CD-ROM application should therefore be: store the text, dictionary and concordance as efficiently as possible, then choose the parameters for bit-maps and/or signatures so as to fill up the remaining space.

Table 1: *Overview of compression methods*

File	Full size	Compression	Compressed size	References
Text	700	65%	245	[26], this paper
Concordance	400	40%	240	[6]
Dictionary	45	40%	18	[2], [18]
Bit-Maps	800	95%	40	[28], [4], [16]

Table 1 summarizes the methods discussed above. The columns entitled Full

size and Compressed size give the approximate sizes of the files for TLF in megabytes. For the bit-maps, the assumption is that the database is partitioned into 80,000 parts, to each of which corresponds one bit-position; thus a single map is about 10 Kbytes long. We further assume that these maps are constructed for 80,000 words, the others occurring rarely enough in the text so that the corresponding bit-maps can be constructed while processing a query. For this example we get a total of 543 MB, which shows that the Trésor de la Langue Française can be transferred to a single CD-ROM.

Acknowledgments: We wish to thank Donald Ziff, head programmer for the ARTFL project, for his participation in discussions preliminary to this paper, and for extracting and providing crucial information from the TLF. We are also indebted to the members of the *Institut National de la Langue Française* for posing the problem that motivated our research.

References

[1] **Aho A.V., Hopcroft J.E., Ullman J.D.,** *The Design and Analysis of Computer Algorithms*, Addison-Wesley, Reading, MA (1974).

[2] **Bratley P., Choueka Y.,** Processing truncated terms in document retrieval systems, *Inf. Processing & Management* **18** (1982) 257–266.

[3] **Choueka Y.,** Full text systems and research in the humanities, *Computers and the Humanities* **XIV** (1980) 153–169.

[4] **Choueka Y., Fraenkel A.S., Klein S.T., Segal E.**, Improved hierarchical bit-vector compression in document retrieval systems, *Proc. 9-th ACM-SIGIR Conf.*, Pisa (1986) 88–97.

[5] **Choueka Y., Fraenkel A.S., Klein S.T., Segal E.**, Improved techniques for processing queries in full-text systems, *Proc. 10-th ACM-SIGIR Conf.,* New Orleans (1987) 306–315.

[6] **Choueka Y., Fraenkel A.S., Klein S.T.,** Compression of concordances in full-text retrieval systems, *Proc. 11-th ACM-SIGIR Conf.,* Grenoble (1988) 597–612.

[7] **Choueka Y., Klein S.T., Neuvitz E.,** Automatic retrieval of frequent idiomatic and collocational expressions in a large corpus, *J. Assoc. Literary and Linguistic Computing*, Vol. **4** (1983) 34–38.

[8] **Choueka Y., Klein S.T., Perl Y.**, Efficient variants of Huffman codes in high level languages, *Proc. 8-th ACM-SIGIR Conf.,* Montreal (1985) 122–130.

[9] **Christodoulakis S., Ford,** Analysis of retrieval performance and fundamental performance tradeoffs for CLV optical discs, *Proc. ACM-SIGMOD Conference* (1988).

[10] **Cichocki E.M., Ziemer S.M.,** Design considerations for CD-ROM retrieval software, *J. Amer. Soc. Inf. Sc.* **39** (1988) 43–46.

[11] **Davies D.H.,** The CD-ROM medium, *J. Amer. Soc. Inf. Sc.* **39** (1988) 34–42.

[12] **Faloutsos C., Christodoulakis S.,** Signature files: An access method for documents and its analytical performance evaluation, *ACM Trans. on Office Inf. Systems* **2** (1984) 267–288.

[13] **Feller W.,** *An Introduction to Probability Theory and Its Applications*, Vol **I**, John Wiley & Sons, Inc., New York (1950).

[14] **Ferguson T. J., Rabinowitz J. H.**, Self-synchronizing Huffman codes, *IEEE Trans. on Inf. Th.* **IT–30** (1984) 687–693.

[15] **Fraenkel A.S.,** All about the Responsa Retrieval Project you always wanted to know but were afraid to ask, expanded summary, *Jurimetrics J.* **16** (1976) 149–156.

[16] **Fraenkel A.S., Klein S.T.**, Novel compression of sparse bit-strings, *Combinatorial Algorithms on Words*, NATO ASI Series Vol **F12**, Springer Verlag, Berlin (1985) 169–183.

[17] **Fraenkel A.S., Klein S.T.,** Bidirectional Huffman coding, Tech. Rep. CS87–02, The Weizmann Institute of Science (1987), submitted for publication.

[18] **Fraenkel A.S., Mor M.,** Combinatorial compression and partitioning of large dictionaries, *The Computer Journal* **26** (1983) 336–343.

[19] **Fraenkel A.S., Mor M., Perl Y.,** Is text compression by prefixes and suffixes practical? *Acta Informatica* **20** (1983) 371–389.

[20] **Heaps H.S.,** *Information Retrieval, Computational and Theoretical Aspects*, Academic Press, New York (1978).

[21] **Huffman D.,** A method for the construction of minimum redundancy codes, *Proc. of the IRE* **40** (1952) 1098–1101.

[22] **Jakobsson M.,** One pass text compression with a subword dictionary, *J. Amer. Soc. for Inf. Sc.* **39** (1988) 262–269.

[23] **Knuth D.E.,** *The Art of Computer Programming, Vol* **I**, *Fundamental Algorithms*, Addison-Wesley, Reading, Mass. (1973).

[24] **Konheim A.G.,** *Cryptography, A Primer,* John Wiley & Sons, New York (1981).

[25] **Longo G., Galasso G.**, An application of informational divergence to Huffman codes, *IEEE Trans. on Inf. Th.* **IT–28** (1982) 36–43.

[26] **Rubin F.,** Experiments in text file compression, *Comm. ACM* **19** (1976) 617–623.

[27] **Storer J.A.,** *Data Compression, Methods and Theory*, Computer Science Press, Rockville, Maryland (1988).

[28] **Vallarino O.**, On the use of bit-maps for multiple key retrieval, *SIGPLAN Notices, Special Issue* Vol. **II** (1976) 108–114.

[29] **Wagner R.A.,** Common phrases and minimum space text storage, *Comm. ACM* **16** (1973) 148–152.

[30] **Weiner P.,** Linear pattern matching algorithms, *Proc. 14-th IEEE Symp. on Switching and Automata Theory* (1973) 1–11.

[31] **Welch T.A.,** A technique for high-performance data compression, *IEEE Computer* **17** (June 1984) 8–19.

[32] **Ziv J., Lempel A.,** A universal algorithm for sequential data compression, *IEEE Trans. on Inf. Th.* **IT–23** (1977) 337–343.