A Systematic Appproach to Compressing a Full Text Retrieval System*

A. Bookstein†
S.T. Klein‡
D.A. Ziff†

Abstract

This paper reports on a variety of compression algorithms developed in the context of a project to put all the data files for a full-text retrieval system on a CD-Rom. In the context of inexpensive preprocessing, a text compression algorithm is presented that is based on Markov-modeled Huffman coding on an extended alphabet. Data structures are examined for facilitating random access into the compressed text. In addition, new algorithms are presented for compression of word indices, both the dictionaries (word lists), and the text pointers (concordances).

1 Introduction

In this paper we discuss the problems of compressing a large textual database, and the auxiliary files necessary for convenient access, for storage on a CD-Rom. Given the remarkable advances being made in computer mass storage technology, the growing interest in data compression, as evidenced in the recent spurt of literature in this area, may be surprising (see, for example,

*This paper is a revision of [5]

†Center for Information and Language Studies (CILS), University of Chicago, 1100 E. 57th St., Chicago, IL 60637

‡Department of Mathematics and Computer Science, Bar Ilan University, Ramat-Gan 52900, ISRAEL

the reviews [13] and [1]). Of course, part of this interest is associated with requirements unrelated to storage–for example the need to transmit large amounts of information over still expensive communication lines. But even within the area of data storage, compression is becoming increasingly important, ironically, driven by the advent of new storage capability. Data storage is one area where the maxim that supply drives demand is in evidence: the existence of new storage technologies, coupled with improved data capture technology, has greatly increased our appetite to put more data in machine readable form. Though interest includes pictorial and sound information, our concern has been with text and auxilliary files.

The problem of data storage often takes one of two forms:

- **The Huge Database Problem.** One may have a mammoth data base, and for operational efficiency, wish to distribute the most heavily used portion over the customer base in the form of a CD-Rom. If the distributed data were selected effectively, most use of the database would be satisfied locally by means of the CD-Rom; only misses would require more expensive interaction with the complete central store. Clearly, the greater the probability of a hit, the more attractive the CD-Rom would be. Here, designing procedures to identify the items that will be heavily used is critical[4], but even in conjunction with a very effective prediction algorithm, the performance will be enhanced if, because of data compression, a larger portion of the database could be fit onto the CD-Rom product.

- **The Large Database Problem.** A second motivation, closer to our own concerns, are situations where it is desirable to put a whole database on a CD-Rom, but where the database is somewhat too large to fit on a CD-Rom without compression[12]. Such is the case with the project described below. But also, we have found, the prospect of creating a database on CD-Rom generates desires that may not have been anticipated for accompanying the text with auxilliary files that improve processing efficiency or are convenient for the user. The more effective the compression algorithms, the more desired, if nonessential, files can we include.

In this paper we continue the discussion of [12], describing the current status of the project. In particular, we shall discuss in detail our text com-

pression technique, including some of the implementation problems that had to be overcome, and which are likely to recur in other realistic projects. The paper also describes our strategies for compressing some of the auxiliary files needed for effective retrieval. These extra data structures are critical for effective use of the database. They take up an amount of space comparable to the database itself, yet their storage requirements are often overlooked. Few other papers have examined these questions; in addition to our presentation at RIAO '91 [5], on which this paper is based, we note Witten, et. al. [14] and [12].

2 The ARTFL Project

The ARTFL project (American and French Research on the Treasury of the French Language), is likely to be a prototype of many future projects involving text distribution. We are dealing with a large corpus of text, here of interest largely to humanistic scholars. The need to compress the text is most apparent. But to access the text, auxiliary files such as dictionaries, concordances, bitmaps, etc., must be included; these are in practice very large and separate techniques must be developed to compress them as well.

ARTFL is a cooperative project between the University of Chicago and France's *Centre National de la Recherche Scientifique* (CNRS); its goal is to promote and facilitate research on the North American continent with the *Trésor de la Langue Française* database (TLF). The TLF database consists of about 680 megabytes of french language material, made up of a variety of complete documents including novels, short stories, poetry and essays, by a variety of authors. The bulk of the texts are from the 17th through 20th centuries, although smaller databases include texts from the 16th century and earlier. The database was created by the CNRS for the *Trésor de la Langue Française* dictionary project. The ARTFL project is the North American repository of the database. In conjunction with the TIRA (Textual Information Retrieval and Analysis) research group at the Center for Information and Language Studies at the University of Chicago, ARTFL has developed retrieval software to support a dial-up and network-oriented database service and has been providing access for researchers at institutions that subscribe to the ARTFL consortium. Recently, ARTFL has considered the feasibility of limited distribution of the database on CD-Rom to its subscribers. The

authors have studied the technical aspects of this problem. Data compression is the primary task, since our initial estimate of the size of the database plus auxilliary indexes was about 1 gigabyte, and our task was to fit this onto a CD-Rom that can store about 550 megabytes of data. If possible, this was to be done in a way that allowed additional information to be included as well.

Let us consider briefly the important technical details of the ARTFL database and the CD-Rom medium.

2.1 The ARTFL Database

2.1.1 The Text

The *text* consists of about 112,000,000 words. It is represented in ASCII, using only upper-case letters, punctuation, digits, and a small number of special characters. As will often be the case for a database of substantial size and history, the coding has qualities convenient for the initial intended uses of the database, but which are awkward when efficient storage considerations become paramount. For example, accented letters appear in the form `<`x`>`y, where $x \in \{$`'`, `` ` ``, `^`, `"`, `+`$\}$ and $y \in \{$`a,e,i,o,u,c`$\}$. Thus the phrase "*Trésor de la Langue Française*" would appear in the database as "`TR<'>ESOR DE LA LANGUE FRAN<+>CAISE`". This makes it easy to detect accented characters, but at the cost of considerably expanding the size of the database.

2.1.2 The Concordance

The *concordance* or *index* exists only for the 48,262,540 non-stop words, where a *stop-word* is defined to be one of the 100 most frequent words in the database. Every occurrence of every word in the database can be uniquely characterized by a sequence of numbers that gives its exact position in the text. In our case, 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 any occurrence can be represented hierarchically by the hexatuple (c, a, d, p, s, w), which we call a *coordinate*.

2.1.3 The Dictionaries

The dictionary is a lexicographically sorted list of the different words in the database. In order to allow the processing of truncated terms, in fact a permuted dictionary[6] is kept. Since, from a first estimate, we found that after compressing the text, the concordance, and the global dictionary, there would still be a substantial amount of space left on the CD, and since most user queries are author specific, we decided to store an additional set of dictionaries—at least, one for each author. Such dictionaries will be stored as bitmaps, indicating words from the global dictionary. Moreover, we shall also store statistical information about the distribution of the words relative to any given author or period.

2.1.4 Auxiliary Files

The coordinates in the concordance are given in logical hierarchical form. To retrieve an item, we need to know the physical location of its coordinate within the text on the CD-Rom. The translation from hierarchical coordinate to physical location is done by means of several small tables, as described below.

2.1.5 Bitmaps

These files are optional, but could enhance the system by speeding up the retrieval process. The possibilities are:

1. storing maps in addition to the concordance for reducing the number of I/O accesses [7];
2. having the maps replace a part of the concordance (or complement it, for example, for the stop-words) [3];
3. bitmaps as signature files [11].

2.2 The CD-Rom Medium

Technical details about CD-Roms can be found in [9] and [10]. The relevant details for our project are the following. Since a CD-Rom is physically identical to the familiar audio-CD's, it has become the practice to partition the

information stored on the disk into major units called *minutes* and *seconds*. Each second is further subdivided into 75 *blocks*, each block to 98 *frames*, and each frame to 24 bytes. Thus, a block holds 2352 bytes; however, part of it is dedicated to error correction codes, so we can store only 2048 bytes = 2K of information in a block, or 150K in a second and 9000K per minute. The standard CD contains around 60 minutes = 527.34 MB and the maximum possible is about 74 minutes, corresponding to 650.39 MB.

3 Compression of the text

The text is the largest and most complex component of our system. In our research, and here, most effort has gone into the text component. Our approach towards text compression is first to extend the alphabet in a useful way and then compress the modified text by means of an extension of Huffman encoding. Our first task, then, was to define the alphabet. Since the accented letters always appear in the form <*x*>*y*, and the characters < and > appear in no other context, all the strings that correspond to accented letters (there are about ten of them) were incorporated into the alphabet as new single letters, yielding immediate substantial savings. We then generated occurrence statistics about the distribution of character pairs, which led to the following observations.

3.1 Alphabet Definition

3.1.1 Preliminary Alphabet

Before beginning our formal analysis, we made some obviously required changes to produce a preliminary alphabet. Punctuation signs (periods, commas, etc.) were treated in the database in a similar way as words; thus they are generally not appended to the words they follow, but separated from them by white space (blank or newline). Thus, almost all the special signs, which are

```
!  "  ’  (  )  ,  -  .  :  ;  <i>
```

are virtually always surrounded by spaces. The one major exception is the apostrophe (’), which is followed, but not preceded, by a space.

There were however individual exceptions. For example, ! (exclamation) is followed by space 579042 times, and 3 times by another character; ’ (apostrophe) is followed by blank more than 7 million times, and about 1000 times by one of 30 other characters. These are almost certainly all errors. We discovered similar phenomena which were also almost certainly errors: individual characters which appeared fewer than 10 times in the entire database, and a few non-ASCII characters.

Since these odd deviants interfere with the efficient encoding of the text, we decided first to rewrite the database, eliminating several obvious errors. Further, though the algorithms used for extending the alphabet by including positively correlated strings can automatically take care of the fact that punctuation signs are (almost) always followed by blanks or newline (represented hereafter respectively by the underscore _ and by \n), we decided to immediately incorporate certain punctuation signs followed by space as primary elements in the alphabet before the systematic search for new characters to expedite processing. The general criterion for doing this is that if a common character is always or almost always followed or preceded by a second given character, the two should be encoded as a single unit. Technically, given a pair xy satisfying these criteria, the entropy, $H_{y|x}$ is very small. But we cannot (using Huffman coding) represent it by less than a single bit. If x occurs often, such a discrepancy incurs a significant cost and the pair is a candidate for immediate inclusion into our alphabet. We thus started with an alphabet consisting of the following 82 “characters”:

```
a b c d e f g h i j k l m n o p q r s t u v w x y z
 0 1 2 3 4 5 6 7 8 9 _ \n <*> <i>\n <i>_ <+>c ~p ~s
<">e <">i <’>e <’>u <^>a <^>e <^>i <^>o <^>u <‘>a <‘>e <‘>u
          ! !\n !_ " "\n "_ ’ ’_ ( (_ ) )\n )_
    ,  ,\n  ,_  -  -\n  -_  .  ..  .\n  ._  :  ;  ?
```

The total number of occurrences of the elements of our initial alphabet is 624,045,965, which could be stored in 595.4 MB using a standard ASCII encoding. Thus we have already saved over 80 megabytes of storage as compared to the original database.

3.1.2 Cooccurrence Analysis

The approach we chose for encoding the text is based on [2]. The text will be modelled as generated by a first order Markov process; thus, to encode it, we prepare one Huffman tree T_x for each character x; given that character x has just been scanned, the following character is encoded using the tree T_x. This squares the storage requirements for the decoding tables, but we will reduce the space complexity by clustering "similar" character distributions.

The next step then was to pass over the database and collect statistics about the co-occurrences of these elements. Ambiguities were resolved using a greedy method, i.e., at each point the algorithm tried to parse the longest possible string.

Co-occurrence information was used in two ways: ultimately it formed the basis for compression using the Markov model. But first, anticipating its intended use as part of a Markov model, co-occurrence data gave us the basis for planning further alphabet expansion: our strategy is to locate those strings whose occurrences deviate most from the Markov prediction and incorporate them into the alphabet. For example, if a trigram occurred substantially more often than the Markov model predicted, then we can improve compression by encoding the trigram trigram as a single character. This differs from some commonly used methods that rely on frequency alone. For example, it would not be helpful to encode a frequently occurring bigram as a single character if the frequency of occurrence is predictable from the model itself.

The impact of using a Markov model, even before alphabet expansion was significant. Applying simple Huffman coding yields an average codeword length of 4.48434 bits, or 333.60 MB for the entire text. Using the Huffman code derived from bigram distribution, and assuming a first order Markov process (see below), one gets a total size of 261.68 MB. We see that the Markov model gives a substantial improvement over the straightforward Huffman code. To further improve the compression, we extend the alphabet as indicated above.

3.1.3 Identifying Deviant Strings

Denote by f_S the frequency of occurrence of the string S, where S can be one or more characters long, and by $P_{y|x}$ the conditional probability of the

character y, given that we have just scanned the character x. We thus have $P_{y|x} = f_{xy}/f_x$; similarly, $P_{yz|x}$, which denotes the probability of yz given x, is f_{xyz}/f_x. As an estimate for the length of the codeword assigned to an element with probability of occurrence p we use $-\log p$ (all logarithms are to base 2), which is the information theoretic lower bound; this bound is achieved for arithmetic coding and usually approached quite well by Huffman coding.

The question now is, is it reasonable to combine two characters, say a and b, into a single character ab. Because of the Markov assumption, we must consider this question separately for each preceding character x. If we don't combine a and b, then after x the storage requirement for ab, under the Markov assumption, is about $-f_{xab}\log(P_{a|x}P_{b|a})$. On the other hand, if ab is treated as a unit, we expect a storage cost after x of about $-f_{xab}\log P_{ab|x}$. Summing over x we get

$$\begin{aligned} E_m([a][b]) &\equiv -\sum_x f_{xab}\log(P_{a|x}P_{b|a}) \\ E_m([ab]) &\equiv -\sum_x f_{xab}\log(P_{ab|x}) + \varepsilon(a,b) \end{aligned}$$

The function ε expresses the fact that including the string ab affects not only the probabilities of mainly the three character a, b and ab, there is also in fact an influence, though generally a minor one, on the probabilities of all the other characters as well. To facilitate the computations, we shall however assume that $\varepsilon(a,b)$ is negligible relative to the other terms, so that our heuristic measure for including the string ab will be

$$D_M(a,b) \;=\; E_m([a][b]) \;-\; E_m([ab]) \;\simeq\; \sum_x f_{xab}\;\log\frac{P_{ab|x}}{P_{a|x}\;P_{b|a}}.$$

Thus we should combine ab into a single character if $D_M(a,b) \gg 0$. We first note that this quantity is always non-negative. To see this, rewrite $f_{xab} = NP_{xab}$ as $NP_{ab}\frac{P_{xab}}{P_{ab}}$; then the sum, after expanding the conditional probabilities, is

$$N\;P_{ab}\;\sum_x \frac{P_{xab}}{P_{ab}}\;\log\;\frac{\frac{P_{xab}}{P_x}}{\frac{P_{xa}}{P_x}\;\frac{P_{ab}}{P_a}} \;=\; N\;P_{ab}\;\sum_x \frac{P_{xab}}{P_{ab}}\;\log\;\frac{\frac{P_{xab}}{P_{ab}}}{\frac{P_{xa}}{P_a}}.$$

But, since $\sum_x \frac{P_{xab}}{P_{ab}} = 1$ and $\sum_x \frac{P_{xa}}{P_a} = 1$, both $\left\{\frac{P_{xab}}{P_{ab}}\right\}$ and $\left\{\frac{P_{xa}}{P_a}\right\}$ are probability distributions, and the sum is of the form $C\sum_i P_i\log(P_i/Q_i)$, with $\{P_i\}$ and

$\{Q_i\}$ probability distributions; this sum is well known to be non-negative [2]. Finally, representing the probabfilities in terms of frequencies, we derive the more useful formula

$$D_M(a,b) \simeq \sum_x f_{xab} \log \frac{f_a \; f_{xab}}{f_{xa} \; f_{ab}}. \tag{1}$$

In this form, D can be computed from frequency tables, which are more easily manipulated than the text itself. The best bigrams to be included as new elements in the extended alphabet are those with largest $D_M(a,b)$.

3.1.4 Algorithm for Alphabet Construction

The simplest form of the general algorithm, which could be applied to an arbitrary database, is therefore:

1. collect frequency statistics; use these to catch errors and to flag obvious pairs to include in our base alphabet;
2. collect statistics on the character, bigram and trigram distribution of the base alphabet;
3. for each triplet (x,a,b) compute $D_M(x,a,b) = f_{xab} \log \frac{f_a \; f_{xab}}{f_{xa} \; f_{ab}}$;
4. for triplets for which $D_M(x,a,b)$ is greatest:
   (a) sort by (a,b), getting the set $\{(x,a,b) : \forall x\}$ as adjacent lines;
   (b) compute for each pair (a,b): $D_M(a,b) = \sum_x D_M(x,a,b)$;
   (c) sort by decreasing $D_M(a,b)$;
   (d) select the n bi-grams.

In the TLF, the elements with largest $D_M(a,b)$ from this list are: `e_ un s_ ue et en r_ ne t_ il ll l_ a_ ur re i_ se`. While these are familiar strings in French, they are hardly the most typical of the language. Note for example the absence of `qu` which appears only in rank 777 in this list. The reason is that the Markov model itself takes care of most of the known dependencies between characters (`q` is almost always followed by `u`), so there is no need to incorporate such strings; recall that our formula aims at detecting those strings which most strongly *deviate* from the Markov model.

Resisting the natural impulse to combine frequent bigrams such as `qu` is one example of the benefits of using a model based approach.

Our next step was to improve the selection of pairs for the extended alphabet. In our elaboration, we try to be careful of possible overlaps. Since the parsing algorithm works on a greedy basis, there is some danger of creating new elements that are never used. For instance, suppose both `th` and `he` are elements in the extended alphabet. The decision to include `he` as an element was based on its frequency of appearance in the original text, to which the word `the` strongly contributed. However, the word `the` will always be parsed as `th-e` rather than `t-he`, leaving `he` unused. There are many ways to take into account this and other higher order overlaps. We decided on an iterative procedure, of which we present a preliminary outline here.

At the first stage, we adjoin only non-overlapping pairs to our alphabet. More precisely, we scan the sequence of pairs (a, b) by decreasing $D_M(a, b)$, and decide to adjoin a pair (a, b) to the alphabet if and only if a did not appear in a pair that was already selected as (x, a), and b did not yet appear in a pair as (b, y). Strongly correlated pairs which are skipped now have another chance to be included in the next iteration.

We decided at the first stage to extend the character set to about half again its initial size. The text was scanned again and parsed using the extended alphabet to produce a new set of frequency tables. If we stoped at this stage and used Markov model based Huffman coding on the new bigram distribution, one gets a total size of 232.34 MB.

The above step was now repeated, using (1) to produce the most promising pairs of new elements. Note that such a "meta-pair" may consist of a string of 2, 3 or 4 characters of our original alphabet. The new filtering is more complicated, because the overlap of the pairs may now be of 1 or 2 characters. The new procedure is thus to scan the list of possible meta-pairs in order of their expected savings as given by (1). In order for the pair (A, B) to be included, the following conditions regarding the pairs *already* chosen during this iteration must be satisfied:

1. that no pair (x, A) was already selected;
2. if A is a single character, that there was no pair (x, yA); if A is a pair Cz, that there was no pair (x, C);
3. that there was no pair (B, y);

4. if B is a single character, that there was no pair (By, z); if B is a pair zD, that there was no pair (D, y).

At the second stage, we decided again to extend the alphabet by about half. Another scan of the text showed that using Huffman coding on the n-grams reduces the size of the text to 211.58 MB. Although we stopped here, the iteration could be continued, increasing the alphabet further. This could be compensated for by testing whether previously formed pairs could be removed because of subsequent changes in the statistics.

For comparison, we also produced new elements assuming that the text was created through independent character generation. Since this method is simpler, it should be considered if the more complex procedure does not produce a substantial benefit. Let N denote the total number of characters, and p_S the (unconditional) probability of occurrence of the string S. In order to decide if a bigram ab should be included as a new element in the extended alphabet, this time we compare the expected number of bits contributed by the pair ab if it will be kept together, $E_I([ab])$, with the expected number of bits if the characters appeared separately, $E_I([a][b])$, under the assumption of independent character generation. As in the previous derivation,

$$\begin{aligned} E_I([a][b]) &= -f_{ab}(\log p_a + \log p_b) \\ E_I([ab]) &= -f_{ab} \log p_{ab} + \varepsilon(a, b) \end{aligned}$$

Ignoring $\varepsilon(a, b)$ again, we get

$$D_I(a, b) \;=\; E_I([a][b]) \;-\; E_I([ab]) \;\simeq\; f_{ab} \;\log \frac{N \;f_{ab}}{f_a \;f_b}. \tag{2}$$

A bigram ab is worth being included in the extended alphabet if $D(a, b) \gg 0$. This suggests the following procedure:

1. compute the weight $D_I(a, b)$ for each pair of characters (a, b);

2. sort the list by decreasing $D_i(a, b)$.

Note that contrary to $D_M(a, b)$, $D_i(a, b)$ might be negative, that is, extending our alphabet by introducing the wrong pairs might in fact lead to performance deterioration.

The results of applying this procedure on `alphabet-0` were quite illuminating. The strongest candidates for being included were (in decreasing order): `e_ qu s_ _, _d '_ ou on ,_ de t_ nt le es _p ai _. en _l re ch`. These are clearly some of the most strongly correlated bigrams in French and could therefore have been expected. We also see that the list is quite different from the one generated with (1). The surprise was however at the other end of the list. The worst bigrams are: `_e _s a_ i_ _t _i _n u_ el _r _u _o rs su nn na tt ni`. All these strings have the following in common: their constituents are frequent characters and they appear frequently as a pair. The reason for their place at the end of the list is that they are all negatively correlated: that is, although they occur often, they occur less than expected given the frequencies of their component characters. Since the frequency f_{ab} is a factor of $D_I(a, b)$, a very frequent, slightly negatively correlated pair may be worse than a less frequent, but strongly negatively correlated one. This shows that the simple minded method of including the most frequent pairs might be a bad idea. For example the pair `_e` appears more than 7 million times (rank 12 in the list of bigrams sorted by frequency), and is still the worst possible choice.

Comparing this list, which assumes independent character generation, with the one obtained from assuming a Markov model, we see that both start with `e_`, however the next element is already different: `qu` for the former and `un` for the latter. We thus devised the following test. Using only the collected works of *Emile Zola* as test database, we extended the base alphabet with `e_` and `qu` in one case, and with `e_` and `un` in the other. The results were of course very close, since only a single character was substituted, but was consistent with our theory: the text would be stored in 6.980 MB in the first case, 6.977 MB in the second (Markov model) case.

Extending the alphabet (using D_I), to the same size as in the Markov model case and taking care not to include overlapping pairs, we got a total size of 234.54 MB for the entire database; that is more than 2 MB (or about 1%) more than with the extension of the alphabet generated by (1). In a second iteration, the alphabet was extended to the size of the corresponding iteration of the Markov-based method, yielding a total size of 215.18 MB, that is 3.6 MB more than for the alphabet of identical size based on the Markov model. Thus we do gain a modest improvement using the Markov based criterion — for some projects, the extra savings may not justify the additional complexity of collecting the statistics.

We did not intend to extend the alphabet further, but we applied formula (2) again to produce new meta-pairs, which could consist of 2 to 8 original characters from our initial alphabet. Here are the strongest pairs: `la_ il_ qui_ les_ est_ dans_ elle_ un_ des_ pour vous_ ne_ our,_ n'_ pas_ eur_ une_ ement_ comm ant_ ,_et_ tion_ con pl lus_ re ass ati c'_es`. This list is interesting in itself since all its members are easily recognized as french word fragments, though the list has been compiled by purely statistical methods. This shows that our method is in fact language independent. Note also the bigram `re` which appeared already towards the beginning of the first list, but has not been included in the alphabet extension because of overlaps.

The next step is now clustering, which is described in detail in [2] (Section 5.3).

4 Compression of the concordance

Concordance compression has received some attention [8], but in our opinion, no approach has so far placed itself within the mainstream of compression theory. We present here an outline of a new method which we intend to study and compare with other methods.

We wish to base our coding of a particular concordance entry on computed probabilities of the entry taking any given value. Thus, we first compute probabilities for possible next entries, then create a Huffman tree for coding/decoding the entry. Thus the code varies over the concordance, adapting itself to the changing probabilities. We want the probabilities to take advantage of all our most useful knowledge.

Thus there are two problems: how to compute the probabilities, and what tree structure to use. For simplicity, suppose that coordinates are of the form (document, part, sentence).

4.1 Probabilistic Model

We model concordance compression and decompression as a sequential process, and we suppose that at any stage we know:

- how many occurrences remain of word;

- how much space is left in the database itself.

The probabilities can be estimated as follows (using independence assumption): let the database be divided into units. Suppose there are x units left in the document at the point we are compressing/decompressing the next coordinate; and that there are y units left in the database. Further, there are z more occurrences of the word. Then let $\lambda = zx/y$ be the expected number of occurrences of the word in the rest of the document. If so, the probability of the word not occurring (again) in the document is $e^{-\lambda}$, while $1 - e^{-\lambda}$ is the probability that the word does occur again (the condition for prefix ommission).

Similarly, we can compute the probability under this model of a skip of 1, 2, 3 or more documents before the word occurs again.

4.2 Tree Structure

These probabilities can then be used to construct a Huffman tree for the document component of the coordinate, taking into account the prefix omission case and the possibility of various skips. If the word does occur again in the same document, then we can compute the probabilities of prefix omission at the part and sentence level, etc., exactly as before. The Huffman code would express these probabilities.

4.3 Practical point

In constructing the Huffman tree, we at various points have to encode a number (e.g. how many documents are skipped before the next occurrence). Instead of encoding each possibility, we should encode ranges. Thus we should create probabilities for skipping m to n documents, where, for efficiency, representing the values m to n requires an integral number of bits, and is as close as possible to .5. (Note: in this case, we encode not $m, m+1, \ldots, n$ but $0, 1, 2, \ldots, n-m$.)

5 Compression of the Dictionaries

The design and storage of the permuted dictionary is explained in [6]. We here comment only on the additional dictionaries we wish to keep for each

author. Let n be the number of different words in the global dictionary. For each author i, we prepare a bitmap B_i of length n, the j-th bit position referring to the j'th word of the global list; the bit in position j of B_i will be 1 if and only if word number j of the global list appears also in the wordlist of author i.

The bit-vector B_i only shows whether a given word is used by the author, but we would also like to have information about the frequencies of word occurrences within the sub-databases corresponding to each author. It is however too costly to store such frequency lists by the straightforward manner of representing each word position by a couple of bytes indicating its frequency of occurrence. The problem is that, whereas most words occur infrequently, we would need enough space for each word to represent the most frequently occurring word. Instead, we store this information in a hierarchical way by a sequence of tables, starting with a table with a large number of entries but where each entry is short, up to a short table having relatively large entries.

More specifically, for author i, we consider the bitmap itself to be table F_0^i, where *zero* indicates no occurrence and *one* indicates *at least one* occurrence. Then, we build a table F_1^i, for which the number of entries equals the number of 1-bits in B_i, that is, one entry for each different word used by author i. Each entry in F_1^i is of 2 bits. The 4 possible patterns encode the numbers 1, 2, 3 or "at least four". We again use here the fact that most of the words occur only rarely in a given context. We then have a table F_2^i, with one entry for each element in F_1^i which contained "more". The entries in F_2^i are of size 1 byte, giving us the possibility to encode frequencies from 4 to 258, in addition to one codeword indicating that the frequency is more than 258. Finally, a table F_3^i, with one entry for each element in F_2^i showing more than 258 occurrences, has 2 or 3 bytes per entry.

If a query is submitted which restricts the database to author i, The bitmap B_i is used to filter out the relevant words from the global dictionary. By ANDing and ORing of these maps, one can easily respond to complex constraints on the author set, such as words which are used by one author but not by a certain other author, etc. The encoding and decoding procedures for the frequency tables should be obvious.

6 Translating hierarchical to physical coordinates

Recall that a coordinate has the form (c, a, d, p, s, w). For a given coordinate, we must find its physical location on the disk. This can be accomplished by having a complete translation table for each coordinate component (for example, a table for all authors; another for documents of an author, etc.). However, this would be equivalent to keeping an additional copy of the concordance, and is therefore ruled out on grounds of cost. In our application, we split the access information between RAM and the disk itself.

The compressed text will be partitioned into blocks of equal size which correspond to units which can be read in a single read-operation, say of size 4K. As we have less than 250 MB, we get about 64,000 blocks, which can be indexed by 2 bytes. We lose about half a codeword at the end of each block by aligning codewords with the beginning of each block.

The text is accessed via a table S which has one entry for each of the sentences in the database. Entry number i in this sequence gives the length of (the uncompressed) sentence i in bytes (not in words). The great majority (all but 3%) of the sentences are shorter than 256 bytes. Thus most entries in S can be just one byte long, and the longer sentences are handled by having 2 (or 3) consecutive bytes in S for them, the first (or first and second) of which is 0. Immediately preceding the entry in S corresponding to the first sentence of a part, we store the index of the block in which this part starts (2 bytes, referred to as I below), and the offset (OF) in bytes from the beginning of the decompressed block (2 bytes). In addition, we have another table B, with one entry for each block, $B(i)$ giving the size in bytes of the decompressed block i. There are about 5 million sentences, so that the lengths will take: for S, about 5.2 MB, then 4 additional bytes for each of the 36,000 parts (about 144K); B requires 2 bytes for each entry (about 128K). Together, less than 5.5 MB are required.

The tables stored in RAM will give us direct access into table S on the disk. We have a table P with one entry for each part, $P(i)$ pointing to the beginning of part i in S. P is accessed through a table D, which has one entry for each document (about 2000), $D(i)$ being the index of the first entry in P which belongs to document i. D itself is accessed from A, a table having one entry for each author (about 400), $A(j)$ being the index

of the first entry in D which belongs to author j. For a given coordinate $X = (c, a, d, p, s, w)$, the pointer to the part in S in which X can be found is therefore $P(p + D(d + A(a)))$. In order to save some space, there is another table DS, giving for each of the 2000 documents, the pointer to its beginning in S, so that table P does not contain absolute addresses, but rather relative offsets within the subtable of S corresponding to a document. Thus the entry in S corresponding to X is given by

$$P(p + D(d + A(a))) \quad + \quad DS(d + A(a)). \qquad (*)$$

Space in RAM: each entry in P uses 2 bytes; in DS, 3 bytes; and in D and A, 2 bytes. The total is roughly $2 \times 36000 + (3 + 2) \times 2000 + 2 \times 400 = 83K$.

Access procedure: given $X = (c, a, d, p, s, w)$, compute (*) using a, d and p and access the disk at the specified location in S. As noted above, the first two entries at this location in S are I (the block index in which the part begins) and OF, the offset within the block. Having set these, read the next s entries and add them to OF (taking care of reading additional bytes if a zero-byte is encountered); this gives us the offset of the sentence. Now compare this quantity with $B(I)$. If $OF \leq B(I)$, then this sentence indeed starts in block number I: access, decompress, and jump to OF. If $OF > B(I)$, that means that though part p starts in block I, sentence s of part p is in one of the subsequent blocks. By comparing OF with $B(I) + B(I + 1) + \cdots$, we can find the index of the required block.

The process of adding the sentence lengths can be sped up by adding one or more layers of pointers and addresses of sentences (for example, every 20-th, 400-th, etc. sentence). The number of additional bytes would be low, and now long parts are scanned much more quickly, skipping over large chunks of sentences in a single operation.

An important parameter is the access time, which is slow for a CD-Rom. We have one seek into table S, then some local processing, then one access to table B and one to the text itself. It might be worthwhile having the table B also in RAM. In fact, when a sequence of coordinates is processed, the access to S is mostly only a virtual one, since an entire page will already be in core.

References

[1] T. Bell, I.H. Witten, J.G. Cleary, "Modeling for Text Compression", *ACM Computing Surveys*, 21 (1989) 557–592.

[2] A. Bookstein, S.T. Klein, "Compression, Information Theory and Grammars: A Unified Approach", *ACM Transactions on Information Systems*, 8 (1990) 27–49.

[3] A. Bookstein, S.T. Klein, "Using Bitmaps for Medium Sized Information Retrieval Systems", to appear in *Information Processing & Management*, 26 (1990).

[4] A. Bookstein, J. Handley, "Comparison of Analytic Models and CHRT for Database Subsettings", OCLC Working Paper, Office of Research, OCLC, 1990.

[5] A. Bookstein, S.T. Klein, D.A. Ziff, "The ARTFL Data Compression Project", *Proc. RIAO Conf. 1991,* Barcelona (1991) 967–985.

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

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

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

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

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

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

[12] S.T. Klein, A. Bookstein, S. Deerwester, "Storing Text Retrieval Systems on CD-Rom: Compression and Encryption Considerations", *ACM Transactions on Information Systems*, 7(3) (1989) 230-245.

[13] D.A. Lelewer, D.S. Hirschberg, "Data Compression", *ACM Computing Surveys*, 19 (1987) 261–296.

[14] I.H. Witten, T.C. Bell, C.G. Nevill, "Models for Compression in Full-Text Retrieval Systems", *Proc. Data Compression Conference 1991*, Los Alamitos, CA: IEEE Computer Society Press (1991), 23-32.