Improved Hierarchical Bit-Vector Compression in Document Retrieval Systems

Y. Choueka[1,2], **A.S. Fraenkel**[3], **S.T. Klein**[3], **E. Segal**[1]

[1] Inst. for Information Retrieval and Computational Linguistics (IRCOL) — The Responsa Project
[2] Department of Mathematics and Computer Science, Bar-Ilan University, Ramat Gan, Israel
[3] Department of Applied Mathematics, The Weizmann Institute of Science, Rehovot 76100, Israel

Part of this work was done while the second and third authors were partially affiliated with IRCOL

April 1986

ABSTRACT

The "concordance" of an information retrieval system can often be stored in form of bit-maps, which are usually very sparse and should be compressed. Hierarchical bit-vector compression consists of partitioning a vector v_i into equi-sized blocks, constructing a new bit-vector v_{i+1} which points to the non-zero blocks in v_i, dropping the zero-blocks of v_i, and repeating the process for v_{i+1}. We refine the method by pruning some of the tree branches if they ultimately point to very few documents; these document numbers are then added to an appended list which is compressed by the prefix-omission technique. The new method was thoroughly tested on the bit-maps of the Responsa Retrieval Project, and gave a relative improvement of about 40% over the conventional hierarchical compression method.

1. Motivation and Introduction

In some full-text retrieval systems, an inverted file, also called "concordance", is constructed that contains, for every different word W in the data base, a list $L(W)$ of all the documents in which W occurs. In order to find all the documents that contain the words A and B, one has to intersect $L(A)$ with $L(B)$. Usually, A and B stand for families of words A_i and B_j, each family consisting of terms which are considered synonymous for the given query. In this case one has to perform

$$\Big(\bigcup_i L(A_i)\Big) \bigcap \Big(\bigcup_j L(B_j)\Big)$$

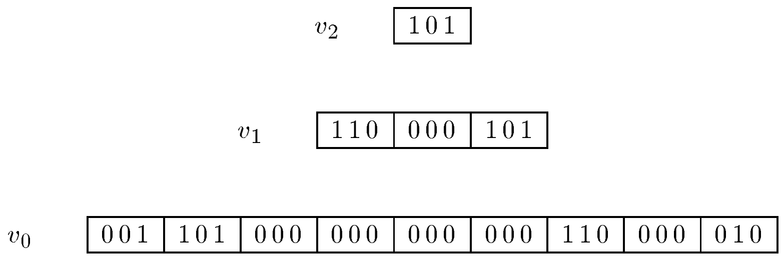
which can seriously affect the response time in an online retrieval system when the number of involved sets and their sizes are large.

A different approach might be to replace the concordance of a system with l documents by a set of *bit-maps* of equal length l. For every different word W in the data base a bit-map $B(W)$ is constructed, the i-th bit of which is 1 if and only if W occurs in the i-th document, in some fixed ordering of the documents. Processing queries reduces here to performing logical operations with bit-strings, which is easily done on most machines. Davis & Lin [2] were apparently the first to propose the use of bit-maps for secondary key retrieval.

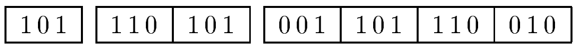
It would be wasteful to store the bit-maps in their original form, since they are usually very sparse (the great majority of the words occur in very few documents). In addition to the savings in secondary storage space, the processing time can be improved by compressing the maps, as more information can be read in a single input operation, thus reducing the total number of I/O accesses. We are interested in a coding procedure which reduces the space needed to store long sparse binary strings without losing any information; in fact, there should be a simple algorithm which, given the compressed string, can reconstruct the original one. Schuegraf [9] proposes to use *run-length coding* for the compression of sparse bit-vectors, in which a string of consecutive zeros terminated by a one (called a *run*) is replaced by the length of the run. A sophisticated run-length coding technique can be found in Teuhola [10], and various other variants are discussed in Nevalainen, Jakobsson & Berg [8]. Jakobsson [5] suggests to partition each vector into k-bit blocks, and to apply Huffman coding on the 2^k possible bit-patterns. In Fraenkel & Klein [4], the latter method is extended, incorporating run-length coding of blocks consisting only of zeros.

In this paper we concentrate on *hierarchical bit-vector compression*: let us partition the original bit-vector v_0 of length l_0 bits into k_0 equal blocks of r_0 bits, $r_0 \cdot k_0 = l_0$, and drop the blocks consisting only of zeros. The resulting sequence of non-zero blocks does not allow the reconstruction of v_0, unless we add a list of the indices of these blocks in the original vector. This list of up to k_0 indices is kept as a binary vector v_1 of $l_1 = k_0$ bits, where there is a 1 in position i if and only if the i-th block of v_0 is not all zero. Now v_1 can further be compressed by the same method.

In other words, a sequence of bit-vectors v_j is constructed, each bit in v_j being the result of ORing the bits in the corresponding block in v_{j-1}. The procedure is repeated recursively until a level t is reached where the vector length reduces to a few bytes, which will form a single block. The compressed form of v_0 is then obtained by concatenating all the nonzero blocks of the various v_i, while retaining the block-level information. Decompression is obtained simply by reversing these operations and their order. We start at level t, and pass from one level to the next by inserting blocks of zeros into level $j-1$ for every 0-bit in level j.



(a) Original vector and two derived levels



(b) Compressed vector

Figure 1: *Hierarchical Bit-vector Compression*

Figure 1 depicts an example of a small vector v_0 of 27 bits and its derived levels v_1 and v_2, with $r_i = 3$ for $i = 0, 1, 2$ and $t = 2$. The sizes r_j of the blocks are parameters and can change from level to level for a given vector, and even from one word of the database to another, although the latter is not practical for our applications. Because of the structure of the compressed vector, we call this the TREE-method, and shall use in our discussion the usual tree-vocabulary: the *root* of the tree is the single block on the top level, and for a block x in v_{j+1} which is obtained by ORing the blocks $y_1, \ldots, y_{r_j}$ of v_j, we say that x is the *father* of the non-zero blocks among the y_i.

The TREE-method was proposed by Wedekind & Härder [12]. It appears also in Vallarino [11], who used it for two-dimensional bit-maps, but only with one level of compression. In Jakobsson [6], the parameters (block size and height of the tree) are chosen assuming that the bit-vectors are generated by a memoryless information source, i.e., each bit in v_0 has a constant probability p_0 for being 1, independently from each other. However, for bit-maps in information retrieval systems, this assumption is not very realistic a priori, as adjacent bits often represent documents written by the same author; there is a positive correlation for a word to appear in consecutive documents, because of the specific style of the author or simply because such documents often treat the same or related subjects. In our approach, the parameters are first restricted so as to simplify the computer processing of r_j-bit blocks. From among this restricted set P of parameters, we select those which yield maximal compression.

The experiments were run on the bit-maps which were constructed at the Responsa Retrieval Project (see for example Choueka [1] or Fraenkel [3]) of about 40 million Hebrew and Aramaic words. They showed, a posteriori, that the compression does not vary much with the different elements of P. Thus any choice of parameters from P will do, so the method is efficient also for dynamically changing data bases.

In the next section we suggest some improvements to the TREE-method. We then report in Section 3 on the experiments which led to the parameter setting. Comparison of the results with those obtained on randomly generated bit-vectors shows that the technique is specially well adapted to bit-maps of document retrieval systems. The new algorithm is compared with other methods in the final Section 4.

2. Improvements to method TREE

We first remark that the hierarchical method does not always yield real compression. Consider for example a vector v_0 for which the indices of the 1-bits are of the form ir_0 for $i \le l_0/r_0$. Then there are no zero-blocks (of size r_0) in v_0, moreover all the bits of v_i for $i > 0$ will be 1, so that the whole tree must be kept. Therefore the method should be used only for sparse vectors.

In the other extreme case, when v_0 is very sparse, the TREE-method may again be wasteful: let $d = \lceil \log_2 l_0 \rceil$, so that a d-bit number suffices to identify any bit-position in v_0. If the vector is extremely sparse, we could simply list the positions of all the 1-bits, using d bits for each. This is in fact the inverse of the transformation performed by the bit-vectors: basically, for every different word W of the database, there is one entry in the inverted file containing the list of references of W, and this list is transformed into a bit-map; here we change the bit-map back into its original form of a list.

A small example will illustrate how the bijection of the previous paragraph between lists and bit-maps can be used to improve method TREE. Suppose that among the $r_0 \cdot r_1 \cdot r_2$ first bits of v_0 only position j contains a one. The first bit in level 3, which corresponds to the ORing of these bits, will thus be set to 1 and will point to a sub-tree consisting of three blocks, one on each of the lower levels. Hence in this case a single 1-bit caused the addition of at least $r_0 + r_1 + r_2$ bits to the compressed map, since if it were zero, the whole sub-tree would have been omitted. We conclude that if $r_0 + r_1 + r_2 \ge d$, it is preferable to consider position j as containing zero, thus omitting the bits of the sub-tree, and to add the number j

to an appended list L, using only d bits. This example is readily generalized so as to obtain an optimal partition between tree and list for every given vector, as will now be shown.

We define l_j and k_j respectively as the number of bits and the number of blocks in v_j, for $0 \le j \le t$. Note that $r_j \cdot k_j = l_j$. Denote by $T(i,j)$ the sub-tree rooted at the i-th block of v_j, with $0 \le j \le t$ and $1 \le i \le k_j$. Let $S(i,j)$ be the size in bits of the compressed form of the sub-tree $T(i,j)$, i.e., the total number of bits in all the non-zero blocks in $T(i,j)$, and let $N(i,j)$ be the number of 1-bits in the part of the original vector v_0 which belongs to $T(i,j)$.

During the bottom-up construction of the tree these quantities are recursively evaluated for $0 \le j \le t$ and $1 \le i \le k_j$ by:

$$N(i,j) = \begin{cases} \text{number of 1-bits in block } i \text{ of } v_0 & \text{if } j = 0, \\ \sum_{h=1}^{r_j} N\big((i-1)r_j + h, j-1\big) & \text{if } j > 0; \end{cases}$$

$$S(i,j) = \begin{cases} 0 & \text{if } j = 0 \text{ and } T(i,0) \text{ contains only zeros}, \\ r_0 & \text{if } j = 0 \text{ and } T(i,0) \text{ contains a 1-bit}, \\ \sum_{h=1}^{r_j} S\big((i-1)r_j + h, j-1\big) & \text{if } j > 0. \end{cases}$$

At each step, we check the condition

$$d \cdot N(i,j) \le S(i,j). \tag{1}$$

If it holds, we prune the tree at the root of $T(i,j)$, adding the indices of the $N(i,j)$ 1-bits to the list L, and setting then $N(i,j)$ and $S(i,j)$ to zero. Hence the algorithm partitions the set of 1-bits into two disjoint subsets: those which are compressed by the TREE-method and those kept as a list. In particular, if the pruning action takes place at the only block of the top level, there will be no tree at all.

Note that in case of equality in (1), we execute a pruning action although a priori there is no gain. However, since the number of 1-bits in v_j is thereby reduced, this may enable further prunings in higher levels, which otherwise might not have been done.

We now further compress the list L (of indices of 1-bits which were "pruned" from the tree) using the *prefix-omission* technique. It consists of storing a common prefix of several consecutive words only once, and is usually applied to the compression of dictionaries. This can be adapted to the compression of a list of d-bit numbers: we choose an integer $c < d-1$ as parameter, and form a bit-map v of $k = \lceil l_0/2^c \rceil$ bits, where bit i, for $0 \le i < k$, is set to 1 if and only if the integer i occurs in the $d-c$ leftmost bits of at least one number in L. Thus a 1-bit in position i of v indicates that there are one or more numbers in L in the range $[i2^c, (i+1)2^c - 1]$. For each 1-bit in v, the numbers of the corresponding range can now be stored as relative indices in that range, using only c bits for each, and an additional bit per index serving as flag, which identifies the last index of each range. Further compression of the list L is thus worthwhile only if

$$d \cdot |L| \ > \ k + (c+1)|L|. \tag{2}$$

The left hand side of (2) corresponds to the number of bits needed to keep the list L uncompressed. Therefore this secondary compression is justified only when the number of elements in L exceeds $k/(d-c-1)$.

For example, for $l_0 = 128$ and $c = 5$, there are 4 blocks of 2^5 bits each; suppose the numbers in L are 36, 50, 62, 105 and 116 (at least five elements are necessary to justify further compression). Then there are three elements in the second block, with relative indices 4, 18 and 30, and there are two elements in the fourth block, with relative indices 9 and 20, the two other blocks being empty. Thus the following information would be kept:

$$v \qquad \boxed{0\ 1\ 0\ 1}$$

$$\overbrace{\boxed{0-00100}\ \boxed{0-10010}\ \boxed{1-11110}} \qquad \overbrace{\boxed{0-01001}\ \boxed{1-10100}}$$

indicates the end of the sequence

Finally we get even better compression by adapting the cut-off condition (1) dynamically to the number of elements in L. During the construction of the tree, we keep track of this number and as soon as it exceeds $k/(d-c-1)$, i.e., it is worthwhile to further compress the list, we can relax the condition in (1) to

$$(c+1) \cdot N(i,j) \leq S(i,j), \tag{3}$$

since any index which will be added to L, will use only $c+1$ bits for its encoding.

In fact, after recognizing that L will be compressed, we should check again the blocks already handled, since a sub-tree $T(i,j)$ may satisfy (3) without satisfying (1). Nevertheless, we have preferred to keep the simplicity of the algorithm and not to check again previously handled blocks, even at the price of losing some of the compression efficiency. Often, there will be no such loss, since if we are at the top level when $|L|$ becomes large enough to satisfy (2), this means that the vector v_0 will be kept in its entirety as a list. If we are not at the top level, say at the root of $T(i,j)$ for $j < t$, then all the previously handled trees will be reconsidered as part of larger trees, which are rooted on the next higher level. Hence it is possible that the sub-tree $T(i,j)$, which satisfies (3) but not (1) (and thus was not pruned at level j), will be removed as part of a larger sub-tree rooted at level $j+1$.

The method of dynamically pruning the tree, forming a list L and compressing the latter using prefix omission, will henceforth be called the PRUNE-method.

3. Experimental Results

3.1 Setting the parameters

The above method was tested on the bit-maps of the Responsa data base of $l_0 = 42272$ documents; accordingly, $d = 16$. To allow for easy computer manipulation, we decided to use only blocks of one, two or four bytes, so that the possible variants had to verify

$$\log_2 r_i \in \{3, 4, 5\} \qquad \text{and} \qquad \sum_{i=0}^{t} \log_2 r_i = 16.$$

There are 24 such variants, which are listed below in Table 1. Each variant is characterized by a $(t+1)$-tuple $(a_0, \ldots, a_t)$, with $t = 3$ or $t = 4$, such that $a_i = r_i/8$ is the number of bytes in the blocks on level i, for $0 \leq i \leq t$.

Table 1: *Block sizes corresponding to various possibilities of hierarchical compression*

11112	1144	1441	2222	4141
11121	1224	2124	2421	4212
11211	1242	2142	2421	4221
12111	1414	2214	4114	4411
21111	1422	2241	4122	

For secondary compression of the list, the parameter c was set to 7, so that one byte was used for any number added into list L, instead of two bytes before L had been compressed. The length k of the bit-vector v used to compress L was therefore $\lceil 42272/128 \rceil = 331$, but we chose a 50-byte block ($k = 400$) as was done at the Responsa Project to facilitate updating in case of growth of the data base in the future. Therefore the condition for further compressing the list L was $|L| > 50$, and the number of bytes needed to store the list L was $\min(2|L|, 50 + |L|)$.

In order to compare the 24 block-size patterns, a test was run on a sample of 1040 bit-maps with different frequencies of 1-bits. The following statistics were collected for each map:

1. the *compression factor* (CF) for each method, which is defined as the ratio of the size of the original map to the size of the compressed map;
2. the block-size pattern giving maximal compression;
3. the difference in compression efficiency between the best and the worst pattern.

The test showed that:

(a) For most vectors with up to 300 1-bits there was no tree, only a list of indices.

(b) The best parameter pattern changed very slightly with the different maps and was almost constant for maps with the same 1-bit frequency. In 91% of the sample the optimum was achieved with 5 levels ($t = 4$), and among the optimal 5-level trees, 54% used the pattern 11112.

(c) The compression factor was a non-increasing function of the number of 1-bits in the bit-map. For 18000 1-bits or more there was practically no compression, on the contrary, there were a few examples for which the optimal tree used more storage than the original bit-string.

(d) Using the best pattern for each of the 1040 maps of the sample, the average CF was about 16.7. This is an estimation based on the sample of 1040 bit-maps and taking into account the appropriate weight for each range of frequencies, e.g., the maps with up to 300 1-bits form about 80% of the file to be compressed.

(e) The total difference in compression between choosing the best or worst block-size patterns was very small, about 2.6% on the average.

As a consequence of **(a)**, we decided to restrict ourselves to the 56588 different words which appear more than 70 times in the current Responsa data base. From **(e)** we concluded that it is not worthwhile to search for the optimal method, and that we could choose a constant pattern for all the maps, which simplifies both the compression and decompression process. The best variant was that corresponding to the pattern 11112, but we preferred the variant with pattern 2222, since the additional amount of storage needed was only about half a percent, on the other side the uniformity of the latter variant and the fact that it used only three levels of compression permitted savings in the time complexity of the decompression routine.

3.2 Comparison with randomly generated bit-vectors

If we assume that each bit in v_0 has a constant probability for being 1, independently from each other, the expected size of the compressed vector obtained by the TREE-method can be evaluated as follows. Let p_i be the probability of a bit in level i to be 1, then for $0 \leq i \leq t$

$$\begin{aligned} p_i &= 1 - (1 - p_{i-1})^{r_{i-1}} \\ &= 1 - (1 - p_0)^{\prod_{j=0}^{i-1} r_j}, \end{aligned}$$

where we define $\prod_{j=0}^{-1} r_j = 1$ for convenience. Since for each block which will be kept at level i, there is a 1-bit in level $i + 1$, the expected number of (non-zero) blocks at level i is $k_i p_{i+1}$ for $i < t$. Thus the expected number of bits we keep in level i is $k_i p_{i+1} r_i = l_i p_{i+1}$, which holds also for $i = t$ if we define $p_{t+1} = 1$. Since

$$l_i = k_{i-1} = \frac{l_{i-1}}{r_{i-1}} = \frac{l_0}{\prod_{j=0}^{i-1} r_j} \qquad \text{for} \quad 0 \le i \le t,$$

we can express the expected size of the compressed tree (in bits) as:

$$\sum_{i=0}^{t} l_i p_{i+1} = l_t + l_0 \cdot \sum_{i=0}^{t-1} \frac{1-(1-p_0)^{\prod_{j=0}^{i} r_j}}{\prod_{j=0}^{i-1} r_j}. \tag{4}$$

Now the parameters in (4) are set as $l_0 = 42272$, $t = 3$, $r_j = 16$ for $0 \le j \le 3$, and p_0, the average frequency of the 1-bits in our sample of Responsa maps, is 0.00817. This yields a theoretical CF of 5.5 for the independent model, whereas the test on the Responsa maps showed a CF of 10.5. The difference should be credited to the inter-dependence of adjacent bits, which tends to produce more "compact" initial vectors, and thereby enhance compressibility.

In order to compare the random model with the Responsa maps also for the PRUNE-method, we decided to use computer simulation of random vectors. To permit fair comparison, the random maps were constructed with the same length (5284 bytes) and with a density-distribution similar to that of the Responsa maps. Toward this end the 56588 Responsa maps were partitioned into 101 classes corresponding to different ranges of the number of 1-bits in the maps. The number of generated maps was taken to be 10% of the number of the Responsa maps for each class. For each map a number K was chosen uniformly from the range, then K "random" bits were set to 1. In fact, the correct method of constructing the maps consists of deciding for every bit with probability p_0 if it should be set to 1, yielding an expected number of 1-bits equal to $p_0 l_0$. We preferred the first mentioned method which is more economical to implement and introduced only a slight error, the probability of a given bit to be set to 1 being $1-(1-1/l_0)^K \simeq K/l_0 = p_0$, since l_0 is large and K is relatively small (there are very few maps with large K). The random numbers were generated following Knuth [7].

The algorithm was applied to the complete file of 56588 Responsa maps, yielding an average CF of 17.5, which is better than expected from the test on the small sample. For the file of 5664 randomly generated maps, the CF was only 15.9. Thus, using the PRUNE-method, compression is still better for the Responsa maps than for the random maps, but the difference in compression is not as striking as for method TREE. As was pointed out earlier, the better results on information retrieval maps are due to the fact that the 1-bits appear in a more clustered form than expected by a model of independent bit-positions; hence for the randomly generated maps, the number of blocks at the lowest level which contain at least one 1-bit, will be larger. However, precisely these isolated bits are the main target of the pruning action. Therefore the improvement should be larger for random than for clustered maps, when both have the same 1-bit density.

3.3 Other Variants

The first variant we have experimented with, was to try to complement the bit-vector before its compression for bit-vectors with more than $l_0/2$ 1-bits. However the test showed that this was almost never worthwhile, even not for the most frequent words. An explanation of this fact could be that although the occurrence of a word W in a document may be strongly related to its occurrence in adjacent documents, this seems not always to be true as to the *absence* of W from consecutive documents. Thus complementing the vectors probably destroyed their clustered appearance.

Another variant was to try to reorganize the vectors into an even more clustered form. This obviously cannot be done simultaneously for all the vectors in an optimal, yet still efficient manner. Therefore the documents were simply arranged by decreasing size, so that reorganizing was in fact applying a fixed permutation on the original bit-vectors. The idea was that the probability of a word to appear is greater for a large document than for a small one. By concentrating the bit-positions corresponding to large documents in the same area of the vector, we expected a migration of the 1-bits towards this area. An indication that such a migration indeed happened was obtained by inspecting r_0, the block-size on the lowest level of the tree, which was constructed according to the optimal pattern among those of Table 1, for each map of the sample: the larger the clusters of 1-bits in the vector, the larger r_0 will be in the optimal pattern. Only for 12.7% of the sample of original maps, r_0 for the optimal pattern was larger than one byte, whereas for the permuted maps, this number increased to 57.4%.

The compression results however were rather disappointing: not a single example in our sample gave an improvement after the rearrangement; on the average, the CF decreased by 10.4%. We conclude that compression is enhanced not as much by the existence of a few large clusters, but more by the existence of many small ones. For the original vector, the documents are grouped by author, and for each author, ordered by topic, which yields many small clusters. This order is completely destroyed by the permutation, as there is almost no correlation between the size of a document and its content. The reorganization splits the clusters into single bits which are scattered throughout the vector more or less randomly. Although clusters will eventually form (in particular at the beginning of the vector, which corresponds to the larger documents), there will be also a large number of single bits and the overall compression efficiency decreases.

4. Comparison with other methods

One of the best known methods for bit-map compression is the run-length coding technique, which is borrowed from image compression. One often uses a fixed number b of bits to encode the length of a run. This is practically equivalent to the method which keeps a bit-map in form of a list of the indices of its 1-bits; instead of these indices, we store the *increments* between them. In our experiments on the Responsa maps, the average distance between 1-bits (which is the average length of runs of zeros) was 287.2 bits, but there were also very long runs, so we chose $b = 16$ to encode the run-lengths.

As to Jakobsson's [5] method of using Huffman codes for the 2^k possible k-bit blocks, any codeword has at least one bit, so we can never expect a better CF than k. On the other hand, k cannot be chosen too large since the number of different codewords to be generated, and the size of the encode and decode tables grow exponentially with k. We chose $k = 8$ to facilitate computer manipulation.

The best method in [4] combines Huffman coding and run-length coding. As in [5], the vector is partitioned into k-bit blocks, then the possible lengths of runs of 0-blocks are partitioned into classes C_i, containing run-lengths h which satisfy $2^{i-1} \leq h < 2^i$, for $1 \leq i \leq \lfloor \log_2(l_0/k) \rfloor$. The $2^k - 1$ non-zero block-patterns together with the classes C_i are then assigned Huffman codes; a run of length h belonging to class C_i is encoded by the codeword for C_i, followed by the $i - 1$ rightmost bits of the binary representation of $h - 2^{i-1}$. Again, $k = 8$ was chosen and we had classes C_1 to C_{13}.

Table 2: *Comparison of various methods of bit-vector compression*

	CF	Relative Improvement of method PRUNE
TREE	10.53	39.7%
Run-length	7.65	56.2%
Huffman coding	6.57	62.3%
Huffman+Run-length	18.77	-7.6%
PRUNE	17.45	–

The compression results are summarized in Table 2. There is one line for each technique; the methods from [5] and [4] are respectively entitled "Huffman coding" and "Huffman+Run-length". The middle column gives the compression factor as computed on the file of Responsa maps, the right hand column shows for each method how much relative additional savings (in percentage) can be obtained when it is replaced by the PRUNE-method. Note that this can be negative, since the method in [4] yields a higher CF. But the difference is small, and on the other hand, the latter method is more complicated; therefore decoding will be time-consuming, so that the new algorithm may be preferable in some applications even to the methods of [4].

Another criterion for the comparison of the compression methods is their sensitivity to errors. For run-length coding and Huffman coding, a single incorrectly transmitted bit usually renders the tail of the encoded string following the error useless. The combined Huffman+Run length method is slightly more robust than either of its two components. For the hierarchical methods, if a single parity change error occurs on the lowest level of the tree (and this is usually the major part of the compressed map), there is no damage other than the affected bit itself. If the error occurs in v_1 (the first level of compression), its effect will be like for Huffman or run-length codes: a suffix of the decompressed map will be garbled. Only errors in higher levels may produce completely false vectors, but these levels form only a small part of the compressed file. Using a few additional bytes per map, we can introduce partial error-detection: all we have to add are the lengths of each level in the compressed map. Since the length of level i is determined by the number of 1-bits in level $i+1$, a single error in the higher levels can be detected.

REFERENCES

[A1] **Choueka Y.,** Full text systems and Research in the Humanities, *Computers and the Humanities* **XIV** (1980) 153–169.

[A2] **Davis D.R., Lin A.D.**, Secondary key retrieval using an IBM 7090–1301 system, *Comm. of the ACM* **8** (1965) 243–246.

[A3] **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.

[A4] **Fraenkel A.S., Klein S.T.**, Novel Compression of sparse Bit-Strings — Preliminary Report, *Combinatorial Algorithms on Words*, NATO ASI Series Vol **F12**, Springer Verlag, Berlin (1985) 169–183.

[A5] **Jakobsson M.**, Huffman coding in Bit-Vector Compression, *Inf. Proc. Letters* **7** (1978) 304–307.

[A6] **Jakobsson M.**, Evaluation of a Hierarchical Bit-Vector Compression Technique, *Inf. Proc. Letters* **14** (1982) 147–149.

[A7] **Knuth D.E.,** *The Art of Computer Programming, Vol.* **II**, *Semi-numerical Algorithms*, Addison-Wesley, Reading, Mass. (1973).

[A8] **Nevalainen O., Jakobsson M., Berg R.**, Compression of clustered inverted files, in Proc. of the 7-th Symp. on Math. Foundations of Comp. Sc., Zakopane, Poland (1978) 393–402.

[A9] **Schuegraf E.J.**, Compression of large inverted files with hyperbolic term distribution, *Inf. Proc. and Management* **12** (1976) 377–384.

[A10] **Teuhola J.**, A Compression method for Clustered Bit-Vectors, *Inf. Proc. Letters* **7** (1978) 308–311.

[A11] **Vallarino O.**, On the use of bit-maps for multiple key retrieval, *SIGPLAN Notices, Special Issue* Vol. **II** (1976) 108–114.

[A12] **Wedekind H., Härder T.**, *Datenbanksysteme II*, B.-I. Wissenschaftsverlag, Mannheim (1976).