Construction of Optimal Graphs for Bit-Vector Compression

Abraham Bookstein and **Shmuel T. Klein**

Center for Information and Language Studies
University of Chicago, 1100 East 57-th Street
Chicago, IL 60637

The second author was partially supported by a fellowship of the Ameritech Foundation

Abstract:

Bitmaps are data structures occurring often in information retrieval. They are useful; they are also large and expensive to store. For this reason, considerable effort has been devoted to finding techniques for compressing them. These techniques are most effective for sparse bitmaps. We propose a preprocessing stage, in which bitmaps are first clustered and the clusters used to transform their member bitmaps into sparser ones, that can be more effectively compressed. The clustering method efficiently generates a graph structure on the bitmaps. The results of applying our algorithm to the Bible is presented: for some sets of bitmaps, our method almost doubled the compression savings.

1. Introduction

Textual Information Retrieval Systems (IRS) are voracious consumers of computer storage resources. Most conspicuous, of course, is the text itself, which constitutes the content of the database. But, to efficiently use the database, auxiliary structures must be created that themselves require a substantial ammount of space. Thus, mecanisms for compressing a wide range of data structures must be sought for the efficient operation of such systems [8]. To date, most attention has been given to, and progress made in, the area of text compression ([1], [11], [14]). In this paper, we shall describe and examine the possibilities of compressing bitmaps, a data structure often proposed for improving the performance of retrieval systems ([5], [16]).

Bitmaps occur often in information retrieval. They can represent the occurrences of a word in the sentences or paragraphs making up a text; they can indicate the documents associated with an index term; they appear as bit slices of a matrix of signatures; they might represent pixels in rows of a raster graphics display. They are useful; they are also large and expensive to store. Much work has been carried out on the compression of bitmaps, and this has been especially successful for those that are very sparse. But not all bitmaps are sparse, and even sparse bitmaps could benefit

from further compression. This paper describes a method that complements existing compression techniques and improves their performance, at least for certain categories of bitmaps.

We concentrate here on sets of bitmaps such as are generally found in an IRS. How such bitmaps can be used to enhance the system is discussed in [2] and [4]. Each bit-position corresponds to a specified sub-unit of the database, henceforth referred to as a *segment*; below, a segment will refer to a paragraph of text, though, in other contexts, a full document (or even a set of documents) may be the preferred unit. For each different word (or index term) W in the database, there is a map $B(W)$, such that the i-th bit of $B(W)$ is 1 if and only if W appears in (or is assigned to) segment i. Such bitmaps can be compressed very efficiently. In part this is because they tend to be very sparse. That bitmaps compress better as they become sparser is expected theoretically. For suppose a bitmap can be considered as having been produced by a random bit generator, with the probability of a one bit being p (the theory can easily be extended to encompass more complex models of bitmap generation). Then the information content of a bit is given by:

$$H = -p \log p - (1-p) \log(1-p),$$

and, for a bitmap of ℓ bits, the quantity ℓH forms a lower bound on the number of bits needed to represent the bitmap. As is well known, H increases monotonically as p increases from 0 to .5, and then decreases monotonically as p continues growing. Since almost all of our bitmaps have p less than .5, we expect compression to improve as p decreases, that is, as the map becomes sparser. (For $p > .5$, we could complement the bitmap before proceding.) Thus we wish to be alert to opportunities for reducing the density of our bitmaps; this is the essence of the approach described in this paper.

Other factors also contribute to our ability to compress bitmaps effectively, as evidenced by the fact that actual IR bitmaps are more compressible than randomly generated bitmaps with the same density of 1-bits [3]. The reason for the better results is a *cluster-effect:* since the segment positions in the bitmaps are usually ordered by topic or chronologically, adjacent bits often correspond to segments treating the same or related subjects. Thus the appearance of a word in a given segment often implies that it also appears in neighboring segments. This effect is exploited by many compression methods, resulting in excellent reduction in storage requirements.

There is, however, another clustering possibility that has hitherto been overlooked, one involving sets of bitmaps (words), rather than sets of bits (segments) within a single bitmap. The occurrences of certain words, especially those taking part in well known phrases like `Security Council` or `Curriculum vitæ`, are sometimes strongly correlated accross segments in the sense that if one word appears in a certain segment, the other is also very likely to do so. Such pairs of bitmaps are likely to be quite

similar. But identifying clusters of such highly associated words is not as direct as it was for bit clusters within a bitmap, because words and their associated bitmaps are generally arranged in lexicographical order, not in order of logical proximity. In this respect, IR bitmaps differ from self-clustered graphic bitmaps, in which adjacent (raster) rows are often similar. The objective of this paper is to show how to usefully identify clusters of correlated words, and then take advantage of these associations to squeeze out some additional compression.

In Section 2, we briefly review some known bitmap compression techniques and propose a new one that is simple and easy to implement; it will then be used as the compression component of a two-stage compression process described in Section 3. The first stage of the two stage process is to partition the bitmaps of our IR system into clusters of correlated bitmaps; the resulting clusters are then used to transform the original bitmaps into another set of bitmaps that are sparser and more effectively compressed in stage 2 of the process. In Section 4 we report on experiments testing the new method; the database we chose to study was the Hebrew Bible.

2. Bit-vector compression techniques

2.1 Overview of some known methods

Suppose we are given a bitmap v of length ℓ bits, of which s are ones and $\ell - s$ are zeros. In our applications, the maps are usually sparse, i.e., $s \ll \ell$. The simplest way to store v compactly for very small n is to **enumerate the positions** of the 1-bits. As one needs $d = \lceil \log_2 \ell \rceil$ bits to identify any position, this method would need sd bits for each map, which may well be much smaller than the ℓ bits required for the uncompressed original map. Alternatively, one could record the distances between successive 1-bits, that is, give the position of a 1-bit relative to the preceding 1-bit position rather than relative to the beginning of the vector. This is known as **run-length coding** (Schuegraf [12]). In its simplest form, the length of every run is encoded by a fixed length codeword; since this codeword must be large enough to accommodate the theoretical maximum run length, this is equivalent to the previous method.

Since, in simple run length coding, the space allocated for each run must be adequate for the largest possible run, such codes can be inefficient if many of the runs are of small or moderate length. The following variant, due to Teuhola [15], improves on simple run length coding by having a **variable length** representation of a **run length**. A run of r zeros is first broken up into successive blocks of zeros of exponentially increasing size; the first block is of size 2^k (for k a parameter selected to optimize this procedure), the second of size 2^{k+1}, etc., until a block is produced that extends beyond the run, i.e., is partially filled. The length of the run, r, can then be represented as follows: (a) each block in the sequence that is completely filled with

zeros is represented in turn by a one, and (b) a zero is appended to the string of ones as delimiter. If t ones are present, then we know $2^k + 2^{k+1} + \cdots + 2^{k+t-1} \leq r < 2^k + \cdots + 2^{k+t-1} + 2^{k+t}$, that is: the first t blocks are filled with zeros, but the last potential block of 2^{k+t} zeros is either empty or partially filled. So, finally, (c) we can explicitly represent the number of zeros in the last block as a binary integer with $k+t$ bits. Thus a run of length r is encoded by $O(\log r)$ bits instead of $O(\log(\text{max length}))$.

Jakobsson [7] suggests the use of **Huffman coding** for bitmaps. The bit-vector is partitioned into blocks of fixed size k, and statistics are collected on the frequency of occurrence of the 2^k bit patterns. Based on these statistics, the set of blocks is Huffman encoded, and the bitmap itself is encoded as a sequence of such codewords. For sparse vectors, the k-bit block consisting of zeros only, and blocks with only a single 1-bit, have much higher probabilities than the other blocks, so the average codeword length of the Huffman code will be smaller than k.

Fraenkel & Klein [6] combine **Huffman coding with run-length coding**. Once again, a parameter k is chosen as a block size. However, since for very sparse vectors the probability of a block of k zeros is high, runs of blocks of k zeros receive special treatment. We first represent the succession of k-bit blocks comprising a bitmap as a sequence of two categories of symbols: beginning with the first block, if a k-bit block includes 1-bits, then we represent it by its own special symbol, as in the previous method. If it is a zero-block, then instead of representing the block itself, we represent the entire run of zero blocks which it starts by a string of integers as follows: suppose the run consists of r zero-blocks, with r represented in binary form as $r = \sum_{i \geq 0} a_i 2^i$, for a_i zero or one. Then the run is represented in the symbol sequence by the string of integers $n_0, n_1, \ldots$, where each n_i is a power of 2 in the representation of r for which $a_{n_i} = 1$; this in effect encodes the run lengths. Next, the frequency of occurrence throughout the bitmap file of each special and integer symbol is recorded, permitting a Huffman tree to be constructed for the $2^k - 1$ special symbols together with the integer symbols. Finally, the bitmap is Huffman encoded using this tree.

A **hierarchical method** for compressing a sparse bitmap was proposed by Wedekind & Härder [17]. The original bit-vector v_0 of length ℓ_0 bits is partitioned into r_0 equal blocks of k_0 bits each ($r_0 \cdot k_0 = \ell_0$), and the blocks consisting only of zeros are dropped. The resulting sequence of non-zero blocks does not by itself allow the reconstruction of v_0; we can, however, append a list of the indices indicating where these non-zero blocks occur in the original vector. This list of up to r_0 indices is itself kept as a bit-vector v_1 of $\ell_1 = r_0$ bits; there is a 1 in position i of v_1 if and only if the i-th block of v_0 is not all zeros. Now v_1 can be further 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 is reached where the vector length reduces to a

few bytes. The compressed form of v_0 is then obtained by concatenating, in order of decreasing i, all the nonzero blocks of the various v_i. The same method appears in Vallarino [16], who used it for two-dimensional bitmaps, but with only one level of compression.

The hierarchical method is refined by Choueka & al. [3], by adding a **pruning algorithm** that removes from the hierarchy-tree, branches pointing to very few segments. The algorithm partitions the set of 1-bits in v_0 into two subsets: the class of 1-bits which are efficiently handled by the hierarchical method; and the complementary class, consisting of more or less isolated 1-bits whose inclusion in the hierarchical tree structure would have been more expensive than their enumeration in an appended list. Either of these two classes may be empty. If the list is long enough, it is further compressed by a variant of prefix omission, to be decribed in more detail in the following sub-section.

It should be noted that since, for each map, the number of runs of zeros is equal to the number of 1-bits plus 1, the size of the compressed file obtained by the first few methods is clearly linearly related to the number of 1-bits in the original file. For the hierarchical and Huffman coding methods this relation is less evident, but has been empirically established. This observation is consistent with the theoretical argument presented in the introduction, and reinforces our intention to design a preprocessing stage that reduces the number of bits in a bitmap.

2.2 A simple new method

The following technique is a simple generalization of the prefix omission method suggested in [3] for the secondary compression of the list of 1-bits which were pruned from the tree. It can also be viewed as a variant of the hierarchical method, using only a single level of compression.

Choose an integer parameter k and partition the original vector v_0 of length ℓ_0 into blocks of 2^k bits. We shall assume that the number of 1-bits in the bitmap is s. As in the hierarchical method, construct a new vector v_1 of length $\lceil \ell_0/2^k \rceil$, in which bit i is zero if and only if block i of v_0 contains only zeros. However, now, instead of storing the non-zero blocks of v_0 themselves, we substitute for each block the string of indices of the 1-bits within that block. *A priori* k bits are sufficient for storing such a relative index; however we need an additional bit per index to serve as a flag, which identifies the boundary of each block. Therefore, in addition to the fixed overhead of storing the vector v_1, $k+1$ bits are needed for representing each of the 1-bits of v_0. When a block has a small number of 1-bits, a significant saving in space could result.

We would now like to find k that optimizes the size of the block to be chosen, that is, the integer k^* that minimizes $f(k) = \lceil \ell_0 2^{-k} \rceil + (k+1)s$, the size in bits of the compressed bitmap. Because of the appearance of the ceiling function in $f(k)$,

finding the minimum value directly is difficult. Instead we shall search for an integer k_1^* that minimizes the related continuous function $f_1(k) = \ell_0 2^{-k} + (k+1)s$. Since f_1 is a convex function, k_1^* is an integer which satisfies

$$f_1(k_1^*) < f_1(k_1^* + 1) \qquad \text{and} \qquad f_1(k_1^*) \leq f_1(k_1^* - 1).$$

Examining the left hand inequality, we find

$$\ell_0 2^{-k_1^*} + (k_1^* + 1)s \;<\; \ell_0 2^{-k_1^*+1} + (k_1^* + 2)s,$$

or, $2^{-k_1^*+1} < s/\ell_0$. Thus $k_1^* > \log_2(\ell_0/s) - 1$. Similarly, the right hand inequality is equivalent to $k_1^* \leq \log_2(\ell_0/s)$. Combining the two, we find that k_1^* must satisfy

$$\log_2 \frac{\ell_0}{s} - 1 \;<\; k_1^* \;\leq\; \log_2 \frac{\ell_0}{s},$$

so $k_1^* = \lfloor \log_2(\ell_0/s) \rfloor$.

If we have a file of m bitmaps, we want to use the same method for encoding each of them, so the optimal k will be determined by the average $\bar{s}$ of 1-bits per map. The total number of bits in the compressed file is thus

$$m \left\lceil \ell_0 2^{-\lfloor \log_2(\ell_0/\bar{s}) \rfloor} \right\rceil \;+\; S\left(\lfloor \log_2(\ell_0/\bar{s}) \rfloor + 1\right), \tag{1}$$

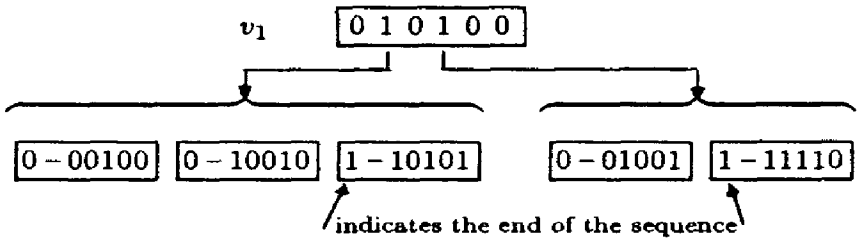
for $S = m\bar{s}$, the total number of 1-bits in the bitmap set.

A priori, k_1^* need not equal k^*; however, it is easy to see that the cost of using k_1^* is identical to the cost of using k^*. To show this, first note that by the definition of the ceiling function, $f(k_1^*) < 1 + f_1(k_1^*)$. Since k_1^* minimizes f_1, $f_1(k_1^*) \leq f_1(k^*)$. But f_1 cannot exceed f for any k, and in particular $f_1(k^*) \leq f(k^*)$. Combining these results with the fact than k^* minimizes f, we get

$$f(k^*) \;\leq\; f(k_1^*) \;<\; 1 + f(k^*).$$

We conclude that using k_1^* for the true optimum, k^*, results in an excess of less than one bit in storage for each bitmap. But $f(k)$ takes only integer values at integer k, so if the difference $f(k_1^*) - f(k^*)$ is smaller than 1, then it must actually equal zero: either $k_1^* = k^*$, or, at least, the storage implications are the same for both values ($f(k_1^*) = f(k^*)$).

For example, suppose $\ell_0 = 180$ and the indices of the 1-bits in v_0 are 36, 50, 53, 105 and 126. Thus $s = 5$, so we get that the optimal k is $\lfloor \log_2(180/5) \rfloor = 5$. There are $\lceil 180/2^5 \rceil = 6$ bits in v_1, each (except the last) corresponding to a block of 32 bits in v_0. There are three 1-bits in the second block, with relative indices 4, 18 and 21, and there are two 1-bits in the fourth block, with relative indices 9 and 30; the four other blocks are empty. Thus the following information would be kept:



The number of bits necessary to store this map is thus $6 + 5 \times (5 + 1) = 36$. With $k = 4$ we would need $12 + 5 \times (4 + 1) = 37$ bits and with $k = 6$ we would need $3 + 5 \times (6 + 1) = 38$ bits. Note that if we list the relative indices of each sub-range in increasing order, the flag identifying the last index of each range is not always needed. In our example, for instance, the list of stored relative indices is 4, 18, 21, 9, 30, so clearly the sublist corresponding to the second 1-bit in v_1 consists of the last two elements. If, however, there were no 1-bit in position 105 of v_0, the list of stored relative indices would have been 4, 18, 21, 30, and the partition of this list into two increasing sub-lists is not uniquely determined.

3. Bitmap Clustering

3.1 Motivation

We have remarked several times above that sparser bitmaps are more effectively compressed. We will now describe a method for reducing the number of 1-bits by making use of a natural clustering of bitmaps. To do this, we take advantage of the fact that many bitmaps are associated in the sense that the presence of a 1-bit in one map increases the likelihood of a 1-bit occurring in the same position in the other. If two bitmaps X_1 and X_2 are strongly associated in this sense, then the bitmap $X_3 = X_1$ XOR X_2 will very possibly have fewer 1-bits than, say, X_2. If we store X_1 and X_3, we can reconstruct X_2. The advantage of doing this is that we may be able to compress X_1 and X_3 more effectively than the original vectors. Since our intention when XORing two vectors is to reduce the number of 1-bits, it is useful to take as a measure of association between two vectors, the number of 1-bits in the XORed vector. But this quantity is the familiar *Hamming distance* between the two vectors. If the maps X_i and X_j are "close" in the Hamming distance sense, we would want to keep $\bar{X}_j$ and the pair $(\bar{x}_i, j)$ instead of $\bar{X}_j$ and $\bar{X}_i$; here $x_i = X_i$ XOR X_j, and a bar indicates that the maps have been compressed, say by the method presented in Section 2.2. Given the retained information, the original bitmap can then be recovered by first decompressing $\bar{x}_i$ and $\bar{X}_j$, which yields x_i and X_j, and finally XORing again, since $X_i = x_i$ XOR X_j.

As described above, the unchanged map X_j is compressed directly. However, X_j may itself be quite close to a third map, X_k, and therefore profitably XORed with that third map, producing the pair $(\bar{x}_j, k)$. Continuing in this manner we impose a structure on the bitmaps that can be represented as a directed graph, $G = (V, E)$,

where the vertices $V = \{X_1, \ldots, X_m\}$ correspond to the bitmaps and (X_i, X_j), the directed edge from X_i to X_j, belongs to E if and only if X_i is compressed as $(\bar{x}_i, j)$.

To be workable, the following restrictions must be imposed on G. **(1)** Any map can be compressed by XORing with at most one other map, so *the outdegree of every vertex is at most 1.* **(2)** In a general graph satisfying condition (1), it might be possible to form a chain of bitmaps $X_1, X_2, \ldots, X_k, X_1$, denoting that X_i is stored as $(\bar{x}_i, i+1)$ for $i = 1, \ldots, k-1$, and X_k is stored as $(\bar{x}_k, 1)$. However, this situation must be prohibited, if we want to be able to recover the original bitmaps: starting with an arbitrary node, the chain must terminate with an untransformed bitmap, that is, with a node with outdegree zero. In other words, *a legitimate graph must be cycle free.*

These conditions impose a strong structure on a legitimate graph. Let $R = \{r_1, \ldots, r_n\}$ be the set of vertices with outdegree zero, and define $\mathcal{T}(r_i)$ as the set of vertices from which there is a directed path to r_i; $\mathcal{T}(r_i)$ also includes r_i (connected to itself by the empty path). Since there are no cycles in G, a directed path starting at any vertex $X \in V$ must eventually terminate, reaching one of the vertices $r_i \in R$. Thus every $X \in V$ is in one of the $\mathcal{T}(r_i)$. If $X \in \mathcal{T}(r_i) \cap \mathcal{T}(r_j)$ for $i \neq j$, then some vertex in the chain starting at X must have outdegree ≥ 2. Since this is impossible, the components $\mathcal{T}(r_i)$ are disjoint and $\{\mathcal{T}(r_i)\}$ is a partition of V into connected *clusters* of bitmaps. Further, there is no linkage between any pair $\mathcal{T}(r_i)$ and $\mathcal{T}(r_j)$: for suppose $X_1 \in \mathcal{T}(r_i)$ and an edge (X_1, X_2) exists with $X_2 \in \mathcal{T}(r_j)$. But then, since a path exists connecting X_2 to r_j, a path exists (through X_2) connecting X_1 to r_j. Such a node X_1 is a member of both $\mathcal{T}(r_i)$ and $\mathcal{T}(r_j)$, which is impossible. The $\mathcal{T}(r_i)$ are thus isolated connected components in G; because of conditions (1) and (2), each $\mathcal{T}(r_i)$ is an *oriented tree*, as defined by Knuth [9, Section 2.3.4.2].

Any forest of bitmaps can serve as the basis of our precompression operations. To maximize compressibility, however, we want to choose that forest among all possible forests that minimizes the total number of ones in the resulting bitmaps. (There could conceivably exist some maps which, because of their special internal structure, yield better compression than others which are sparser. But until a quantitative relationship can be derived between detailed bitmap characteristics and compression, sparseness is the best measure we have for bitmap compressibility.) We define the quantity to be minimized, that is, the total number of 1-bits in the roots plus the total number of 1-bits in the XORed bitmaps, as the *cost* C of the forest. Note that adopting this criterion prevents our XORing two vectors when the result would increase the number of 1-bits — for example, in the extreme case, the set of original bitmaps, with no XORed maps, is forest and thus a legitimate graph.

An exhaustive search generating all the possible graphs satisfying our constraints and checking for each the cost for the forest, must be ruled out on the grounds of computational expense, even if we have only a moderately large number m of bitmaps. Fortunately, such a search is unnecessary, as the problem is equivalent to another for which there are well known polynomial algorithms. To see this, we first recall that,

except for the roots, the number of 1's in a XORed bitmap is just the Hamming distance between it and its successor in the directed graph. Thus, if we assign this distance as a weight to each edge, the cost of a forest is simply the sum of the weights of all edges in the graph plus the sum of the number of ones in each root. But we can further simplify the statement of the problem by noting that the number of ones in a map is its Hamming distance to the zero bitmap (the bitmap, all of whose values are zero), denoted by X_0. Thus, given any forest, if we introduce the zero bitmap and include the weighted edge between each root and X_0 (thereby transforming the forest into a tree), then the cost of the original forest is just equal to the sum of edge-weights over all the edges of the resulting tree in the enhanced graph. The latter sum will be called the *cost* of the tree. Since only the weights are significant when computing the cost, we can consider the tree as being non-directed. Such a simplification is well defined since the weight on an edge does not depend on its orientation (the Hamming distance is a symetric measure). Thus given any directed forest over the set of nodes V, we can define a non-directed tree over $V^* = V \cup \{X_0\}$ having the same cost.

The converse is also true. First note that given the set of vertices V^*, any (undirected) spanning tree on V^* defines a directed tree on V^*: the root of the directed tree in X_0; the directed edge (X_i, X_j) is in the directed tree if a path $(X_i, X_j, \ldots, X_0)$ exists in the undirected tree. Next, by removing X_0 and the edges incident on it from the directed tree, we obtain a directed forest G, on V. Furthermore, if the edge weights are as defined above, the cost of the forest is equal to the cost of the tree that induced it. Because of this equivalence, an optimal forest is associated with an optimal (lowest cost) tree. Thus our problem is equivalent to the following one: given a complete non-directed graph whose vertex set is the union of our bitmaps with X_0, and for which the weight on edge (i, j) is the Hamming distance between vertices i and j, find the tree for which the total edge weight is minimum. The directed forest induced by this tree is the solution to our problem.

More formally: we are looking for a graph G, which is a forest of oriented trees spanning the vertex set V, optimizing our problem. To find the graph G, we consider the weighted undirected graph $G^* = (V^*, E^*)$, where the set of vertices V^* is obtained by adjoining a new vertex, the zero vector X_0, to the set V of bitmaps; $E^* = V^* \times V^* - \{(X_i, X_i) : X_i \in V^*\}$ (ignoring order), that is, G^* is a complete graph from which self-loops are removed; and the weight $w(i, j)$ associated with the edge $(X_i, X_j) \in E^*$ is the Hamming distance between X_i and X_j. We then define as a legitimate sub-graph of G^* a non-directed tree T connecting all the vertices in V^*. Our task is to find the legitimate sub-graph for which the sum of all the weights of the edges in T is minimized — in fact, a minimum spanning tree (MST) of G^*. The MST in G^* now induces the optimal directed forest, G, on the original set of bitmaps, as described above. The vertices that were adjacent to vertex X_0 in T are the roots of the oriented trees in G. G is the optimal forest we were seeking.

Many algorithms for finding a MST for a non-directed graph appear in the literature, ranging from Kruskal's simple greedy algorithm [10], which has in our case

complexity $O(m^2 \log m)$, to Yao's more involved technique [18], which would need $O(m^2 \log\log n)$ operations for our application.

3.2 Algorithm statement

Summarizing, we suggest the following procedure as the first stage for compressing a set of m bitmaps $X_1, \ldots, X_m$. This method in principle improves any given compression algorithm $\mathcal{C}$ for individual bitmaps, provided our assumption of strong correlation between some of the maps holds. As output, we get a table B of compressed bitmaps, the compressed form of X_i being stored in $B(i)$, $1 \leq i \leq m$. In addition, the algorithm produces a small table F of size m, defined by $F(i) = j$ if the map X_i is compressed as $(\bar{x}_i, j)$ (i.e., if X_j is the father of X_i in the oriented rooted tree T), or by $F(i) = 0$, if X_i is the root of one of the trees.

1. Choose a compression method $\mathcal{C}$ for an individual map: given a bitmap X, $\mathcal{C}(X)$ is the result of $\mathcal{C}$ applied to X.
2. Extend the set of bitmaps by adjoining X_0, the zero-vector.
3. (a) Using the Hamming distances as weights on the complete graph without self-loops having $\{X_0, X_1, \ldots, X_m\}$ as set of vertices, compute a minimum spanning tree T.

   (b) Consider T as an oriented tree rooted at X_0.

   (c) The subtrees of X_0 in T partition the original set of bitmaps.
4. (a) If X_i is a vertex adjacent to X_0 in T, then it is the root of one of the oriented trees: these bitmaps (one per tree) are compressed directly using $\mathcal{C}$.

   $$B(i) \leftarrow \mathcal{C}(X_i)$$
   $$F(i) \leftarrow 0$$

   (b) A bitmap X_i, which is not the root of a tree, has a directed edge to another bitmap X_j in the same tree; X_i is compressed by first computing $x_i = X_i$ XOR X_j and then compressing x_i using $\mathcal{C}$.

   $$B(i) \leftarrow \mathcal{C}(x_i)$$
   $$F(i) \leftarrow j$$

In the case of a set of bitmaps of an IR system, the problems of compression and decompression are not exactly symetric. Compression is performed only once, during the construction of the system, and is applied to the entire set. Decompression, on the other hand, is practically never needed simultaneously for the entire set, but only for those maps associated with the keywords of a submitted query. We thus present the procedure **decompress**(i) which returns the original map X_i. It uses the function $\mathcal{C}^{-1}$ as the inverse of the compression function $\mathcal{C}$ — that is $\mathcal{C}^{-1}$ decompresses bitmaps which have been compressed by $\mathcal{C}$.

```
decompress(i)
        if  i = 0    return( C^{-1}(B(i)) )
        else         return( C^{-1}(B(i)) XOR decompress(F(i)) )
```

We see that the savings in storage space gained by our clustering procedure come at the expense of increased processing time. In order to recover the bitmap X_i, we need decompress all the maps forming the path from X_i to the root of the cluster X_i belongs to.

4. Example

The database we chose for testing our algorithm is the Hebrew Bible, consisting of 305514 words which are partitioned into 929 chapters. The number of different words is 39647. Following the suggestion in [4] that bitmaps should be constructed only for words which appear more often than some fixed frequency threshold, we restricted ourselves to the 1478 words which appeared in at least 20 chapters. As a text segment, we defined a set of four consecutive chapters. The resulting bitmaps were $\lceil 929/4 \rceil = 233$ bits long. The total number of 1-bits in the 1478 maps was 65734, or $s = 44.47$ 1-bits per map.

We first used the compression technique of Section 2.2 by itself. The optimal parameter k was $\lfloor \log_2(233/44.47) \rfloor = 2$. From equation (1) we thus get that the total number of bits needed to store the set of bitmaps in compressed form is 284404. For the uncompressed file we would need $1478 \times 233 = 344374$ bits, so that the simple method yields 17.4% compression. This k is indeed optimal for this method, since with $k = 1$ we get 11.6% compression, and with $k = 3$ we get 10.8%.

We then applied Kruskal's MST algorithm, which partitioned the set of bitmaps into 716 clusters. Of these, 530 were singletons, i.e., maps which couldn't effectively be XORed with some other map and which were therefore compressed without transformation. The other 948 bitmaps were partitioned into 186 clusters, each containing at least two elements. Since in each cluster, the root is compressed directly, the number of bitmaps which were XORed before compression was $948 - 186 = 762$. For these, the total number of 1-bits decreased from 48590 to 33538, that is, by 31%.

Considering the entire file of bitmaps, the overall number of 1-bits decreased from 65734 to 50658, or to $\bar{s} = 34.27$ 1-bits per bitmap. The optimal parameter k was thus $\lfloor \log_2(233/34.27) \rfloor = 2$, as before. Substituting the values for m, S and $\bar{s}$ in equation (1), we find that the total number of bits needed to store the set of bitmaps if we use the clustering method of Section 3 is 239176. Relative to the noncompressed file this is a 30.5% reduction, and relative to using only the method of Section 2.2 without clustering, this is a 15.9% improvement.

It is interesting to compare this to the information theoretic estimate of compresibility mentioned in the introduction. The probability of a 1-bit in the original

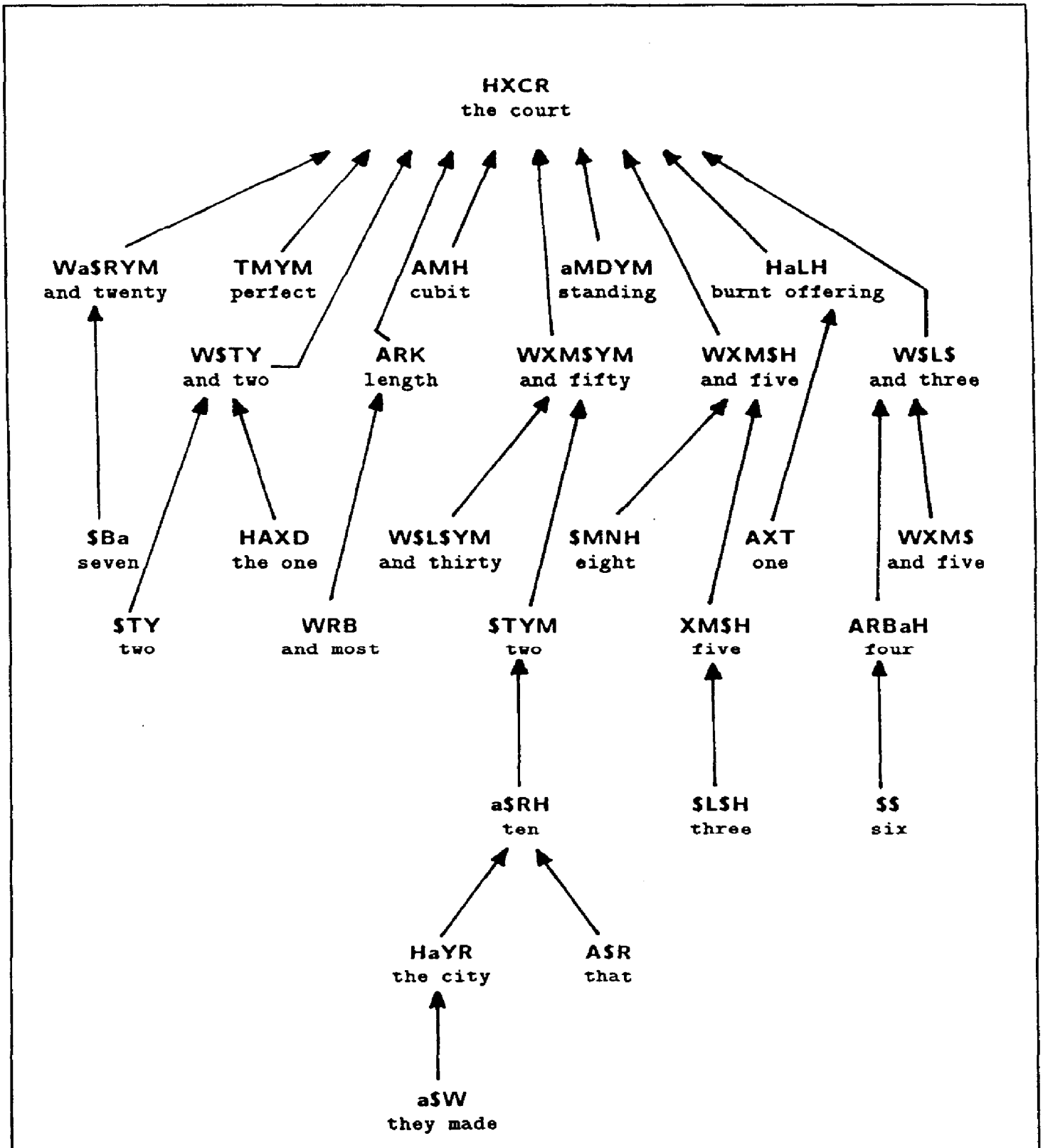


Figure 1: Sample cluster produced by the MST algorithm

file is 0.19, yielding an entropy per bit of $H = 0.703$. This means that if the 1-bits appear with the given frequency but independently from each other, the best possible compression would be 29.7%. Indeed, we got only 17.4% when the bitmaps were compressed individually. Introducing the clustering, we exploit the dependencies between different bitmaps, yielding compression savings of 30.5%, which is beyond those possible for independently generated maps.

While most of the generated clusters were small (two to four elements), some formed deep trees with tens of bitmaps, and the largest consisted of a tree of depth 15 with 112 vertices. A closer look at some of the larger clusters revealed interesting associations. Figure 1 shows a typical example. For each node in the tree, the Hebrew word is first given in English translitteration, using {**ABGDHWZXtYKLMNSaPCQR$T**} respectively for {*aleph, beth, ..., tav*}, as well as the translation of the word into English.

In this cluster, 18 out of 28 words are numerals; these are clearly connected, as the Bible tends to give exact dimensions (note the words `length` and `cubit`) in certain detailed descriptions. See, for instance, Exodus 27:9–19, where a description of the `court`, the root of this cluster, is given. The depth of this tree is 5, which is therefore the maximal depth of the recursion for the decompression algorithm. Note also that the root of this cluster has a high in-degree. This was not always the case, as can be seen in the following example.

To present the second example, we use a more compact representation, based on pre-order traversal of a tree. A tree can be represented recursively by its root, followed by the list of its subtrees enclosed in parenthesis. To improve readability, the level of the root of a subtree in the full tree appears as subscripts to the parentheses. We now give only the English translation of the word at each node (many Hebrew words must be translated into several English words).

> `and they camped (`$_1$ `night, and they saw, and they went out (`$_2$ `the men (`$_3$ `and they came (`$_4$ `from before, and he sent (`$_5$ `and they said, and he sat (`$_6$ `and he, his people, and he went out )`$_6$`, and he called (`$_6$ `and he came (`$_7$ `and now (`$_8$ `please )`$_8$ `)`$_7$`, and he gave (`$_7$ `in the hand of )`$_7$`, and he took (`$_7$ `bread, and he did (`$_8$ `two )`$_8$ $)_7$ $)_6$ $)_5$`, and they sat )`$_4$`, and they went )`$_3$ $)_2$ $)_1$

The 25 elements in this cluster form a tree of depth 8, but no node has higher in-degree than 3. Note that most of the words are verbs related to motion, all in the past tense, and in third person singular or plural.

We also tried to apply the algorithm to sparser bitmaps, by defining a segment to be one, instead of four, chapters. The 1478 bitmaps then had a total of 95472 1-bits, so they were compressed with $k = 3$ and gave 59.9% compression. The clustering algorithm however produced only 300 bitmaps that were XORed; for these, the reduction in the number of 1-bits was about 21%, but nonetheless, the total number of 1-bits remained quite large at 85195 bits. The optimal k now shifted to be 4, and

compression was improved by 7.5%; measured relative to the full file, a 62.9% reduction was achieved. In this case, the theoretical optimum for independently generated bitmaps with this 1-bit density is 63.8%.

In order to check the influence of the language of the database on the algorithm, we repeated the experiments with the *King James Bible*, again with a segment equal to one chapter. The improvement of the clustering method in this case was only 5.7%, again because only a small number of bitmaps (377 of 1454) were XORed. There were nevertheless some interesting clusters. For example: `Asher` $(_1$ `Ephraim, Joseph, Manasseh, Simeon` $(_2$ `Levi, Reuben` $(_3$ `Gad` $)_3$ $)_2$, `Zebulun` $(_2$ `Naphtali` $)_2$, `Issachar` $(_2$ `Benjamin, Dan` $)_2$ $)_1$. This cluster contains the names of all the tribes, except `Judah`. The latter appears in another cluster, together with words like `Jerusalem`, `reign`, `reigned`, `kings`, etc. Clearly, `Judah` differs from the other tribes, as his name often refers to the kingdom or land of `Judah`.

Summarizing our experiments, we see that the clustering algorithm works better when the bitmaps are not so sparse: very sparse vectors tend to have very few overlapping 1-bits, so that there is often no gain to be achieved by XORing. However, for the very sparse vectors, many of the known techniques already yield excellent results. Thus the clustering algorithm helps especially for those maps that are most difficult to compress.

5. Conclusion and Future Work

We have presented a new algorithm for transforming a set of bitmaps, which in principle may improve any previous compression method that does not take into account possible interrelationships among the different bitmaps. The experimental results suggest that the new method is particularly effective for bitmaps which are not extremely sparse. This may have several applications.

For example, bit-slices of signature methods are often chosen so that the density of 1-bits is $\frac{1}{2}$ [13]. Such vectors are almost impossible to compress individually. There may however be a possible gain by using clustering. Also, the possibility of compression would permit us to increase the size of the signature, resulting in more efficient retrieval, whitout affecting the space requirements [2]. Another application would be to IR bitmaps which have already been slightly compacted, such as the maps obtained by applying one iteration of the hierarchical bit-vector compression technique referred to in Section 2.1. Finally, there might be applications to areas outside of IR, such as image compression, where adjacent raster rows may be similar, or processing of genetic information, where different DNA strings often share long identical substrings.

REFERENCES

[1] **Bell T., Witten I.H., Cleary J.G.,** Modeling for Text Compression, *ACM Computing Surveys* **21** (1989) 557–591.

[2] **Bookstein A., Klein S.T.,** Using Bitmaps for Medium Sized Information Retrieval Systems, to appear in *Inf. Proc. and Management* (1990).

[3] **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; ACM, Baltimore, MD (1986) 88–97.

[4] **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.

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

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

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

[8] **Klein S.T., Bookstein A., Deerwester S.,** Storing Text Retrieval Systems on CD-ROM: Compression and Encryption Considerations, *ACM Trans. on Information Systems* **7** (1989), 230–245.

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

[10] **Kruskal J.B.,** On the shortest spanning subtree of a graph and the Travelling Salesman Problem, *Proc. Amer. Math. Soc.* **7** (1956) 48–50.

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

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

[13] **Stiassny S.,** Mathematical analysis of various superimposed coding methods, *Amer. Documentation* **11** (1960) 155–169.

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

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

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

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

[18] **Yao A.C.C.**, An $O(|E| \log\log |V|)$ algorithm for finding minimum spanning trees, *Inf. Processing Letters* **4** (1975) 21–23.