

- [19] J. Teuhola. A compression method for clustered bit-vectors. *Information Processing Letters*, 7(6):308–311, October 1978.
- [20] B. Tuthill. Refer—a bibliography system. In *Unix User's Manual Supplementary Documents*. 4.2 Berkeley Software Distribution, Berkeley, California, 1984.
- [21] I.H. Witten, T.C. Bell, and C. Nevill. Models for compression in full-text retrieval systems. In J.A. Storer and J.H. Reif, editors, *Proc. IEEE Data Compression Conference*, pages 23–32, Snowbird, Utah, April 1991.
- [22] I.H. Witten, R. Neal, and J.G. Cleary. Arithmetic coding for data compression. *Communications of the ACM*, 30(6):520–541, June 1987.

Acknowledgements

This work was supported by the Australian Research Council.

References

- [1] A. Bookstein and S.T. Klein. Compression of correlated bit-vectors. *Information Systems*, 16(4):387–400, 1991.
- [2] A. Bookstein and S.T. Klein. Flexible compression for bitmap sets. In J.A. Storer and J.H. Reif, editors, *Proc. IEEE Data Compression Conference*, pages 402–410, Snowbird, Utah, April 1991.
- [3] A. Bookstein and S.T. Klein. Generative models for bitmap sets with compression applications. In *Proc. 14th ACM-SIGIR Conference on Information Retrieval*, pages 63–71, Chicago, 1991.
- [4] Y. Choueka, A.S. Fraenkel, and S.T. Klein. Compression of concordances in full-text retrieval systems. In *Proc. 11th ACM-SIGIR Conference on Information Retrieval*, pages 597–612, Grenoble, France, June 1988. ACM, New York.
- [5] Y. Choueka, A.S. Fraenkel, S.T. Klein, and E. Segal. Improved hierarchical bit-vector compression in document retrieval systems. In *Proc. 9th ACM-SIGIR Conference on Information Retrieval*, pages 88–97, Pisa, Italy, September 1986. ACM, New York.
- [6] W.B. Croft and P. Savino. Implementing ranking strategies using text signatures. *ACM Transactions on Office Information Systems*, 6(1):42–62, 1988.
- [7] P. Elias. Universal codeword sets and representations of the integers. *IEEE Transactions on Information Theory*, IT-21:194–203, March 1975.
- [8] A.S. Fraenkel and S.T. Klein. Novel compression of sparse bit-strings—Preliminary report. In A. Apostolico and Z. Galil, editors, *Combinatorial Algorithms on Words, Volume 12*, NATO ASI Series F, pages 169–183, Berlin, 1985. Springer-Verlag.
- [9] R.G. Gallager and D.C. Van Voorhis. Optimal source codes for geometrically distributed alphabets. *IEEE Transactions on Information Theory*, IT-21(2):228–230, March 1975.
- [10] S.W. Golomb. Run-length encodings. *IEEE Transactions on Information Theory*, IT-12(3):399–401, July 1966.
- [11] D.A. Huffman. A method for the construction of minimum redundancy codes. *Proc. IRE*, 40(9):1098–1101, September 1952.
- [12] M. Jakobsson. Huffman coding in bit-vector compression. *Information Processing Letters*, 7(6):304–307, October 1978.
- [13] M. Lesk. Some applications of inverted indexes on the Unix system. In *Unix Programmers Manual, Volume 2A*. Bell Laboratories, Murray Hill, New Jersey, 1988.
- [14] M.D. McIlroy. Development of a spelling list. *IEEE Transactions on Communications*, COM-30(1):91–99, January 1982.
- [15] A.M. Moffat and J. Zobel. Coding for compression in full-text retrieval systems. To appear in *Proc. IEEE Data Compression Conference*, Snowbird, March 1992.
- [16] G. Salton and M.J. McGill. *Introduction to Modern Information Retrieval*. McGraw-Hill, New York, 1983.
- [17] E.J. Schuegraf. Compression of large inverted files with hyperbolic term distribution. *Information Processing & Management*, 12:377–384, 1976.
- [18] Z. Somogyi. The Melbourne University bibliography system. Technical Report 90/3, Department of Computer Science, The University of Melbourne, Parkville, Victoria 3052, Australia, March 1990.

	Compression %		
	<i>Manuals</i>	<i>GNUbib</i>	<i>Comact</i>
γ	20.1	7.4	9.7
δ	22.3	7.8	10.9
Huffman	19.7	7.0	8.9
Huffman (batched)	19.5	7.0	8.8
Entropy	19.4	3.2	8.1
Entropy (batched)	18.8	2.9	7.7

Table 9: Coding word-document frequencies

that would be achieved if arithmetic coding was used with the same models. The decrease in size for *GNUbib* is dramatic.

Use of arithmetic coding would mean that the word frequencies would have to be stored separately from the bitvector, but adjacent storage would mean that no extra disk time would be required to retrieve both bitvector and frequencies. Although not physically interleaved, the arithmetic coded frequencies would still be logically interleaved.

Regardless of the particular model or coding method used, it is worth making the point that storing the word frequencies for each word-document pair within the document collection can be achieved for a relatively small additional cost. To put all of the compression ratios back into perspective, *Comact* is a collection of 261,829 documents totalling 132 Mbyte uncompressed and without indexing; the techniques described here allow the bitmap to be represented in under 8 Mbyte, and the word frequency information to be stored in an additional 2.8 Mbyte. Compared with a raw bitmap, at 2 Gbyte, and an uncompressed inverted file, at 30.5 Mbyte, significant savings have been achieved.

10 Summary

We have shown, for several different methods for encoding bitmaps, that parameterising the compression for individual bitvectors allows improved compression. The methods we have successfully modified include the uniform random (or Bernoulli) model of bitmap generation; the ‘Exp-Gol’ code of Teuhola [19]; the LLRUN method of Fraenkel and Klein [8]; and a general Huffman code on run lengths. We have also described a technique whereby the frequency of each word within each document can be added to the bitmap at small additional cost.

We have applied the new methods to three experimental document collections. In each case we were able to demonstrate improved compression performance. The methods we have described also appear to outperform other bitmap compression techniques that have appeared in the literature.

Although we have been able to improve the compression obtained, it has been at the cost of some sophistication in terms of models and extra processing time during encoding. It is worth noting that Elias’ γ and δ codes are surprisingly effective for storing these bitmaps and frequency maps. For practical implementations these simple one-pass codes should also be considered.

and correspond to the ‘Total’ rows of Tables 4 to 7. The best of the methods described by Bookstein and Klein gave rise to a compression improvement of 16% on the Hebrew Bible [3]; the methods here give a 30% improvement on roughly comparable data. We also obtained similar results when we repeated the experiments selecting bitvectors for words that appeared in 25 or more of the chapters, where $25 = 1189 \times 20/929$.

Bookstein and Klein [3] compared their methods with several others, and claimed to have achieved improved compression. We thus have some confidence in the general usefulness of the techniques we have described. Nevertheless, the comparative results are indicative only, and should not be regarded as conclusive. It would be interesting to compare the various methods on identical test data.

9 Storing Word Frequencies

A bitmap is used to determine whether a particular word appears in a particular document. There will also be situations in which it is useful to know not only whether or not the word appears, but also how many *times* it appears. For example, documents can be ranked more effectively, via similarity measures such as the cosine measure [16], if term frequency information is known: including term frequency information results in a 10%-20% improvement in ranking effectiveness [6]. To allow such ranking, the bitmap should store a frequency count for each word-document combination instead of a bit, and, strictly speaking, will not be a bitmap at all. We call this structure a *frequency map*; and will refer to one row of the frequency map as a *frequency vector*.

We have also considered how frequency vectors might be economically stored. Most frequency vectors will still be sparse, with many zeroes (representing documents in which that word does not appear at all), some ones (documents in which the word appears once), and a few greater values (documents in which the word appears several times).

It is relatively straightforward to add the frequency information to the compressed bitmap forms already described. Each runlength of zeros is encoded in the normal manner, but is then followed by a code describing the number of occurrences of the term in that document. The decoder must extract this code before decompressing the next run length. Because the frequency counts will normally be small integers, predominantly 1, it follows that δ and γ are again suitable coding methods, since both use just one bit to describe the value one. Table 9 shows the extra cost of storing the frequency information using γ , δ , and a Huffman code based upon the observed word-document frequencies. For consistency with the previous tables we have again measured the compressed size as a percentage of the uncompressed inverted file, although in this case there is little relevance in the ratio.

Adding the frequency information to the map increases its compressed size by about one-third. Although the frequency information is stored interleaved with the bitmap, these three methods make no use of any commonality, and so are global methods.

If we assume that the value $\lfloor \log p_w \rfloor$ is known for ‘free’, that is, that a parameterised compression scheme is being used for the bitmap component of the frequency map, we can do fractionally better. Table 9 also shows the compression attained with a batched version of the Huffman code, where the encoding used for the frequency is conditioned on the selector $\lfloor \log_2 p_w \rfloor$. All of the values in Table 9 include the small overhead of storing the model(s).

It is also worth noting that in this case there is a definite gain from converting to arithmetic coding [22] rather than using Huffman coding, since the distributions are skewed very heavily in favour of ‘one’. The rows marked ‘Entropy’ in Table 9 show the compression

	Compression %		
	<i>Manuals</i>	<i>GNUbib</i>	<i>Comact</i>
Huffman (global)	49.8	32.3	25.9
Global model	0.5	0.7	0.1
Total	50.3	33.0	26.0
Huffman (batched)	44.1	30.2	24.6
$\lceil \log N \rceil$ models	1.6	0.9	0.2
Selection overhead	1.5	0.4	0.1
Total	47.2	31.5	24.9

Table 7: Huffman coding the run lengths

Bookstein and Klein [3] describe experiments based upon the Hebrew Bible. They report that this Bible has 929 chapters and 305,514 words, and they considered a retrieval system in which each chapter was considered to be a document (that is, $N = 929$). In their experiments they extracted the bitvectors for all words that occurred in 20 or more of the chapters, which resulted in $n = 1,428$ bitvectors in which a total of $95,472 = 7.2\%$ bits were set to one. Their best methods resulted in compression equivalent to a 16% saving compared with the zero-order self entropy of the bitmap, considered globally, where, for a bitmap containing n_0 zero bits and n_1 1-bits the zero-order self entropy is given by

$$n_0 \cdot \log \frac{n_0 + n_1}{n_0} + n_1 \cdot \log \frac{n_0 + n_1}{n_1}.$$

Because all infrequent words were discarded, their results are not directly comparable with ours. However in an attempt to duplicate their experiment we tested our methods on a roughly similar selection of bitvectors drawn from the King James Bible. That Bible contains 1,189 chapters; 824,965 words; and 13,777 distinct words. To achieve roughly the same overall percentage of 1-bits we needed to extract the bitvectors for all words that occurred in 10 or more of the chapters, and this resulted in a bitmap of $n = 3,266$ bitvectors, each $N = 1,189$ bits long, and with a total of $p = 276,820 = 7.1\%$ 1-bits.

Compression results using this modified bitmap are shown in Table 8. The figures in the second column are compression ratios calculated as described in Section 1; the figures in the third column are *compression improvements* relative to the self entropy of the bitmap, calculated in the manner used in [3]. All of the values given include the appropriate overheads,

Method	Compression %	% Saving relative to self-entropy
Zero order self entropy	47.3	0.0
γ	35.3	25.3
V_G (local)	34.1	27.9
V_T (optimal)	32.7	30.7
LLRUN (batched)	32.4	31.5
Huffman (batched)	32.1	32.1

Table 8: Compression on selected bitvectors in the Bible

	Compression %		
	<i>Manuals</i>	<i>GNUbib</i>	<i>Comact</i>
LLRUN [8]	51.1	32.7	26.1
LLRUN (batched)	46.2	30.4	24.9
Overheads	1.5	0.4	0.1
Total	47.7	30.8	24.9

Table 6: Coding with method LLRUN

Instead, we propose another ‘batching’ step. Rather than use one set of prefix codes for each bitvector, we build several different prefix codes and choose an appropriate one for use with each bitvector. The ‘per bitvector’ cost then becomes small, just a selector variable. The selection is based on p_w , the number of 1-bits; and the batching we propose is logarithmic.

For the bitvector for word w with frequency p_w we use the $\lfloor \log p_w \rfloor$ ’th prefix code; and so on a collection of N documents we need at most $\lfloor \log N \rfloor$ different prefix codes, and each bitvector must be prefixed by a selector in the range $0.. \lfloor \log N \rfloor$ bits. Table 6 shows the compression resulting from this parameterised version of LLRUN, and the overhead cost when the selector values are Huffman coded. There is a small but clear improvement over LLRUN.

7 Exact Models

It is also interesting to consider the performance of exact models, based upon the actual distribution of gap values. We can again classify these into global models, which treat the bitmap as a whole; and local models, which treat each bitvector independently.

A Huffman code [11] based upon gap frequency is an obvious possibility. However we need to be a little cautious. Since there are potentially N different gap values, the description of the code itself might consume a great deal of space. For example, the simplest description would be a list of N integers recording the length in bits of the corresponding codeword. If codewords are to be up to $\lfloor \log N \rfloor$ bits long then $N \cdot \lfloor \log \log N \rfloor$ bits are required. On *Comact* this might cost 160 Kbyte. While this may be practical for a global code, it is impossible for a local model. Other more compact descriptions might be appropriate when the Huffman code is itself sparse over the range (and indeed, this is what we have used), but for bitvectors with less than a few thousand 1-bits the cost of describing the code will almost certainly outweigh the savings to be had from using the code.

Instead of using n different codes the logarithmic batching described above can be used, resulting in $\lfloor \log N \rfloor$ different codes. Table 7 shows the compression achieved with a single global Huffman code, and with a batched variant, where the code was selected according to $\lfloor \log p_w \rfloor$. Again there is a small, but definite, improvement in the compression performance.

8 Comparison with Other Methods

There has been a great deal of previous work on bitmap compression, and many methods proposed [1, 2, 3, 4, 5, 8, 12, 17, 19, 21]. It is interesting to compare the compression possible with the parameterised models with those previous methods.

	Compression %		
	<i>Manuals</i>	<i>GNUbib</i>	<i>Comact</i>
V_T (geometric mean)	46.6	31.1	26.0
γ for N/b values	4.8	1.4	0.4
Total	51.4	32.6	26.5
V_T (median)	46.3	30.5	25.8
γ for N/b values	4.4	1.3	0.4
Total	50.8	31.8	26.2
V_T (optimal)	46.2	31.0	26.2
γ for b indices	1.5	0.4	0.1
Total	47.6	31.4	26.3

Table 5: Coding with V_T

Although requiring more time for encoding than γ or δ , decoding during query processing is just as fast. During encoding each bitvector will need to be processed two or more times, but the bitmap as a whole only needs to be processed once, since each bitvector is compressed entirely independently of other bitvectors, and so, in a sense, the methods are still one-pass. By way of contrast, the Huffman coded models described in Sections 6 and 7 need to make two passes over the entire bitmap, markedly increasing the resources required during creation of the index.

Given that one parameter leads to improved compression, it is natural to ask whether a second parameter allows further gains. In the same framework, we have also experimented with a ‘two-parameter’ vector V_2 :

$$V_2 = (b, ab, a^2b, \dots, a^{i-1}b, \dots),$$

where the growth rate in the widths of the intervals is also controlled. For example, when $a = 1$, $V_2 = V_G$ and the geometric distribution will be catered for.

To date, however, we have been unable to obtain consistently improved compression from this technique. The cost of specifying the value a for each bitvector is not compensated for by shorter codes.

6 Parameterising the Prefix

Fraenkel and Klein [8] have also considered the problem of bitmap compression. Their method LLRUN uses the same vector V as the γ code, but instead of coding the prefix k in unary uses a Huffman code based upon the observed frequencies of k over all runlengths in the bitmap. In a sense, the method is Huffman coding on buckets of numbers rather than single numbers, in recognition of the fact that a run length of (say) 93 is probably just as likely as runlengths of 92 or 94.

Despite the fact that this method is global, it gives good compression. Table 6 shows the compression obtained by LLRUN. The overhead cost of the Huffman code is insignificant— $\lceil \log N \rceil$ codeword lengths suffice, each requiring $\lceil \log \log N \rceil$ bits.

The obvious option of using a different Huffman code for *each* bitvector leads to a dramatic growth in the amount of model information that must be stored, and results in compression degradation rather than compression improvement.

	Compression %		
	<i>Manuals</i>	<i>GNUbib</i>	<i>Comact</i>
V_G (globally)	77.6	78.2	65.8
V_G (locally)	48.8	41.1	33.8
γ for p_w values	2.3	0.7	0.2
Total	51.1	41.8	33.9

Table 4: Coding using V_G

equivalent compression scheme using a Bernoulli model and arithmetic coding [21], where the frequency of each word was also used to control the arithmetic coding of the text of the retrieval system.

Of course, in practice the 1-bits are *not* uniformly and randomly distributed. The natural ordering of the documents means that most of the words will be relatively frequent over small sections of the collection, and relatively *infrequent* in the remainder. No advantage of this is taken by the simple code described above.

Teuhola [19] described a similar encoding that exploits this skewness in the distribution of run lengths. His ‘Exp-Gol’ code, a generalisation of γ , corresponds to the coding vector

$$V_T = (b, 2b, 4b, \dots, 2^{i-1}b, \dots),$$

where it was suggested that b be chosen as some small power of two. In fact there is no reason to restrict b to be a power of two; and, in general, nor is there any reason to choose b to be small. We experimented with various mechanisms for choosing the best value of b , and compared several ad-hoc heuristics with the cpu-intensive technique of finding by trial and error the best value of b for each bitvector. Table 5 shows the compression that was obtained. The best of the ad-hoc methods for choosing b was to take the median of the gap values. This choice matches the observed effect—when the distribution is very uniform, the median will be close to the arithmetic mean and the code will be closest to that generated by V_G ; and when the distribution is very skew b becomes correspondingly less. Taking b to be the geometric mean of the run lengths within the bitvector was also a useful heuristic. Both of these techniques resulted in performance only a few percent worse than the ‘optimal’ technique of scanning a wide range of b values, and choosing the value that resulted in the best compression.

The parameter b must again be prefixed to each bitvector. When the median or the geometric mean are being used this is best effected by using γ to encode N/b , since N/b will usually be smaller than b itself. For the ‘optimal’ value of b , γ was used not to encode the value N/b , but instead to encode the index of the minimising value in the decreasing sequence

$$\left(\frac{N}{2}, \frac{N}{2\sqrt{2}}, \frac{N}{4}, \frac{N}{4\sqrt{2}}, \dots, 1\right)$$

which was the set of b values tested for each bitvector. This is why the cost of storing b is smaller in this third case.

As can be seen from Table 4, use of V_T yields improved compression, even if b is simply taken to be the median of the gaps within each bitvector. Using the median will be faster during encoding than finding the minimising b value; but decoding speed during query processing will be similar for all of these methods.

of blocks of zero bits rather than the number of individual zero bits as we do, following their runlength they encode the position in the next block of any 1-bits. We believe that the speed degradation caused by coding runlengths in terms of bits rather than blocks will be small, and that improved compression will result.

5 Parameterising the Suffix

To make the compression sensitive to the characteristics of particular words, we wish to use a *local* model, controlled by some parameters associated with *each* bitvector. One approach is to allow V to be different for each bitvector—parameterising in some way the suffix part of the general code. Some representation of V must also be stored with each compressed bitvector, a non-trivial overhead. Nevertheless, in some cases the compression gains compensate for this.

Another way in which the encoding might be parameterised is to use the same V for each bitvector, but to allow the codes used to represent k to depend on some characteristic of the vector being coded. In Section 6 we explore this approach, using as a starting point the LLRUN method of Fraenkel and Klein [8]. Again, there is some overhead, both to store the description of the prefix codes, and, in each bitvector, to indicate which prefix code should be used.

Finally, we investigate in Section 7 the application of the same techniques to a general Huffman code on the run lengths.

We now consider how the vector controlling the suffix coding might be varied, and first describe two simple modifications to the Elias codes detailed in the previous section. Both γ and δ give good compression, since most of the inter-word gaps are relatively small compared to N . However they are *insensitive* to word frequency within each bitvector, and so offer some hope of improved compression.

An obvious parameter that might be used in the encoding of the bitvector for word w is p_w , the number of 1-bits present. The average inter-word gap is given by N/p_w ; setting $b = 0.69N/p_w$ and taking

$$V_G = (b, b, b, \dots)$$

gives an encoding [9, 10, 14] in which each bitvector is represented in at most

$$p_w \cdot \left(\log \frac{N}{p_w} + 2 \right)$$

bits, which, on a per pointer basis, is just a small additive constant from the minimum number of bits possible if it is assumed that the 1-bits are uniformly random in the bitvector [9, 15]. Compression values obtained using this encoding are listed in Table 4. For comparison, the table also shows the compression that would be achieved if this same method was used globally on the entire bitmap rather than locally on each bitvector. The improvement achieved by changing from a global model to a set of local models is dramatic.

For each word w the frequency of the word must be prefixed to the coded vector, so that the decoder can know the parameter controlling the coding of each bitvector. Since most words have a low frequency, γ and δ are appropriate for encoding this parameter, with short codes for small integers. Table 4 also shows the cost of using γ to do this. The overhead is much less than the savings attained by compressing each of the bitvectors individually. It is also worth noting that in some cases this parameter will not be an overhead at all. For example, the frequency of a term might be used in document ranking, or for controlling the compression of the text of the collection. Witten, Bell and Nevill have recently an

x	γ	δ
1	1,	1,
2	01,0	010,0
3	01,1	010,1
4	001,00	011,00
5	001,01	011,01
6	001,10	011,10
7	001,11	011,11
8	0001,000	00100,000

Table 2: Examples of codes

Let V be a (possibly infinite) vector of positive integers v_i , where $\sum v_i > N$. To code integer $x > 1$ relative to vector V we find k such that

$$\sum_{j=1}^{k-1} v_j < x \leq \sum_{j=1}^k v_j$$

and code k in some appropriate representation followed by the difference

$$d = x - \sum_{j=1}^{k-1} v_j - 1$$

in binary, using either $\lfloor \log v_k \rfloor$ bits if $d < 2^{\lfloor \log v_k \rfloor} - v_k$ or in $\lceil \log v_k \rceil$ bits if $d \geq 2^{\lceil \log v_k \rceil} - v_k$. For example, γ is an encoding relative to the vector

$$V_\gamma = (1, 2, 4, 8, \dots, 2^{i-1}, \dots),$$

with k coded in unary. Because V_γ is infinite, any positive value x can be represented. Another example (with a finite V) is a slightly modified binary encoding, in which $V = (N)$ ensures that k is always 1 and need not be coded at all, and $d = x - 1$ is coded in either $\lfloor \log N \rfloor$ or $\lceil \log N \rceil$ bits.

This approach was first explored by Fraenkel and Klein [8]. However they did not consider the possibility of a different V being used for each word, and so the model in their scheme (and in the Elias codes above) is *global*—applying to the bitmap as a whole rather than to the bitvectors individually. This has the advantage of simplicity, but means that for many of the bitvectors in the bitmap an inappropriate encoding will be used, to the detriment of compression performance. Fraenkel and Klein also chose to encode the number

	Compression %		
	<i>Manuals</i>	<i>GNUbib</i>	<i>Comact</i>
γ	57.6	39.5	28.3
δ	55.0	34.6	26.8

Table 3: Compression factors for γ and δ

The transformation is reversible; and given an ordered list of runlengths it is straightforward to recompute either the inverted file entry or the original bitvector. Since the inverted file entry is always accessed sequentially, there is no loss of generality in considering only a list of runlengths.

Run length encoding does not of itself result in any compression, since the maximum runlength might still be N , and each entry in a binary coded inverted file must still be allocated $\lceil \log N \rceil$ bits. What run length coding does emphasise is any regularity in the runlengths, exposing patterns that can be exploited for compression purposes. For example, a common word can be expected to have a mean runlength that is less than the mean runlength of a less frequent word.

Thus, the mean runlength, or equivalently, the frequency of a word, might be used in some way to *parameterise* the encoding of each bitvector, and leads, as we shall see in Section 5, to appreciable compression gains. We might also consider allowing the coding of each bitvector to be controlled by two or more parameters. In the limit, for asymptotically large document collections, the runlengths within each bitvector are best modelled by using the exact gap frequencies to drive either a Huffman coder [11] or an arithmetic coder [22]. However this does not result in the most economical representation of the bitmap for practical sized collections, since the parameters that control the coding must also be stored, and can constitute a significant overhead.

Before describing the various ways in which the runlengths might be modelled it is also useful to consider some *parameterless* models. Surprisingly, these can also give rise to significant compression.

4 Parameterless Models

To store the list of runlengths we need an encoding of the positive integers. Elias [7] described a number of such encodings. Each of the encodings has the property that small integers are allocated short codes, while larger integers are allocated longer codes. For example, his γ code represents x as $\lfloor \log x \rfloor$ zero bits, followed by the minimal binary representation of x (which must start with a one) in $1 + \lfloor \log x \rfloor$ bits. An alternative way of describing the same code is to say that $\lfloor \log x \rfloor$ is coded in unary, then $x - 2^{\lfloor \log x \rfloor}$ is coded in binary. For reasons that will become clearer below, we prefer this latter definition. The γ code requires $1 + 2\lfloor \log x \rfloor$ bits to code integer x .

Some values of γ are shown in Table 2. For ease of understanding, a comma has been placed between the prefix and suffix parts; this comma does not, of course, appear in the actual code.

Elias' δ code can similarly be thought of as using γ to code the value $\lfloor \log x \rfloor$, followed by the same suffix. Integer x will be represented in $1 + 2\lfloor \log(1 + \log x) \rfloor + \lfloor \log x \rfloor$ bits. The δ code is longer than the γ code for a few values of x smaller than 15, but thereafter δ is never worse.

Table 3 shows the compression ratios that resulted when γ and δ were used to code the runlength representations of the three sample document collections. Both give rise to significant compression, saving nearly half of the inverted file space for *Manuals* and more than two thirds of the space required by the inverted file for *Comact*. The δ code outperforms γ by a few percent. Both are easy to implement and very fast in both encoding and decoding, and both require just one pass over the input bitmap to generate a compressed representation.

The γ encoding can also be thought of as being one example of a more general coding paradigm [8], described as follows.

	Collection Name		
	<i>Manuals</i>	<i>GNUbib</i>	<i>Comact</i>
Text Size (Mbyte)	5.15	14.12	132.11
Documents (N)	2,496	64,344	261,829
Distinct Words (n)	27,554	70,866	68,074
Word Occurrences	958,744	2,575,411	23,100,786
Word Pointers Stored (p)	317,431	2,296,210	14,219,077
Average Words per Document	384	40	88
Bitmap Size (Mbyte)	8.20	543.57	2124.76
Inverted File Size (Mbyte)	0.45	4.38	30.51

Table 1: Sizes of document collections

Table 1 also shows the space that would be required by a raw bitmap ($N \cdot n$ bits) and by an uncompressed inverted file ($p \cdot \lceil \log N \rceil$ bits). In this paper we will quantify compression effectiveness by giving the fraction of this latter quantity required by the encoding. For example, an algorithm that stores the bitmap for *Comact* in 20 Mbyte will be described as having a compression of $20/30.51 = 65.6\%$. Compression factors greater than 100% indicate expansion relative to the binary coded inverted file.

In compiling Table 1 we indexed all words, where a ‘word’ was defined to be any non-empty string of up to 15 alphanumeric characters, provided that the maximum number of numeric characters was not greater than four. The latter constraint was added after initial experimentation showed that the page numbers of *Comact* ran in an unbroken run from 1 to 261,829 and produced an unpleasantly large number of ‘words’. Strings of alphanumerics longer than 15 characters, or containing more than 4 numeric characters, were broken, with the offending character marking the start of a new word. We did not remove any *stop words*, and indexed even single character and single digit ‘words’. Although the words removed as stop words are typically very frequent, and their inverted file entries correspondingly long, the savings that would accrue from removing them are relatively small, since the compression techniques we describe will also handle dense bitvectors efficiently. Moreover, there will only be a few such words in any document collection; and so for simplicity and generality we indexed all words.

3 Run Length Encoding

All of the compression methods we discuss are based upon the well known technique of *run length coding*. Rather than storing, in each inverted file entry, the absolute position of each 1-bit in the corresponding bitvector, we store the differences. For example, the bitvector

00011000101100001...

normally regarded as corresponding to the inverted file entry

4, 5, 9, 11, 12, 17, ...

would be stored as

4, 1, 4, 2, 1, 5, ...

1 Introduction

Full-text retrieval systems are used for storing and accessing document collections such as newspaper archives, office automation systems, and on-line help facilities. Two common methods of providing the index needed for efficient keyword-based query processing on these systems are *bitmaps* and *inverted files*. A bitmap contains, for each word (or *term*) w appearing in the document collection, a *bitvector* in which the d 'th bit is 'one' if and only if word w appears in document d . If the collection contains N documents and n distinct terms the bitmap will occupy $N \cdot n$ bits, and might be an order of magnitude larger than the text it represents. For example, one of our experimental document collections occupies 132 Mbyte; has $N = 261,000$ documents; has $n = 68,000$ indexed terms; and contains $p = 14,000,000$ term-document pairs. The raw bitmap for this collection would occupy 2.1 Gbyte, more than 15 times the volume of the text it represents.

An inverted file is a more economical method of storing the bitmap; it contains one list for each word w , with each list containing the document numbers d that contain w . Because the bitmap is typically sparse, an inverted file will usually occupy less space than the bitmap it represents, and is just as easy to manipulate. For example, to find documents containing both of two terms we must (in the same example) **AND** two 32 Kbyte bitvectors if the index is stored as a raw bitmap. With an inverted file this is achieved by taking the intersection of the two lists of occurrences, where each list will contain perhaps a few hundred or a few thousand integers. Taking the same example collection, each inverted file pointer would require¹ $\lceil \log N \rceil = 18$ bits, and the inverted file would occupy 30 Mbyte, one seventieth the space of the bitmap.

Here we consider how each inverted file entry might be further compressed. Our hypothesis is that additional compression can be achieved if each bitvector (that is, each inverted file entry) is described and coded in terms of some relatively small number of parameters, normally just one. We describe several such schemes as modifications of existing global (unparameterised) schemes for bitmap compression, and detail the savings produced by each of these methods. The best of these methods is capable of storing the same example bitmap in about 8 Mbyte, corresponding to less than 5 bits per word pointer.

We have previously described a compression regime for use with the main text of a full-text retrieval system [15]. In conjunction with the techniques presented here for inverted file storage, a complete compressed representation of this document collection requires just 43 Mbyte, less than one third of the space required by the initial text alone. Moreover, the compression techniques are sufficiently fast in decoding that they have little or no effect on run-time retrieval performance.

2 Document Collections

We have tested our bitmap compression techniques on three collections of documents. Table 1 shows the sizes of these collections. The collection *Manuals* was a collection of Unix manual pages (`/usr/man/man[1-8]/*` on a Sun SPARCstation), including embedded formatting commands. Collection *GNUbib* [18] stores 64,000 citations to journal articles, technical reports, conference papers, and books, all stored in 'refer' format [13, 20]. Each citation was taken to be a 'document' for indexing purposes. The third document collection is the one we have already mentioned; it is a collection of 261,829 pages of legal text storing the complete Commonwealth Acts of Australia, from federation (in 1901) to 1990.

¹All logarithms are binary.

Technical Report 92/1

Parameterised Compression for Sparse Bitmaps

Alistair Moffat *Justin Zobel*

`alistair@cs.mu.oz.au`

Department of Computer Science
The University of Melbourne
Parkville 3052, Australia

`jz@kbs.citri.edu.au`

Department of Computer Science
Royal Melbourne Institute of Technology
Melbourne 3001, Australia

January 1992

Abstract: Full-text retrieval systems typically use either a bitmap or an inverted file to identify which documents contain which words, so that the documents containing any combination of words can be quickly located. Bitmaps of word occurrences are large, but are usually sparse, and thus are amenable to a variety of compression techniques. Here we consider techniques in which the encoding of each bitvector within the bitmap is parameterised, so that a different code can be used for each bitvector. Our experimental results show that the new methods yield better compression than previous techniques.

Categories and Subject Descriptors: E.4 [Coding and Information Theory]: *Data compaction and compression*; H.3.2 [Information Storage]: *File organisation*.

Keywords: Full-text retrieval, data compression, document database, Huffman coding, geometric distribution, inverted file.