Crossroads
The ACM Student Magazine

Prefix Compression of Sparse Binary Strings

by ***David Salomon***

Note from ACM Crossroads: Due to errors in the layout process for printing on paper, the version of this article in the printed magazine contained several errors (mostly related to superscripts). This HTML version is the accurate version. Please refer to this HTML version instead of the printed version and accept our apologies for any inconvenience.

Introduction

A **sparse binary string** is a sequence of bits in which the vast majority of elements are zero. A typical sparse binary string, of a million bits, may have just 100 nonzero bits. Sparse strings can be compressed efficiently. Several simple, efficient compression methods are described in Fraenkel and Klein [**2**]. The method described here uses the concept of *prefix*, proposed by Anedda and Felician [**1**].

Before getting to the details of the method, we should consider the occurrences of sparse binary strings in practice. For instance:

- Drawing - Imagine a drawing, technical or artistic, done with a black pen on white paper. Many non-complex drawings remain mostly white. When such a drawing is digitized, most of the resulting pixels are white, and the percentage of black pixels is small. The resulting bitmap is an example of a sparse string.
- Databases - Imagine a large database of text documents. A bitmap index for such a database is a set of bitstrings (or bitvectors) that simplifies the identification of a set of documents, containing a given word. To implement a bitmap, we first prepare a list of all distinct words `wj`, in all documents. Suppose there are `W` such words. The next step is to search each document `di` and prepare a bit-string `Di` of length `W`, containing a `1` in

position `j` if word `wj` appears in document `di`. The bitmap is the set of all bit-strings. If the documents in the database tend to contain only a small fraction of the set of words, then the bit-strings will be sparse.

Details of the Method

The principle of the prefix compression method is to

1. assign an address to each nonzero bit in the sparse string
2. divide each address into a prefix and a suffix
3. select all the one-bits whose addresses have the same prefix and write them on the compressed file by writing the common prefix, followed by all the different suffixes

Compression can be achieved if the addresses of many one-bits have the same prefix. In order for several one-bits to have the same prefix, their addresses should be close. However, in a long, sparse string, the one-bits may be located far apart. Prefix compression tries to bring together one-bits that are separated in the string by breaking up the string into equal-size segments. These segments are then placed one below the other, effectively creating a sparse *matrix* of bits. It is easy to see that one-bits, widely separated in an original string, may be closer together in this matrix. As an example consider a binary string of 2^{20} bits (a megabit). The maximum distance between bits in the original string is about a million, but when the string is rearranged as a matrix of dimensions 2^{10}x2^{10} = 1024x1024, the maximum distance between bits is only about a thousand.

Our challenge is to assign addresses to matrix elements such that

1. the address of a matrix element is a single number
2. spatially related elements are assigned similar addresses

The usual way of referring to matrix elements is by row and column. We can create a one-number address by concatenating the row and column numbers of an element, but this is unsatisfactory. For instance, consider the two matrix elements at positions (1,400) and (2,400). They are certainly close neighbors but their addresses would be 1400 and 2400, not very close!

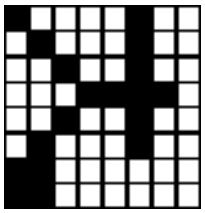
We therefore use a different method. Think of the matrix as a digital image where each bit becomes a pixel (white for a zero-bit and black for a one-bit) and we require that this image be of size 2^nx2^n for some non-negative integer n. This normally requires extending the original string with zero bits until its size becomes an even power of two (2^{2n}). The original size of the string should therefore be written to the compressed file for the use of the decompressor. If the string is sparse, the corresponding image has few black pixels. Those pixels are assigned

addresses using the concept of a *quadtree.*

To understand how this is done, let's assume that an image of size $2^n x 2^n$ is given. We divide it into four quadrants and label them (0_2 1_3). Notice that the digits 0--3 are 2-bit numbers. Each quadrant is subsequently divided into four subquadrants labeled in the same way. Each subquadrant thus gets a four-bit (two-digit) label. This process continues recursively and as the subquadrants get smaller, their labels grow. When this numbering scheme is carried down to individual pixels, the number of a pixel turns out to be 2n bits long. Figure 1 shows the pixel numbering in a $2^3 x 2^3$ = 8x8 image and also a simple image consisting of 18 black pixels. Each pixel number is six bits (three digits) long and they range from 000 to 333. The original string being used to create this image is

```
10000100 01000100 00100100 00011110 00100100 01000100 11000000 11000000
```

(where some of the six trailing zeros may have been added to make the string size an even power of two).



000	001	010	011	100	101	110	111
002	003	012	013	102	103	112	113
020	021	030	031	120	121	130	131
022	023	032	033	122	123	132	133
200	201	210	211	300	301	310	311
202	203	212	213	302	303	312	313
220	221	230	231	320	321	330	331
222	223	232	233	322	323	332	333

Figure 1: Example of Prefix Compression.

The first step is to use quadtree methods to calculate the three-digit labels of the 18 black pixels. They are 000, 101, 003, 103, 030, 121, 033, 122, 123, 132, 210, 301, 203, 303, 220, 221, 222, and 223.

The next step is to select a *prefix* value. For our example we select P = 2, which is justified below. The code of a pixel is now divided into P prefix digits followed by 3-P suffix digits. The last step goes over the sequence of black pixels and selects all the pixels with the same prefix. The first prefix is 00, so all the pixels whose labels start with 00 are selected (i.e., 000 and 003). They are removed from the original sequence and are compressed by writing the token 00|1||3 on the output stream. The first part of this token is a prefix (00), the second part is a count (1), and the rest are the suffixes of the two pixels having prefix 00. Notice that a count of one implies two pixels. To conserve space, the count is always one less than the number of pixels being counted. Sixteen pixels now remain in the original sequence and the first of them has prefix 10. The two pixels with this prefix, namely 101 and 103, are removed and compressed by writing the token 10|1|1|3 on the output stream. This continues until the original sequence becomes empty. The final result is the nine-token string

```
00|1|0|3 10|1|1|3 03|1|0|3 12|2|1|2|3  13|0|2 21|0|0 30|1|1|3 20|0|3  22|3|
                                    0|1|2|3
```

or in binary

```
0000010011 0100010111 0011010011 011010011011 01110010 10010000 1100010111
                          10000011 10101100011011
```

(without the spaces). Such a string can be decoded uniquely since each token starts with a two-digit prefix, followed by a one-digit count c, followed by c+1 one-digit suffixes. Preceding the tokens, the compressed file should contain, in raw form, the value of n, the original length of the sparse string, and the value of P used in the compression. The decompressor reads the values of n and P. These two numbers are all it needs to read and decode all the tokens unambiguously.

Note that adjacent pixels can have different prefixes. This occurs if they are located in different quadrants or subquadrants. Pixels 123 and 301, for example, are adjacent in Figure 1 but have different prefixes.

Notice that our example results in expansion because our binary string is short and therefore not sparse. A sparse string has to be at least tens of thousands of bits long.

To determine P, we first figure out the maximum number of suffixes per prefix as a function of both n and P. This yields the maximum size T(n,P) of a token. We then find out the maximum number R of different prefixes, also as a function of n and P. The product of T and R is the total size S(n,P) of all the tokens; we select the value of P that minimizes S(n,P). Finally, we improve the results by considering the expected, rather than the maximum, number of prefixes

and suffixes in a sparse string.

In general, the prefix is P digits (or 2P bits) long and the count and each suffix are n-P digits each. Since each digit can take on four different values, the maximum number of suffixes in a token is therefore 4^{n-P}. The maximum size of a token is thus $P+(n-P)+4^{n-P}(n-P)$ digits. Since each token corresponds to a different prefix, the maximum number of tokens is 4^P. The entire compressed string thus occupies at most

$$4^P[P+(n-P)+4^{n-P}(n-P)] = n\cdot 4^P+4^n(n-P)$$

digits. To find the optimum value of P we differentiate the expression above with respect to P

$$d\,dP\ [n\cdot 4^P+4^n(n-P)] = n\cdot 4^P\ln 4-4^n,$$

and set the derivative to zero. The solution is

$$4^P = 4^n\, n\cdot\ln 4 \quad \text{or } P = \log_4 4^n \backslash n\cdot\ln 4 = 1 \backslash 2\ \log_2 4^n\, n\cdot\ln 4\ .$$

For n = 3 this yields

$$P = 1 \backslash 2\ \log_2 4^3 \backslash 3x1.386 = \log_2 15.388 \backslash 2 = 3.944\backslash 2 = 1.97.$$

This is why P = 2 was selected in our example. A practical compression program should contain a table with P values precalculated for all expected values of n. Table 1 shows such values for n = 1,2,*,12.

Table 1: Dependence of P on n.

n:	1	2	3	4	5	6	7	8	9	10	11	12
P:	0.76	1.26	1.97	2.76	3.60	4.47	5.36	6.26	7.18	8.10	9.03	9.97

```
Clear[t]; t=Log[4]; (* natural log *)
Table[{n,N[0.5 Log[2,4^n/(n t)],3]}, {n,1,12}]//TableForm
```

Mathematica Code for Table 1.

This method for calculating the optimum value of P is based on the worst case. It uses the maximum number of suffixes in a token, but many tokens have just a few suffixes. It also uses the maximum number of prefixes, but in a sparse string many prefixes may not occur. Experiments indicate that for large sparse strings (corresponding to n values of 9-12), better compression is obtained if the P value selected is one less than the value suggested by Table 1. However, for short sparse strings, the values of Table 1 are optimum. Selecting, for example, P

= 1 instead of P = 2 for our short example (with n = 3), results in the four tokens

```
0|3|00|03|30|33   1|5|01|03|21|22|23|32   2|5|10|03|20|21|22|23   3|1|01|03
```

that require a total of 96 bits, more than the 90 bits required for the choice P = 2.

In our example the `count` field can go up to three, which means that an output token, whose format is `prefix|count|suffixes`, can compress at most four pixels. A better choice may be to encode the `count` field so its length can vary. Even a simple unary code might produce good results. The unary code of a non-negative integer n is defined as n-1 ones followed by a single zero or, alternatively, as n-1 zeros followed by a single one. A better choice may be a Huffman code where small counts are assigned short codes. Such a code may be calculated based on distribution of values for the `count` field determined by several ``training'' sparse strings.

Encoding the `prefix` and `suffix` fields does not produce the same benefits because all prefixes have the same probability of occurrence. In our example the prefixes are four bits long and all 16 possible prefixes have the same probability since a pixel may be located anywhere in the image. A Huffman code calculated for 16 equally-probable symbols has an average size of four bits per symbol, so nothing would be gained. The same is true for suffixes.

Results

Table 2 lists test results from six sparse strings. The numbers listed are the compression factors (the size of the original string divided by the size of the compressed file). The factors obtained are large, as expected from a sparse string. The following, intuitive compression method gives order of magnitude estimates for the compression factors expected for sparse strings. Given a 1Mbit sparse string, we can compress it by writing the positions of the one-bits on the compressed file. Since the original string is 2^{20} bits long, a bit position is a 20-bit number. If there are 100 one-bits, the compressed file will be 100x20 = 2000 bits long, yielding a compression factor of $2^{20}/2000 \approx 524$. With 1000 one-bits, the factor should be about 52. For our more complicated method to be of practical use, it should produce much better results.

The six test strings used are denoted by S_1 through S_6 and are 1Mbit (= 2^{20} = 1,048,576) long each. String S_1 has 100 bits of 1, for a density of $9.54 \cdot 10^{-5}$. The 100 bits are uniformly distributed over the entire string. String S_2 is similar except that the 100 one-bits are uniformly distributed in the central half of the string. Strings S_3 and S_4 are similar to S_1 and S_2, respectively, except that they have 1,000 bits of 1 (i.e., ten times the density) and in S_4 the

1,000 bits are uniformly distributed in the central half of the string. Strings S_5 and S_6 are similar, with 10,000 bits of 1 (i.e., a density of about 1%). In S_6, those 10,000 bits are distributed as in S_2 and S_4.

The method described here (denoted by M1 in Table 2) was compared to three of the compression methods described in Fraenkel and Klein [**2**]. These methods are: ORing bits (M2), variable-size codes (M3), and variable-size codes for base 2 (M4). The results are consistent and show that M1 is slightly better than M2 and M3, and slightly outperforms M4.

Table 2: Performance Comparison of Four Methods.

	S_1	S_2	S_3	S_4	S_5	S_6
M1	5280	2335	550	320	31	25
M2	4277	2305	410	277	24	10
M3	4360	2449	456	208	29	12
M4	5330	2450	548	322	34	29

Conclusion

The prefix method described here is easy to implement and achieves respectable results. It is one of just a handful of methods specifically developed for the compression of sparse strings, and thus can be extremely useful in an application where many such strings have to be compressed and decompressed. The idea of encoding the *count* field seems attractive and may considerably improve the performance of the method.

References

1

Anedda, C., and Felician L. P-Compressed Quadtrees for Image Storing. *The Computer Journal*, 31, 4 (1988), 353-357.

2

Fraenkel, A. S., and Klein S. T. Novel Compression of Sparse Bit-Strings-Preliminary Report, in A. Apostolico and Z. Galil, eds., *Combinatorial Algorithms on Words*, Vol. 12, NATO ASI Series F:169-183, New York, Springer-Verlag, 1985.

Biography

David Salomon is with the computer science department at California State University,

Northridge. His interests are in the areas of computer graphics and data compression. He can be reached at **david.salomon@csun.edu**