A simple data compression scheme for binary images of bacteria compared with commonly used image data compression schemes

Wilkinson, M.H.F.

Published in:
Computer Methods and Programs in Biomedicine

Publication date:
1994

ELSEVIER

Computer Methods and Programs in Biomedicine 42 (1994) 255–262

computer methods and programs in biomedicine

A simple data compression scheme for binary images of bacteria compared with commonly used image data compression schemes

M.H.F. Wilkinson

Laboratory for Medical Microbiology, University of Groningen, Oostersingel 59, 9713 EZ Groningen, The Netherlands

(Received 24 November 1993; accepted 10 January 1994)

Abstract

A run length code compression scheme of extreme simplicity, used for image storage in an automated bacterial morphometry system, is compared with more common compression schemes, such as are used in the tag image file format. These schemes are Lempel-Ziv and Welch (LZW), Macintosh Packbits, and CCITT Group 3 Facsimile 1-dimensional modified Huffman run length code. In a set of 25 images consisting of full microscopic fields of view of bacterial slides, the method gave a 10.3-fold compression: 1.074 times better than LZW. In a second set of images of single areas of interest within each field of view, compression ratios of over 600 were obtained, 12.8 times that of LZW. The drawback of the system is its bad worst case performance. The method could be used in any application requiring storage of binary images of relatively small objects with fairly large spaces in between.

Key words: Image processing; Run length code; Data compression; Binary images

1. Introduction

In our laboratory much work has been done in the field of bacterial morphometry [1–3]. For this purpose a microbiological image processing system has been developed [4]. Any image processing system is faced with a data storage problem. The data bulk acquired, even on systems with modest spatial and grey-level resolution, usually requires some form of data compression. In our case, the images of interest are 512 × 384 pixels in size, and require one bit per pixel storage, or 24 kilobytes (kB) in total storage space. Though these single images are not very large, several hundreds may be processed in a single day, amounting to some 10 megabytes (MB) of storage. Although multiple grey-level images, let alone full colour images, require an order of magnitude more space, it is obviously useful to reduce the size of the data bulk, even in the case of these binary images.

Originally, our system used 'home grown' file

* Corresponding author.

SSDI 0169-2607(94)01479-Y

formats, and a simple run length code (RLC) scheme, which could easily be implemented in Pascal. As our image processing now includes multiple grey-level processing, and full-colour processing is under development, our image file formats have had to be reviewed. As a result, we have compared a number of commonly used compression schemes with our own early effort, to evaluate its efficiency. Code size, speed, average compression in a set of 25 randomly chosen bacterial images, and a set of 100 images of single areas of interest (bacteria), and worst case performance were compared.

2. Computing methods and theory

2.1. TIFF compression schemes

Many different methods of data compression exist, all of which rely on redundancies in the information in the image. One of the simplest forms of image data compression is run length coding (RLC), which relies on the fact that many images contain large areas of constant grey-level. Binary images are especially amenable for this form of data compression. Other schemes, such as Lempel-Ziv and Welch (LZW) [5], rely on string translation tables, converting repetitive strings of bytes into short code words. These are usually more complicated, but also far more flexible, as they adapt their string table to suit the needs of a particular type of image, or even part of an image.

The tag image file format (TIFF) has three built-in compression schemes for general image storage: LZW, Macintosh Packbits (MPB), and CCITT Group 3 Facsimile 1-dimensional modified Huffman RLC (CCITT) [6]. The first is a highly adaptable string table compression, useful in practically any type of image. Though the words used in this code do vary in size from 9 to 12 bits in the TIFF implementation, they do so in a predictable manner. This means that some bit manipulation is necessary, since the word and byte boundaries do not coincide. However, implementation is straightforward in C, and only slightly less so in Pascal. LZW code is relatively fast, and has a relatively good worst case performance (1.4-fold expansion in theory). The other two schemes are RLC schemes. MPB is the simplest. Each line in the image is scanned for repeating bytes. If such a string, of length n, is found, and $n \leq 128$ bytes, $n - 1$ is written to the file, followed by the byte to repeat. If $n > 128$, runs of 128 are written to the file using the above scheme, until the remainder is smaller than 128. If a string of n non-repeating bytes is found, and $n \leq 128$, $-n + 1$ is written to the file, followed by n bytes of the string itself. If $n > 128$, repeating runs of 128 are written in the above way, until the remainder is smaller than 128. The obvious advantages of this scheme are: ease of implementation in high level languages, as C or Pascal, and good worst case performance (1.0094-fold expansion). CCITT coding is a more pure form of RLC for binary images. It uses a table of unequal length code words for black and white runs, which was optimized for documents, in which small black runs alternate larger white runs [7,8]. Its worst case performance is not impressive (4.5-fold expansion), and the code is somewhat more complicated, especially for the decompressor, which must read one bit at a time and consider if the bits read so far constitute a valid code word. CCITT requires that a table of valid codes is kept in memory at run time, which makes the code and data rather bulky. CCITT code can be generated using either black or white as background. White is standard in TIFF, but black can be selected optionally. As our images have a black background, CCITT compression was done using black as background.

2.2. Continuation bit RLC

Our own compression scheme arose from the need to have a compression scheme to store the images obtained using the morphometrical package developed in our laboratory [1]. A typical image is shown in Fig. 1. Both for storage and processing efficiency reasons, these images were not stored in memory as a bit-map, but as a linked list of x and y coordinates on which the image changed from black to white or vice versa, when scanning the image from top to bottom and left to right. Such a linked list is shown in Fig. 2. The x and y coordinates of each point can be considered as two unsigned short (2-byte) integers, or as a single unsigned long (4-byte) integer, called the key. The value of the key can be expressed as:

$$key = 65\,536 \cdot y + x \tag{1}$$

Fig. 1. A typical binary image of a full field of view of bacteria, as acquired and segmented by the morphometrical program of our image processing system.

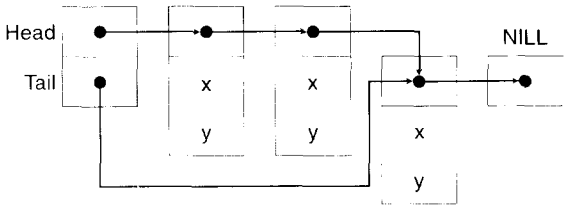

Fig. 2. The linked list structure (imagelist) used to store binary images in memory. All morphological operations in our system are performed on such imagelists.

The boundary points are sorted by key. Note that the order of storage of the x and y coordinates in Fig. 2 corresponds to Intel processor byte order. Using Motorola byte order, x and y would have to be interchanged. Simply storing the keys does provide some compression compared with bit-maps, but more can be achieved. Run lengths can be computed from the linked list structure in a very straightforward way, simply by taking the difference of subsequent key values, but a slight im-

provement can be made by altering the keys used in computing the run lengths:

$$key = imagewidth \cdot y + x \tag{2a}$$

$$\Leftrightarrow runlength = imagewidth \cdot (y_n - y_{n-1}) + (x_n - x_{n-1}) \tag{2b}$$

Storing the run lengths as long integers does not improve the compression in any way. Therefore, a simple, byte oriented, unequal word length code was devised, based on continuation bit code, such as found in textbooks such as Gonzales and Wintz [9]. In continuation bit code, also called B-code, each code word consists of an integer number of 'sub-words' of fixed size. The size of a sub-word in B-N code is $N + 1$ bits: N data bits and 1 continuation bit. This last bit flags if the sub-word belongs to an even or an odd word. In the implementation discussed by Gonzales and Wintz, this is the most significant bit of the sub-word, and only B-1 and B-2 codes are shown, but the principle could apply to any N, and any bit could be chosen as continuation bit. In this case, $N = 7$ was chosen, making the sub-words coincide with bytes. The least significant bit was chosen as continuation bit, as it can easily be tested in ISO Pascal, using the ODD function. An example of three B-7 code words of different length is given in Fig. 3. The data bits are considered to be natural code, so no further arithmetic is needed to convert them to run lengths. The coding scheme compresses the run lengths only by removing leading zeros. As short runs are numerous in our images, and no run has more than 18 data bits, code words are at most 3 bytes in length, but are often shorter. The minimum savings with respect to storing run lengths as long integers is 25%.

Reading a code word is very simple: if the current byte is not a stop byte, the value is divided by 2 to remove the continuation bit, and stored in the run length. The next byte is then read, and if its continuation bit equals that of the previous byte, the run length is shifted left by 7 (multiplication by 128 in Pascal), the byte is divided by 2 and added

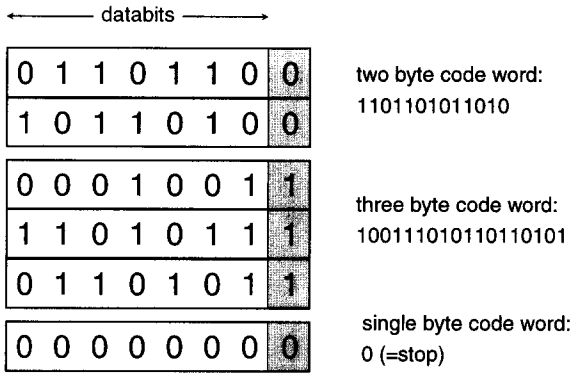

Fig. 3. B-7 code words of various lengths. The continuation bits are shown in shaded boxes, and the corresponding binary natural code values are shown beside.

to the run length. This process is repeated until the continuation bit of the next byte is not equal to that of the preceding ones.

If a stop byte (a run of length zero) is encountered, it is considered to flag the last run, the length of which can be computed from the current position in the image and the image dimensions. The last run of the image is thus not coded for, as its length can be deduced anyway.

Writing a word is equally straightforward, and is performed by the *WriteCodeBytes* procedure. Once a run length has been computed (using Eq. 2b), a single sub-word is created by computing the remainder of division by 128 (run length MOD 128), storing this in the byte, multiplying it by 2, and adding the continuation bit. The run length is then divided by 128, and if the result is not zero, the *WriteCodeBytes* procedure is called with the result. Only when the recursion ends are the bytes written. In pseudo-code:

```
WriteCodeBytes (runlength : LONGINT;
                oddnode : BOOLEAN);
begin
  compute subword from (runlength MOD 128)
    and oddnode;
  divide runlength by 128;
  if runlength <> 0 then
```

```
    WriteCodeBytes (runlength);
  send subword to output;
end
```

Using this algorithm, words of 1–3 bytes in size are written for images of the dimensions used in our image processing work (512 × 384). After all runs but the last have been written, i.e. when the last transition from black to white or vice-versa has been reached, a stop byte is written, consisting of a 0-run length. Colour sychronization is maintained by defining the first run to be black (or background). If the first pixel is white (or foreground) a zero run must be written, but this is not considered as a stop byte. The first run is considered ODD, and the continuation bit is therefore 1. If the first byte (not run) of an image is 0, the image is empty. The full source code of this algorithm can be received from the author. The implementation shown reads and writes a linked list of the type shown in Fig. 2, rather than an image. The implementation tested has been altered slightly, in that it uses the same (buffered) C file I/O functions as our TIFF-package, rather than the ISO Pascal GET and PUT functions. The implementation tested is distinctly speedier than the one shown. For a clearer comparison of methods a version of the algorithm was made which reads and writes an image directly, rather than converting an image to a linked list, and then writing it, or reading a linked list and converting it to an image. Timings for both this implementation and the implementation which reads and writes linked lists were measured.

3. System and methods

3.1. Hardware

Although the images were acquired on a different system, this test was run on an IBM PC-AT compatible, 80386 processor based computer, running at 25 MHz, equipped with 8 MB of memory (RAM), a Bustec 542-C Fast SCSI-II controller and a 512-MB hard disk. The images were displayed on a Trident 8900 super-VGA display board equipped with 1 MB of video RAM, using the 1024 × 768 pixel, 256 colour mode. Image acquisition was performed on a 16-MHz 80286-based PC-AT compatible computer, equipped with 2 MB of RAM, an 85-MB hard disk, and an MVP-AT frame grabber and image processor board (MATROX Electronics Ltd, Dorval, Quebec, Canada). It was connected to a Loral Fairchild CCD 5000/1 camera (Loral Fairchild, Sunnyvale, CA), which was mounted on an Olympus BH2 fluorescence/phase-contrast microscope.

3.2. Software

The image acquisition software was developed in our laboratory and has been described elsewhere [1,4]. After noise removal, the bacterial images are segmented automatically and stored in a binary image file of the proprietary format described previously. These files are normally processed by a second program which analyses the images automatically, separating them into individual bacterial objects, and computing morphological parameters such as area, perimeter, moment of inertia, convex hull area, etc. The individual bacterial objects' shapes are stored in an image file of the same format as full fields of view, for graphical display and future analysis purposes. A separate file containing the morphological data is also created.

A general purpose TIFF-file read and write was implemented in Microsoft C 5.1. It supports all general purpose image compression schemes of the TIFF 5.0 standard, and handles all TIFF classes as defined in the standard (binary, grey-scale, palette and full-colour). Only the binary class is relevant to the current discussion. The compression schemes are implemented in modules separate from the central TIFF-administration module, and the object file sizes and run-time memory allocation are shown in Table 1.

A number of special purpose programs were written to read and write the selected images in different

Table 1
Object file sizes and dynamic memory allocation

Coding Scheme	.OBJ Size (bytes)	Dynamic allocation (bytes)
MPB	2699	512
CCITT	7797	2592
LZW	3429	32 768
RLC–B-7	2494	512

Table 2
Total file sizes and average compression ratios of 25 full field of view images

Coding scheme	Total size (bytes)	Compression ratio
MPB	123 969	4.96
CCITT	64 578	9.51
LZW	64 101	9.58
RLC–B-7	59 722	10.29

formats and measure the time used by the different compression schemes.

3.3. Images

Two categories of images were used. First, 25 randomly chosen, full fields of view of bacterial slides segmented into a binary image were used. The second category was 100 randomly chosen images, each of a single bacterial object isolated from the first set. The latter type of image is practically empty, so compression ratios should be higher than in the former type.

4. Results

The results of the full field of view test are shown in Table 2. The total uncompressed size of the 25 images was 614 400 bytes. The compressed total data sizes in Table 2 do not include the 211-byte header in each TIFF file. Thus they reflect the actual compression performance. The mean compression of B-7 RLC is some 7% better than LZW, and 8% better than CCITT.

In the individual object test the result are somewhat different, as can be seen from Table 3.

Table 3
Total file sizes and average compression ratios for 100 images of a single object of interest

Coding scheme	Total size (bytes)	Compression ratio
MPB	80 794	30.42
CCITT	78 820	31.18
LZW	47 420	51.8
RLC–B-7	3 704	663.5

Table 4
Average compression and decompression times for 25 full field of view images

Coding scheme	Compression time (s)	Decompression time (s)
MPB	2.03	1.93
CCITT	1.18	0.74
LZW	2.15	1.74
RLC–B-7	0.98 (0.17[a])	0.60 (0.44[a])

[a]Times needed for linked list storage and retrieval

CCITT and MPB almost reach their theoretical limit of 32-fold compression for images of 512 pixels wide. LZW also does particularly well with more than 50-fold compression, but RLC–B-7 with over 600-fold compression is much more efficient: 12.8 times as efficient as LZW and 21 times as efficient as CCITT.

The compression and decompression times are given in Table 4. RLC–B-7 proves to be the faster method, although it only beat CCITT (the runner up) by about 20%. LZW and MPB were approximately equal in speed, lagging by some 200% in decompression, and 110% in compression.

5. Discussion

In both test sets the performance of our (almost trivially simple) run length code is rather impressive. Nevertheless, in the full fields of view, the small difference in compression ratio with standard formats probably does not justify the continued use of this proprietary compression scheme in the future. Predictably, the run length code systems worked very well in this test. In a sense they were on 'home ground', whereas LZW was playing an 'away match'. Nevertheless, in compression ratio, it outperformed the CCITT scheme in full field of view images, albeit by a slim margin.

The area of interest images show a completely different picture. RLC–B-7 out-performs the others by an order of magnitude. The 660-fold compression achieved here is no fluke either, as the full set of 1559 objects in the set of 25 test images could be stored as a file of 1559 images of on average 38.45 bytes per image, yielding an average compression ratio of 639-fold. This massive compression ratio

for these sparse images is not the result of the B-7 code used, but of the method used to construct the run lengths. Both other run length codes compress each row separately, which means that each empty row is coded by a single code word. If the image were completely empty this would require 384 code words of 2 bytes, yielding a compression ratio of 32, close to what was found. Our coder can code multiple row runs, needing just 1 byte (a stop byte) to code an empty image. If an image contains a single object of interest, all the black area above it is coded in a single word, of 1–3 bytes, and the area below it is coded in a single byte (the stop byte), this means that the largest part of the image can be stored in just 2–4 bytes. For convex areas of interest 3 further bytes are needed on average to compress each scan-line occupied by the object of interest. Limiting the number of code words needed, rather than an efficient choice of code words is the basis for the efficiency of RLC–B-7 compression. This is highlighted by the fact that writing the run lengths without any encoding (as 4 byte long integers), yields a compression ratio of 234 in the case of individual objects. It might be suggested that the run lengths generated in this way could be compressed using Huffman code, which is a minimum redundancy code [10], yet this would sacrifice simplicity, which is one of the main attractive features of this scheme. The same objection holds for compressing the run lengths using LZW, but since we already have a compressor and decompressor for LZW, the objection is somewhat less severe.

LZW's adaptive nature is again evident in its compression ratio of 52, but RLC–B-7 clearly makes better use of the a priori knowledge we have about the image, i.e. that it is practically empty.

The object file size in Table 1 reflects the code complexity to a certain extent, although it must be stated that in the case of CCITT code this includes a static array containing the code words. Nevertheless, it is evident that the RLC–B-7 and MPB are most compact in code size. On the other hand, MPB does not compress nearly as well as RLC–B-7. These two schemes are closely followed by LZW, which does need rather more memory at run time.

RLC–B-7 is also the fastest coder and decoder, which is probably due to its simplicity. It is closely followed by CCITT, which is just 20% slower. MPB and LZW lag behind somewhat. This is due to the need to pack the (byte oriented) super-VGA display format scan-lines into packed arrays of bits before coding, and unpacking them after decoding. The time needed to pack 384 rows of 512 bytes into 384 rows of 512 bits is 1.48 s. Unpacking costs the same amount of time. It can easily be seen from Table 4 that this accounts for the difference in speed entirely. In fact, MPB and LZW are faster than the other two, when taking packing and unpacking into account. Neither CCITT or RLC–B-7 need to pack the pixels into bits, as run lengths are detected far more efficiently in a byte array than a bit array. When decoding an image, byte rows can be filled very rapidly using such C functions as memset.

The worst case performance of the schemes is rather different. For both RLC–B-7 and CCITT, the worst case situation arises when adjacent pixels never have the same value, i.e. all run lengths are 1. RLC–B-7 would then store each pixel as a byte, yielding an expansion of 8 times. For CCITT, each background pixel would be coded in 6 bits, and each foreground pixel in 3, yielding 4.5 times expansion as its worst case performance. MPB only adds 1 byte in each 128 bytes of packed image data in the worst case. The worst case for LZW arises when each sequence of two bytes (in the packed image data) occurs only once within each 4 kB of image data. In this case an expansion of some 40% is expected.

Comparing image compression schemes is very difficult, unless a single scheme stands out on all points, and in all situations. Weighing speed, average compression, worst case performance, memory requirements and portability is not easy. Nevertheless, when turning to our specific application, worst case performance need not be considered very important. Any image approaching the worst case is considered to be too noisy to be meaningful in our context. The same probably holds for many applications. The memory requirements of the schemes under review here are different, but none are prohibitive, given the memory usually available on modern computers. Speed and average compression are far more important criteria in this comparison. For full field of view images, RLC–B-7 is best, but 8% gain in compression, and 20% gain in speed (both with respect to CCITT) is not really sufficient to merit the use of this proprietary format. When

storing individual bacteria in separate images, the 12.8-fold gain in compression with respect to LZW, and more than 20-fold gain with respect to CCITT clearly merits the continued use of this scheme in the future. Any other application which uses large numbers of binary images which contain only a few small objects in a uniform background could well benefit from this scheme.

Acknowledgements

I would like to thank the Dutch Foundation for Micro-Morphological Systems Development and the Institut für Microöcologie, Herborn-Dill, Germany, for their support of this research.

References

[1] B.C. Meijer, G.J. Kootstra and M.H.F. Wilkinson, A theoretical and practical investigation into the characterisation of bacterial species by image analysis, Binary Comput. Microbiol. 2 (1990) 21–31.

[2] B.C. Meijer, G.J. Kootstra, D.G. Geertsma and M.H.F. Wilkinson, Effects of ceftriaxone on faecal flora: analysis by micromorphometry, Epidemiol. Infect. 106 (1991) 513–521.

[3] B.C. Meijer, G.J. Kootstra and M.H.F. Wilkinson, Morphometrical parameters of gut microflora in human volunteers, Epidemiol. Infect. 107 (1991) 383–391.

[4] M.H.F. Wilkinson, G.J. Jansen and D. Van der Waaij, Groningen reduction of image data: a microbiological image processing system with applications in immunofluorescence and morphometry, Microecol. Ther. (in press).

[5] T.A. Welch, A technique for high performance data compression. IEEE Comput. 17(6) (1984).

[6] Tag Image File Format (TIFF): Specification Revision 5.0. In: Practical Image Processing in C, C.A. Lindley (Wiley, New York, 1991).

[7] Standardization of Group 3 facsimile apparatus for document transmission. In: Recommendailon T.4, Vol. VII., Fascicle VII.3, Terminal Equipment and Protocols for Telematic Services, pp. 16–31 (The International Telegraph and Telephone Consultitative Committee (CCITT), Geneva, 1985).

[8] Facsimile coding schemes and coding control functions for Group 4 facsimile apparatus. In: Recommendation T.6, Vol. VII., Fascicle VII.3, Terminal Equipment and Protocols for Telematic Services, pp. 40–48 (The International Telegraph and Telephone Consultitative Committee (CCITT), Geneva, 1985).

[9] RC. Gonzales and P. Wintz, Digital Image Processing, 2nd edition, pp. 263–268 (Addison-Wesley, Reading MA, 1987).

[10] D.A. Huffman, A method for the construction of minimum redundancy codes, Proc. IRE 40(10) (1952) 1098–1101.