Practical Evaluation of a Data Compression Algorithm

Douglas W. Jones[1]

Department of Computer Science
University of Iowa
Iowa City, Iowa 52242
(319) 353-7479
jones@herky.cs.uiowa.edu

1 Introduction

The idea of using splay trees as the basis of a prefix code for data compression was introduced in 1988 [4]. At around the same time, the University of Iowa Physics department began development of the Visual Imaging System for the ISTP POLAR satellite, to be launched in July 1993 [6]. What follows is a report on the feasibility of using splay-tree based compression for image data transmitted from this satellite. In short, we concluded that this algorithm is appropriate for use in this and other similar contexts.

The on-board processing resources available to satellite-based systems are significantly limited by a number of factors. For example, a microprocessor may be considered obsolete for use on earth by the time it is available in low-power, radiation hardened, launch certified form. Additionally, high radiation levels limit the use of dynamic memory technology, and power restrictions further limit the available memory resources. Finally, downlink bandwidths are severely restricted for numerous reasons.

The ISTP POLAR Visual Imaging System provides an example of these constraints. At the time this work was done, it was expected that this system would be based on a pair of 80C86 processors clocked at 3.5 MHz. Each processor was to have only 64K of private RAM, only a fraction of which would be available for compression. Finally, the downlink bandwidth allocated to the Visual Imaging System is only 11 KBaud.

The scientific context of this system places a high value on obtaining sequences of images in quick succession. In this context, transmitting each image as it is collected is not adequate because of the limited downlink capacity. For example, a 256x256 image that is collected in 4 seconds would take 48 seconds to transmit over the downlink. This provides ample motivation for the use of data compression.

There are a large number of data compression algorithms [1, 5, 7], but most are unsuited to this application. For example, although the widely used LZW algorithm is quite fast, the memory required to hold a dynamically constructed dictionary of

[1]This work was supported, in part, by NASA contract number NAS5-30316

common strings found in the data being compressed is greater than the available onboard memory.

Vector-quantization has been widely investigated as an image compression method for use in High Definition Television, but the working assumption used has been that large amounts of processing power were available to the transmitter while only limited amounts of processing power were available at the receiver, exactly the opposite of the situation with the hardware originally proposed. Furthermore, vector quantization is lossy; the reconstructed image may be visually equivalent to the original, but it is only an approximation.

The splay-tree based compression algorithm offers a new alternative. This was originally presented in [4], and it has the following characteristics: The code is a prefix code, as used in Huffman codes and their variants. Thus, each byte (or pixel) to be transmitted is represented by a string of bits, with the more common bytes represented by shorter strings of bits in the compressed data.

At its simplest, splay-tree based compression requires only 2310 bytes of RAM to hold a single tree and the stack used to reverse bit order. Unlike Huffman codes, splay-tree based codes require no advance knowledge of the statistical character of the data. Adaptive Huffman codes also avoid the need for prior statistical knowledge, but the splay-tree algorithm is faster and uses less memory. As with all adaptive codes, the bit string used to encode a particular byte may vary from one occurence of that byte to the next.

Finally, unlike any of the other common data compression algorithms, splay-tree based codes are locally adaptive; that is, if an image changes character in mid-stream, the splay-tree code will re-adapt to the new context. Splay-tree based codes are not optimal in the sense that Huffman codes are, but their locally adaptive behavior frequently allows them to outperform other codes on images or other data that consists of regions with differing statistical characteristics.

The remainder of this paper is devoted to a study of the performance of the splay-tree based data compression algorithm in the context of the hardware originally proposed for the ISTP POLAR Visual Imaging System. This includes estimates of the expected compression ratios, the speed of compression, and the impact of transmission errors on the compressed data.

It should be noted that this work is not relevant to the ISTP POLAR Visual Imaging Subsystem as actually built. The expected radiation hardened 80C86 processors were not available, so four 2 MHz 80C85 processors were used. These are insufficient to accomplish any useful data compression without auxiliary hardware, but it was possible to achieve acceptable degrees of compression using vector quantization by adding a DMA controller chip and and auxiliary arithmetic units to each processor.

2 Compression Ratios

The original tests of the splay-tree based compression algorithm reported in [4] included tests of the algorithm on three digitized portraits of human faces. In these experiments, the algorithm reduced these images to 0.235 times their original sizes, giving a compression ratio of 4.25 : 1 (original : compressed).

The data expected in the ISTP POLAR Visual Imaging System would not be expected to have the same character as digitized portraits. The cameras will include image intensifier hardware, and as a result, the value associated with each pixel in an image will be simple function of the small number of of photons arriving at that pixel. The net effect of this will be equivalent to superimposing random snow on each image.

A tape containing 385 images from the Dynamics Explorer 1 satellite was used to test the splay-tree based compression algorithm on such data. These images are lower resolution than those expected from the ISTP POLAR hardware, and they were obtained by significantly simpler camera hardware [2], but they are expected to have similar statistical characteristics.

When these images were compressed using the splay-tree based compression algorithm, the average compression ratio was 2.42 : 1 and the median was 2.33 : 1. One image was compressed to 6.8 : 1, but the remaining ones were compressed in the range of 1.3 : 1 to 5.2 : 1, with only a few files compressed to better than 3.3 : 1.

It is significant that the splay-tree based compression algorithm never made things worse. Most compression algorithms have a hard time dealing with completely random inputs, and the snow resulting from the photon counting behavior of the imaging systems is random. In the presence of purely random data, the splay-tree based algorithm would be expected to perform as poorly as 0.8 : 1.

In examining the distribution of compression ratios for the 385 images, there were two large peaks, one at around 1.5 : 1 and one at around 2.3 : 1. The former peak, with poor compression, was characterized by bright snowy images, typically those taken of the day-side of the earth. The latter peak was the largest and was typified by night-side images with dark backgrounds.

In the absence of noise or other high frequency components, delta coding can improve the performance of many data compression algorithms. Delta coding involves storing the pixels of an image as a sequence of differences, where each difference encodes the change in brightness between a pixel and its predecessor on the scan line. Averaged over all 385 images, delta coding was not an improvement; it degraded the average compression ratio to 2.29 : 1, but improved the worst case, with no image compressed to worse than 1.5 : 1.

3 Speed

The initial attempt to estimate the speed of the splay-tree based compression algorithm on a 3.5 MHz 80C86 indicated that the algorithm, as presented in [4], was too slow, but within a factor of two of an acceptable speed. Inspection of the code indicated that the primary problems were caused by a shortage of registers, by too many branch instructions, and by expensive array indexing.

On the 8086 processor, branch instructions impose a significant performance penalty because they cause pipeline flushes. To avoid this, the conditionals in the innermost loop in the program were unfolded, resulting in considerable duplication of code but the elimination of all but the essential branches.

Array indexing involves multiplying the offset into the array by the size of an array element. The splay-tree based compression algorithm uses three arrays of 16-bit words, and in the initial 8086 version of this code, the necessary multiplications by two were performed by adding registers to themselves. All of the array elements in the program contain integers used to simulate pointers, and as a result, it was possible to eliminate the multiplication operations at run-time by pre-doubling all the array elements as they are initialized.

After both of these changes were made, it was possible to squeeze all of the working variables of the compression algorithm into registers. The central part of the resulting algorithm, the `compress` procedure, was coded in 8086 assembly language for performance analysis. The analysis was done by counting clock cycles with reference to the 8086 technical documentation [3].

From this analysis, the average and worst-case times for a call to the `compress` routine were determined. This analysis is valid for compression ratios of up to 5.33 : 1. The best possible compression ratio with this algorithm is 8 : 1. The results of this analysis are given below and plotted in Figure 1. Here, T denotes an average time and W denotes a worst case time, where all times are measured in 8086 machine cycles and b/p is the compression ratio in bits per 8 bit pixel:

$$T_{pixel} = 74 + 330 + 368\frac{b/p - 1.5}{2} = 128 + 184b/p$$

$$W_{pixel} = 74 + 417 + 384\frac{b/p - 1.5}{2} = 203 + 192b/p$$

$$T_{bit} = \frac{T_{pixel}}{b/p} = 184 + \frac{128}{b/p}$$

$$W_{bit} = \frac{W_{pixel}}{b/p} = 192 + \frac{203}{b/p}$$

The worst case and average case times per bit of compressed data differ because the cost of a conditional branch instruction depends on whether the branch is taken. If there is sufficient buffering of compressed data, only the average case matters, but

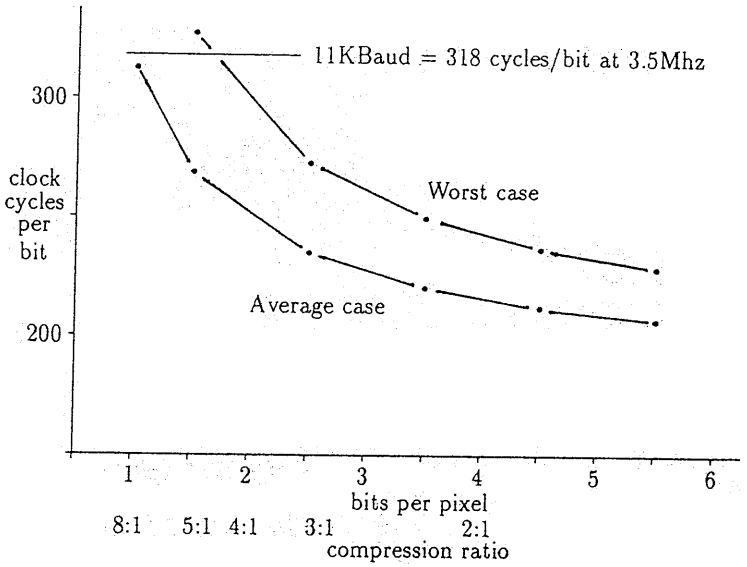

Figure 1: Compression time as a function of compression ratio.

if compressed data is to be output directly to the transmission line from the compress function, the worst case would be significant.

Given the empirical determination that the splay-tree algorithm rarely compresses data better than 5 : 1, compression can be done using roughly 85% of the available CPU time on an 3.5 MHz 80C86 transmitting at 11KBaud; this leaves about 50 clock cycles per transmitted bit available to other purposes. In the worst case analysis, however, 100% of the available cycles can be consumed at a 5 : 1 compression ratio.

If data to be transmitted is broken up into frames or blocks, with buffers delivered periodically to the transmission subsystem, then the likelyhood of the worst case being repeated over the length of a buffer becomes infinitesimal and can be ignored. Thus, it is safe to draw the conclusion that splay-tree based compression is computationally feasible in the environment of the ISTP POLAR Visual Imaging System as originally proposed.

4 Transmission Errors

Computational feasibility is not sufficient to justify use of a data compression algorithm for transmitting downlink data. Transmission errors in compressed data can destroy the utility of that data. Some compression schemes are comparatively

immune to errors. Among these, vector quantized data and data transmitted with static Huffman codes are particularly resistant to corruption.

Unfortunately, data compressed with the splay-tree algorithm is highly vulnerable to transmission errors. Although there is a small probability that an error will cause only one pixel to be received in error, even single-bit errors are likely to completely corrupt the entire stream of compressed data from the point of error to the end.

An accurate characterization of the expected error rates is needed to determine whether this is a serious problem. The downlink error rate included in the specifications for the ISTP POLAR mission is a maximum of 1 error per 100,000 bits, a figure which translates to one error per 12,500 bytes. This suggests that, on the average, every 256x256 image would be expected to contain more than 4 errors.

In order to further understand the nature of the errors, the patterns of errors in the 385 Dynamics Explorer 1 images were examined. These files contained a total of 6,798,000 8 bit pixels, of which 10920 were set to a value of 255 indicating that the pixel may have been received in error. Most images were 152 by 120 pixels, or 18240 pixels, and only 20 images out of 385 contained 255 codes.

Visual inspection of the 20 images containing 255 codes revealed that the 255 codes in 3 of the files were not error indications but indicated extremely bright areas where the imaging system was saturated. The remaining 255 codes were clearly grouped into 22 distinct error events, 6 of which involved isolated pixels and 16 of which involved runs of two or more pixels, usually many more.

22 error events in 6,798,000 bytes of data represents an error rate of roughly 1/500 the specified rate of one error in 100,000 bits. Whatever the cause of the difference between observed and specified error rates, the fact that a majority of the observed errors involved runs of pixels suggests that the addition of explicit error correcting features to the data compression system would be of little use.

As proposed, the ISTP POLAR Visual Imaging Subsystem was to be able to send images as large as 512x512 pixels; at the observed error rate of one uncorrectable error event every 300,000 or so bytes, 62% of all images would be expected to be received with an uncorrectable error. Using data compression reduces file sizes and thus reduces the probability of an error corrupting any particular file, but at compression ratios of 2.5 : 1, roughly 25% of all transmitted images would contain errors.

The effective gain G_e of a data compression system in the presence of uncorrectable errors can be stated as the ratio of the number of undamaged images or other messages received to the number of images that would have been received had compression not been used. For the purpose of conservative analysis, it is assumed that transmission errors never corrupt an uncompressed image.

In the absence of errors, G_e is the same as the compression ratio C. G_e decreases with increasing probability of an image being damaged by an error; this probability increases with increased image size S and with increased error rate R.

$$G_e = C - SR$$

Fortunately, it is quite easy to reduce the image size by the simple expedient

of transmitting each image in multiple parts, where each sub-image is compressed independently and the start of each sub-image represents a possible error recovery point. If the subimages are interlaced in a manner comparable to the way television pictures are interlaced, for example, by transmitting every other pixel of every other scan line as a subimage, it may be practical to reconstruct approximations of the pixels of a lost subimage from their neighbors in other subimages. The effect of this scheme is that lost subimages would be evident in the final data as lost resolution.

At the error rates observed in Dynamics Explorer 1 data, transmitting 512 by 512 images as 4 subimages of 256 by 256 pixels would lead to an effective gain greater than 1 for all observed compression ratios. At compression ratios of around 2.5 : 1, as observed for Dynamics Explorer 1 data, G_e would be better than 2.3 : 1.

5 Recommendation

The cost of subdividing an image into multiple subimages is small but significant. Prior to transmitting each subimage, the data structure representing the splay-tree must be reinitialized. This occupies 1026 16 bit words. Using computational approaches to initialization, this data structure can be built in about 44000 clock cycles; a block-copy from a pre-constructed data structure in ROM can be done in about 24000 cycles.

Given the empirical determination that the splay-tree algorithm rarely compresses data by more than 5 : 1, and that at this compression ratio, there are roughly 50 spare clock cycles per bit of compressed data, at least 24000/50 bits must be transmitted before the time taken to initialize the data structures can be ignored. This allows reinitialization as frequently as every 300 pixels during the transmission of an image.

Because the splay-tree algorithm is adaptive, it must be given time to adapt before it begins to perform well. Specifically, it must encounter the more common pixel values in the image a few times each so that the tree branches associated with those values can be shortened. As a rule of thumb, if there are 256 possible values for each pixel, then the minimum size image segment needed to make the splay-tree approach the optimal balance will be some small multiple of 256 pixels.

This suggests that a 512 by 512 image can be safely divided into about 256 sub-images (each composed of every 16th pixel of every the 16th scan line), but it should be noted that another consideration argues for larger sub-images: To allow the start of each sub-image to be easily identified, it must be clearly marked. This is easily accomplished if the compressed data for each sub-image is sent as a sequence of frames or blocks, where each includes a header that identifies the initial block of each sub-image. This scheme will waste an average of half of a block at the end of each sub-image, and this waste can only be ignored if a fairly large number of blocks are used per sub-image.

Thus, it is reasonable to think in terms of subdividing 512 by 512 images into something like 16 sub-images, sending every 4th pixel of every 4th scan line in each

sub-image. If 256 by 256 images are common, this would allow them to be divided into 4 sub-images of the same size as the sub-images used for 512 by 512 images.

6 Code

Both the optimized C and assembly language versions of the compression algorithm are available from the author. The following optimized C code can be contrasted with the code from [4]. This code was written so that each statement corresponds to one machine instruction.

```
#define MAXCHAR 256          /* number of distinct pixel values */
#define TWICEMAX 513         /* 2 * MAXCHAR + 1 */
#define ROOT 0               /* tree root is left[0],right[0],up[0] */

#define prefix *(WORD *) &   /* allows byte indexing to each word */

BYTE left [MAXCHAR * 2];   /* prefix left[i] = left child of i */
BYTE right [MAXCHAR * 2]; /* prefix right[i] = right child of i */
BYTE up [TWICEMAX * 2];    /* prefix up[i] = parent of i */
BYTE stack[MAXCHAR];       /* used to reverse order of bits sent */

compress(plain)
WORD plain;
{   register BYTE *sp;  /* stack pointer */
    register WORD a, b; /* children of nodes c and d */
    register WORD c, d; /* pair of nodes to be semi-rotated */

    a = plain + MAXCHAR; a <<= 1;
    sp = &stack[0];
    for (;;) { /* walk up tree semi-rotating pairs of nodes */
        c = prefix up[a];
        if (a == prefix left[c]) { /* a is the left son of c */
            *sp = 0; sp++;
            if (c == ROOT) break;
            d = prefix up[c];
            b = prefix left[d];
            if (c == b) { /* c is the left son of d */
                *sp = 0; sp++;
                b = prefix right[d];
                prefix right[d] = a;
                prefix left[c] = b;
                prefix up[b] = c;
                prefix up[a] = d;
```

```
                a = d;
                if (a != ROOT) continue;
                break; /* loop exit! */
            } else { /* a is left son of c, the right son of d */
                *sp = 1; sp++;
                prefix left[d] = a;
                prefix left[c] = b;
                prefix up[b] = c;
                prefix up[a] = d;
                a = d;
                if (a != ROOT) continue;
                break; /* loop exit! */
            } /* control never reaches here */
        } else { /* a right son of c */
            *sp = 1; sp++;
            if (c == ROOT) break;
            d = prefix up[c];
            b = prefix left[d];
            if (c == b) { /* c is the left son of d */
                *sp = 0; sp++;
                b = prefix right[d];
                prefix right[d] = a;
                prefix right[c] = b;
                prefix up[b] = c;
                prefix up[a] = d;
                a = d;
                if (a != ROOT) continue;
                break; /* loop exit! */
            } else { /* a is right son of c, the right son of d */
                *sp = 1; sp++;
                prefix left[d] = a;
                prefix right[c] = b;
                prefix up[b] = c;
                prefix up[a] = d;
                a = d;
                if (a != ROOT) continue;
                break; /* loop exit! */
    }   }   } /* control never reaches here */

    /* all break statements above branch to here */
    for (;;) { /* pop bits off the stack and transmit them */
        --sp; bitbuf <<= 1;
        bitbuf |= *sp; --bitcnt;
        if (bitcnt != 0) { /* normal case, bitbuf not full */
```

```
            if (sp != &stack[0]) continue;
            return;
        } else { /* abnormal case, bitbuf full, transmit it */
            *byteptr = (char)(bitbuf & 0xff); ++byteptr;
            if (byteptr != bytemax) { /* normal, bytebuf not full */
                bitcnt = MAXBITCNT;
                if (sp != &stack[0]) continue;
                return;
            } else { /* abnormal , bytebuf full, transmit it */
                sendbuf();
                bitcnt = MAXBITCNT;
                if (sp != &stack[0]) continue;
                return;
}   }   }   } /* control never reaches here */
/* end compress function */
```

References

[1] Bell, T., Witten, I.H., and Cleary, J.G. "Modeling for Text Compression." *ACM Computing Surveys, 21,* 4 (Dec. 1989) 557-591.

[2] Frank, L.A., and Craven, J.D. "Imaging Results From Dynamics Explorer 1." *Review of Geophysics, 26,* 2 (May 1988) 249-283.

[3] Intel. *iAPX 86, 88, 186 and 188 User's Manual: Programmer's Reference.* Intel, Santa Clara, California, 1983.

[4] Jones, D.W. "Application of Splay Trees to Data Compression." *Communications of the ACM, 31,* 8 (Aug. 1988) 996-1007.

[5] Lelewer, D.A., and Hirschberg, D.S. "Data Compression." *ACM Computing Surveys, 19,* 3 (Sept. 1987) 261-296.

[6] NASA Request for Proposal RFP5-15800/504 for the International Solar Terrestial Physics (ISTP) / Global Geospace Science (GGS) space segment. Dec 2, 1987.

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

DATA COMPRESSION

CONFERENCE

1951-1991
IEEE Computer Society Press
The Institute of Electrical and Electronics Engineers, Inc.