NSUCRYPTO 2020: JPEG Compression

Ioan Dragomir[1], Gabriel Tulba-Lecu[2], and Mircea-Costin Preoteasa[3]

[1] `ioandr@gomir.pw` – Technical University of Cluj-Napoca
[2] `gabi_tulba_lecu@yahoo.com` – Polytechnic Univeristy of Bucharest
[3] `mircea_costin84@yahoo.com` – Polytechnic Univeristy of Bucharest

Table of Contents

1 Problem summary 1
2 Minimal output size, extreme Kolmogorov complexity 2
  2.1 Comparing Exp-Golomb to the optimal coding 2
3 Proposed encoding 3
  3.1 Zig-zag order alternatives 3
  3.2 Distinction between the first element and the rest 3
  3.3 Compression scheme 3
  3.4 Attempting to rethink sparsity 4
4 Conclusion, TL;DR 5

1 Problem summary

An important part of the JPEG image format is the compression. Each 8x8 block of pixels is converted to as short a bit string as possible, by computing its Discrete Cosine Transform, quantizing it, and then efficiently encoding the quantized matrix, using the property that most non-zero data is accumulated in the top left corner, corresponding to lower spacial frequencies. For example, here are some of the quantized matrices in the input set:

$$\begin{bmatrix} 127 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix};
\begin{bmatrix} 9 & -5 & -2 & -1 & -3 & 1 & 0 & 0 \\ -2 & 5 & 4 & 0 & -1 & -1 & 0 & 1 \\ 3 & 1 & 6 & 2 & 3 & 2 & 1 & 0 \\ 1 & -2 & 1 & 0 & -1 & 0 & 0 & 0 \\ -2 & 0 & -1 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & -1 & 0 & 0 & 0 & 0 & 0 \\ 0 & -1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix};
\begin{bmatrix} -55 & -18 & 6 & -3 & 0 & 0 & 0 & 0 \\ -3 & 8 & -2 & -3 & 1 & -1 & 1 & 0 \\ 2 & 1 & 1 & -3 & 0 & 0 & 0 & 0 \\ 0 & 4 & -4 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & -1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

Notice the greatest values appear in the top left corner, and the bottom right corner rarely contains anything but zeros. Most compression schemes, then, use this property to their advantage, rearranging the matrix to a vector by starting in that corner and zig-zagging their way starting through the top-left triangle, in such a way that the vector will usually end in a long string of zeros, which can be easily reduced.

Question: Propose an alternative algorithm to compress and decompress such matrices. It will be graded on how it performs given a predefined list of 108216 matrices.

2 Minimal output size, extreme Kolmogorov complexity

We present a construction which obtains the lowest possible size for the output, but is impractical for use as an actual compression algorithm because it requires storing the expected data set, or at least a slightly condensed version, as part of the algorithm. Thus its Kolmogorov complexity scales with the size of the data set, and it cannot be effectively used generally, as the possible matrix set needs to be known in advance.

Briefly, our construction assigns each unique matrix in the input set a different symbol, and uses Huffman coding to generate an encoding which maps short bit strings to matrices which appear often, (such as matrices corresponding to monochrome blocks, which have a single nonzero value in the top-left corner) at the expense of inefficiently encoding matrices which only appear a handful of times in the whole data set.

The implementation is trivial, and it manages to encode all 108216 matrices to a remarkable 1648250 bits ($\approx$ 201kb), for an average of 15.23 bits per matrix. As expected, the most common matrices (such as $[127, 0, 0, \dots, 0]$, which appears 2544 times) have short encodings (in our implementation, 01001), whereas the matrices which only appear once get longer encodings. (e.g. $[79, -1, 0, 0, 0, 0, 0, 0, 1, 0, \dots, 0] \to$ 01011110000100010)

By the nature of Huffman coding, this is an optimal solution for the given data set. Also, we cannot divide the input data in any other way than matrix-wise, as that would break the requirement that we be able to reverse some bit string to a exactly a matrix.

2.1 Comparing Exp-Golomb to the optimal coding

The task authors gave an example of such a compression algorithm – the Exp-Golomb code. We will compare its performance to the optimal coding we devised earlier.

It manages to compress the entire matrix list to 6618432 bits, which gives an average of 61.15 bits per matrix. Longer encodings do indeed correspond to rare matrices, e.g.:

```
101001111111101000111111110100011111111000001011111000111101111111100110011
111010001111100011111100111111011011110100011011111101001111010111111001101
110101111100010111100100011110110111110100111010111010011100001111100010100
 110111110000100010011001100010001011100010000001000101001011010100000100
```

corresponds to

$$\begin{bmatrix} 71 & 17 & -28 & 17 & -14 & 5 & -2 & 0 \\ -69 & 3 & -11 & 5 & -10 & 3 & -1 & 0 \\ -11 & -15 & 9 & -12 & -1 & 0 & 0 & 0 \\ 6 & 3 & 0 & -10 & -1 & 0 & -1 & 0 \\ 8 & 13 & -4 & -3 & -1 & 0 & -1 & 0 \\ 9 & 4 & -1 & -2 & 1 & 0 & 0 & 0 \\ 2 & 0 & 1 & 0 & 1 & -1 & 0 & 0 \\ -1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \end{bmatrix}$$

But the shortest encountered encoding `00000111001` corresponds to the matrix $(-3, 0, \ldots, 0)$, which only appears once in the data set. The most common matrix, $(127, 0, \ldots, 0)$, is encoded as `0000011111111011111111`, which is rather long.

3 Proposed encoding

3.1 Zig-zag order alternatives

While the zig-zag order makes intuitive and practical sense, we can obtain slightly better results by counting, for each position, how many input matrices have a nonzero element there, and using these values to infer a better matrix linearisation order:

$$\begin{bmatrix} 0 & 2 & 3 & 9 & 18 & 24 & 30 & 39 \\ 1 & 4 & 8 & 13 & 19 & 28 & 33 & 35 \\ 5 & 7 & 10 & 17 & 23 & 31 & 40 & 38 \\ 6 & 11 & 14 & 20 & 26 & 42 & 46 & 48 \\ 12 & 15 & 21 & 27 & 36 & 50 & 54 & 44 \\ 16 & 22 & 29 & 34 & 43 & 53 & 59 & 55 \\ 25 & 32 & 41 & 47 & 51 & 57 & 60 & 61 \\ 37 & 45 & 49 & 52 & 56 & 58 & 62 & 63 \end{bmatrix}$$

If we change the order but keep the exp-Golomb encoding, we reduce the total size by 1.24%, which isn't a remarkable improvement by itself, but it is a sign we're heading in the right direction.

3.2 Distinction between the first element and the rest

If we measure the frequencies of each value for each position, we notice the top-left element is significantly different from all the rest. Intuitively, it represents the average intensity over the 8x8 block, so we can expect there to be a great amount of variance. On the other hand, the other spatial frequencies follow a rather regular pattern of the values being, from most common to least, $0, 1, -1, 2, -2, 3, -3, \ldots$ with their relative frequencies following a seemingly exponentially decreasing trend (e.g. for the second element, $p(x) \approx e^{-0.2|x|-0.6}$).

3.3 Compression scheme

We will use Huffman coding for the predictable elements. The constructed tree has the great property that zero elements get mapped to a single bit, because they are more common than all other values combined.

Top-left element encoding. The first element can either be encoded as a raw 8-bit value, in the idea that its high variability won't lead to great compression, or we can also use Huffman coding to try to reduce its size as much as possible. We will compare the output size for each method, using the given matrix list as a representative sample to build the Huffman tree.

Storing the value as 8 uncompressed bits would lead to $8 \cdot 108216 = 865728$ bits added in total. Using a compressed coding, however, we get 843018 bits, for an average of 0.2 bits less per matrix. While it is an improvement, it is marginal, and for the sake of generality and simplicity, we will not do any compression on the top-left element.

Thus, we will compress elements in the order deduced in the last part, where the first element is encoded as an 8 bit two's complement value, and the rest use a predefined Huffman tree. For now, we will also use a 6 bit prefix for the number of nonzero elements.

With this method, we compress the matrices to 6013327 bits for 55.56 bits per matrix. This is already better than Exp-Golomb, but not by much. The shortest encodings are always 14 bytes long (6 for the length, 8 for the top-left element), and the longest encoding is shorter than Exp-Golomb's as well, but only by 3 bits.

3.4 Attempting to rethink sparsity

In the current algorithm, all encodings start with 6 bits for the "number of nonzero elements", even though many times this value is very small. We will attempt two solutions to this problem. We thought this would improve performance, but we were wrong. Nevertheless, here are our attempts.

1. Exp-Golomb for the length. First, we tried using the Exp-Golomb code without the sign bit to store the length. Small values, which appear often, will get reduced to a couple of bits, and we only get larger strings for $n > 14$, which accounts for 17.7% of matrices. Unfortunately, this method leads to longer compressed data, with an avg. of 56.08 bits/matrix.

2. Partitioning the array. Another attempt bypasses the need for a length field completely. We will split the 63 elements (we skip over the first one) in two halves, such that there is an equal probability that each of the halves is full of zeros. We can then recursively split each half further. We will encode any empty half as a zero bit, and any half with at least one nonzero element with a one bit, followed by its contents, which may either be encoded elements, or two such "halves".

For example, if we split the data in quarters, and the first and third quarters contain the encoded blocks A and B, the matrix encoding will look like `11A010B`. The first 1 means we divide the whole matrix in half, the second one divides the first half further, then A is the encoding of the first quarter, 0 is the encoding of the second quarter. We then step one level higher and the next 1 divides the second half into quarters, the first of which has encoding 0 "empty", and the second encoding B.

We thought it was is inefficient to split the arrays at the halfway point based on their length, so we will search for a better splitting point so as to balance the cases in which each half is full of empty elements. In this way, given the input data, we will make the first split at position 36, its left half will be split at position 22 and its right half at position 50, etc.

Against our intuition, this happens to be wildly inefficient, the best variant merely reaching 58.8 bits/matrix, which is worse than all attempts thus far.

4 Conclusion, TL;DR

All in all, we devised an algorithm which impractically produces the theoretical minimum compressed size of 15.23 bits/matrix, hoping to also get close to its performance by more general means. Unfortunately, we were not able to significantly improve the given Exp-Golomb encoding (61.15 bits/matrix), our best attempt only being able to reduce it to 55.56 bits/matrix.