Oscar Brennwald

COMP 590

3/25/2018

Performance of LZW-Arithmetic Coding Compression on Video

**Introduction**:

My approach to our video compression assignment was to implement a version of LZW backed by arithmetic encoding to achieve compression with respect to common and repeated byte sequences. Although this approach can be applied for any file represented in a sequence of bytes, for video, we can interpret the bytes being read as pixels of the video in row order from one frame to the next. Additionally, by storing the video’s byte sequence in a three dimensional array, we can traverse through the video’s three dimensionally adjacent pixels in 5 additional ways (such as by frame, row, and height). If we consider the byte sequence of the video as a three dimensional object in space, this approach allows us to apply LZW to take advantage of strong coherence of a line of bytes in sequence parallel to one of its three axes (one axis with respect to each of height, width, and depth).

**Implementation**:

To implement this algorithm, I relied on the framework for symbol, models, and encoders provided to us as source code. Before implementing the LZW algorithm on the sequence, the byte sequence of the video is read into a three dimensional array to be stored in memory. A runtime argument provided by the user allows them to determine the order in which the bytes of the sequence are traversed (with row, height, frame order being default).

The LZW algorithm is then used to create a dictionary (implemented as a HashMap) of indices mapped to their corresponding byte sequences. Due to memory pressure and runtime considerations, I limited the size of the dictionary to 4096 elements. This constraint was implemented by simply limiting the step of adding of the “wc” sequence of the dictionary only in the case it had less than 4096 elements.

Instead of writing each entry of the sequence of dictionary values emitted by the algorithm as a fixed number of bits corresponding to the maximum permissible size of the dictionary, the output of the values are stored in an ArrayList<Integer> that I then returned as its corresponding integer array. After this, a symbol model capable of holding 4096 elements is created and then trained the model on the integer array of emitted dictionary indices. Because the provided 8-bit model could only store 256 symbols, I created a similar 12-bit model that had similar functionality but instead could store 4096 symbols.

With the model trained, the provided arithmetic encoder is used to output the encoded indices in sequence to. My intuition was that there would be enough entropy between the appearance of emitted dictionary indices to achieve significantly more compression than a fixed length approach. To decode the file, the arithmetic decoder with the same model is used to decode each value of the original indices in order, and returned the resulting array. With the dictionary of the indices regenerated, a map of byte sequences to indices is reconstructed by iterating through the array of indices, and the three dimensional video array is reconstructed in its original order. The appropriate byte values are then outputted in row-height-frame order to the decompressed file.

**Results**:

For context, 450p versions of all of the listed videos were used. The dictionary size for each video reached the maximum (4096 entries) and the timing for training the model on the integer array of indices was less than 1 second. Table 1 refers to results from my first implementation of this algorithm which did not read the video into memory and only traversed the byte sequence in its default row-height-frame order. Table 2 refers to results for the optimal traversal sequences of the videos.

Table 1:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | Bunny | Candle | Jellyfish | Pinwheel | Tractor |
| LZW Compressed Size | 46.2 MB | 1.2 MB | 40.7 MB | 19 MB | 41.4 MB |
| MP4 Compressed Size | 8.1 MB | 256 KB | 19.4 MB | 6.5 MB | 16.3 MB |
| LZW Size/Raw Video Size | 0.856 | 0.022 | 0.753 | 0.352 | 0.767 |
| MP4 Size/ Raw Video Size | 0.150 | 0.005 | 0.359 | 0.120 | 0.302 |
| Array Of Indices Length | 39594553 | 990346 | 38269298 | 24072086 | 33759465 |
| LZW Compression Time | 2m 5s | 3m 36s | 2m 9s | 2m 8s | 2m 19s |
| Arithmetic Encoding Time | 12m 43s | 28s | 10m 41s | 5m 38s | 12m 29s |
| Arithmetic Decoding Time | 8m 6s | 23s | 6m 52s | 3m 54s | 8m 6s |
| LZW Decompression Time | 6m 33s | 6m 39s | 6m 19s | 6m 31s | 7m 11s |
| Total Runtime | 29m 27s | 11 m 6s | 26m 1s | 18m 11s | 30m 5s |

Table 2:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | Bunny | Candle | Jellyfish | Pinwheel | Tractor |
| Optimal Sequence | Frame-width-height | Height-width-frame | width-frame-height | Width-frame-height | Width-height-frame |
| New LZW Compressed Size | 33.9 MB | 967 KB | 38.8 MB | 17.9 MB | 41.4 MB |
| MP4 Compressed Size | 8.1 MB | 256 KB | 19.4 MB | 6.5 MB | 16.3 MB |
| New LZW Size/Raw Video Size | 0.628 | 0.018 | 0.719 | 0.331 | 0.767 |
| MP4 Size/ Raw Video Size | 0.150 | 0.005 | 0.359 | 0.120 | 0.302 |
| Array Of Indices Length | 29013868 | 825118 | 34102102 | 22139864 | 33759465 |
| LZW Compression Time | 2m 4s | 3m 56s | 1m 49s | 2m 8s | 2m 20s |
| Arithmetic Encoding Time | 10m 13s | 29s | 11m 59s | 5m 41s | 11m 24s |
| Arithmetic Decoding Time | 6m 55s | 20s | 8m 35s | 3m 46s | 7m 7s |
| LZW Decompression/Output Time | 6m 9s | 7m 2s | 5m 51s | 6m 27s | 6m 0s |
| Total Runtime | 25m 25s | 11 m 47s | 28m 17s | 18m 2s | 26m 51s |

**Analysis**:

The performance of the old LZW compression technique (results in Table 1) across the videos yielded some interesting results. Although none of the compressed files managed to be more compressed than the corresponding contents of mp4 files, all of the files managed to be at least less than 90% of the raw video size. Interestingly, the relative compression of LZW-compressed files was quite different than what is seen in the mp4 files. Bunny, for example, was the least compressed file for LZW while being the third most compressed file amongst mp4s. One justification bunny’s poor compression could have been a stronger coherence of pixels in the file in non-row order sequence, such as the vertically related individual hairs on the bunny, or the diversely colored right side of the video that is relatively static for the last 2-3 seconds. This was motivation for making the traversal sequence with which the LZW compression is executed more modular.

The performance of updated technique (results in Table 2) showed that we could achieve significantly improved compression by optimally tuning how we traversed a video’s byte sequence for four out of the five provided videos. The lone exception was tractor, who’s optimal traversal sequence was the default of width-height-frame. A justification of this could be the presence of very strong lateral coherence in certain parts of the video such as the white ceiling and cover of the truck. The relative compression of these newly generated files was also more in line with that of the mp4 videos, as candle, pinwheel, and bunny were the first, second, and third most compressed files respectively for both techniques. This new technique presented some potential runtime improvements as compared to Table 1’s technique for some of the videos, but these were likely attributed to improved iteration through data structures (arrays versus ArrayList). Additionally, these runtimes only consider the optimal choice of traversal order, and do not encapsulate how that order is properly determined.

Considering the runtimes of individual components of the algorithm (for Table 2), the arithmetic encoding/decoding seemed to provide the greatest range of values, with runtimes ranging to about a minute combined for each in candle to over twenty minutes combined for jellyfish. Although the arithmetic encoder does allow our compression scheme to capture the entropy of the dictionary entries in our output, it becomes increasingly inefficient with respect to the number of bits in our model.

**Improvements**:

Several improvements could be made to this technique as it stands. Although it currently allows the user to specify the traversal order at runtime, this relies on having a strong intuition to predict which of the six techniques will work optimally or perhaps running all six variations of the algorithm. Considering the relatively quick ability of the algorithm to generate the LZW output list of indices, the algorithm could concurrently generate the output for all six techniques and use some sort of heuristic, such as the size of the list of generated output, to determine which traversal order is optimal to encode and subsequently decode. Based on the given data, the smaller lists of index outputs are correlated with better compression.

Given the previously mentioned performance issues associated with using arithmetic coding on the list of LZW outputs, a fixed bit length encoder could be used in its place and potentially allow us to increase the limit on our dictionary size without as many performance scaling issues. I attempted, but did not finish debugging, an up to 32-bit fixed length encoder version of this algorithm that encodes each output to x bits where 2^x is the lowest power of two greater than the size of the dictionary. Although there is some troubleshooting to be done with respect to the proper use of encoding/decoding with the provided BitSink and BitSource classes, the initial available output seems to suggest that increased dictionary size more than compensates for the loss of entropy encoding of output and the encoding/decoding of output can be much quicker.