Joshua Spisak

CS 1501

Professor Khattab

Assignment 3

There are 4 different implementations:

|  |  |
| --- | --- |
| **A** | The base implementation in LZW, unmodified. |
| **B** | The implementation from A modified to use dynamically sized codewords of size 9 bytes to 16 (run “*LZWmod - n*” in the submitted code to use this implementation), and also parse the file in terms of Bytes instead of a large string. |
| **C** | The implementation from B modified to reset the code book whenever it fills (run “*LZWmod - r*” in the submitted code to use this implementation). |
| **D** | The compress program in MacOs. |

Table of Compression Time (seconds), ratio and size (Bytes) for A, B, C, D:

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **File** | | **Compression A** | | | **Compression B** | | | **Compression C** | | | **Compression D** | | |
| **Name** | **Size (B)** | **Time** | **Size** | **Ratio** | **Time** | **Size** | **Ratio** | **Time** | **Size** | **Ratio** | **Time** | **Size** | **Ratio** |
| winnt256.bmp | 157044 | 1.2 | 159050 | 0.987 | 0.535 | 62930 | 2.496 | 0.516 | 62930 | 2.496 | 0.010 | 62931 | 2.495 |
| bmps.tar | 1105920 | 37.54 | 925079 | 1.195 | 9.183 | 80912 | 13.668 | 9.047 | 80912 | 13.668 | 0.016 | 80913 | 13.668 |
| medium.txt | 25407 | 0.291 | 13197 | 1.925 | 0.289 | 12530 | 2.028 | 0.252 | 12530 | 2.028 | 0.009 | 12531 | 2.028 |
| Lego-big.gif.temp | 93371 | 0.884 | 128973 | 0.723 | 0.391 | 122492 | 0.762 | 0.449 | 122492 | 0.762 | 0.011 | 122493 | 0.762 |
| code.txt | 72351 | 0.356 | 30980 | 2.335 | 0.291 | 24544 | 2.948 | 0.311 | 24544 | 2.948 | 0.006 | 24545 | 2.948 |
| edit.exe | 236328 | 2.833 | 250742 | 0.942 | 0.477 | 156408 | 1.511 | 0.501 | 152230 | 1.552 | 0.011 | 151111 | 1.564 |
| code2.txt | 57701 | 0.368 | 24138 | 2.390 | 0.274 | 20515 | 2.813 | 0.288 | 20515 | 2.813 | 0.004 | 20516 | 2.812 |
| texts.tar | 1382400 | 54.129 | 1012179 | 1.365 | 1.283 | 597846 | 2.312 | 1.357 | 590584 | 2.341 | 0.059 | 589697 | 2.344 |
| large.txt | 1220703 | 28.177 | 605184 | 2.017 | 1.095 | 501776 | 2.433 | 1.261 | 527449 | 2.314 | 0.053 | 522673 | 2.336 |
| assig2.doc | 87040 | 0.659 | 74574 | 1.167 | 0.291 | 40039 | 2.174 | 0.319 | 40039 | 2.174 | 0.013 | 40040 | 2.174 |
| frosty.jpg | 126748 | 1.504 | 177453 | 0.714 | 0.482 | 163788 | 0.774 | 0.511 | 171169 | 0.740 | 0.015 | 163789 | 0.774 |
| gone\_fishing.bmp.Z | 8964 | 0.241 | 12702 | 0.705 | 0.272 | 12598 | 0.712 | 0.261 | 12598 | 0.712 | 0.006 | 12599 | 0.711 |
| wacky.bmp | 921654 | 0.543 | 4302 | 214.23 | 8.517 | 3950 | 233.33 | 8.47 | 3950 | 233.33 | 0.010 | 3952 | 233.21 |

**Compression Time:**

|  |  |
| --- | --- |
|  |  |
| A, B, C, D compression times and relative file size. | B, C, D compression times and relative file size. |

We can see that for implementation A there is a correlation between file size and run time, this is seems to be due to the number of strings that need created with respect to the file size. When ignoring those we can see that B and C both have relatively the same compression time, but D has a much better compression time. This might be due to a better implementation of the symbol table, I would speculate that an efficient structure that traversed reading directly from the input stream until it reaches a prefix could be much more efficient by reducing queries (which my current implementation can still make several of to find a keyboard for the current input).

|  |
| --- |
|  |
| Compression Ratios for all files except wacky.bmp (excluded due to its’ large ratio) |

**Compression Ratio / General Comparison:**

We can see that B/C/D have comparable compression ratios, all of which are better than A. The most stunning difference was in bmps.txt. Some introspection into showed that A wrote 616717 code words to the output file whereas B/C wrote 44406.

As another test I ran my program with another set of options (*“LZWmod – r s”)* that does not use dynamic length code words, but does reset the dictionary whenever it fills. This allowed it to reach almost the same compression ratio in most cases and even beat B/C/D in 4 cases! – one of which is this case, bmps.tar, with a compression rate of 14.695.

The installed compression program (D), which I was unable to analyze more deeply likely has the same strengths that B and C have in terms of being able to generate more code words.

This leads me to believe that since bmps.tar contains differing files, all of which have different patterns that need matched; filling up the dictionary earlier on (when it wrote codeword 3840 specifically) stunted A’s compression ability but B/C/D were able to adapt to new patterns later on giving it an impressive compression ratio.

**Best Compression Analysis; A, B, C, D:**

For all implementations, the best case was wacky.bmp. This is due to the fact that most of the image is whitespace, as a result in the BMP format there will be a large contiguous region of repeated Pixel values of white, the size of the region that can be described by a single codeword will increase until it covers a massive part of the photo, this results in a fantastic compression ratio for all of them, and since B and C (and assumedly D) can encode codewords with fewer bits they have a slightly better compression ratio.

**Worst Compression Analysis; A:**

This applies to the worst ratio seen by implementation A that still resulted in compression (ratio > 1). That would be bmps.tar. The reason for this is mostly discussed above, since there is a variety of different images all of which have different patterns, filling up the number of code words early on results in not being able to create code words for the later files, resulting in a large file since a large number of codewords which encode relative small patterns end up being written to the file.

**Worst Compression Analysis; B, C, D:**

This applies to the worst ratio seen by implementation B, C or D that still compressed the file (ratio > 1), this happened to be the same for all of them: edit.exe. While I have no good way to introspect the file en masse (such as I can with an image), I can certainly speculate that there was a large variety of patterns within the executable, all of which occupied only a few contiguous bytes. The instructions themselves that might repeat are only a single byte, then permutate that with different registers and addresses, then permute the order of those instructions, there are a lot of patterns that can be generated I imagine. While an image can be expected to repeat colors and even patterns, code can be even further complex with no expectation of repeating. Also read the next section on edit.exe.

**No Compression Analysis; A:**

This applies to any test files where the compression ratio was less than 1, and no compression happened, the only files that did not compress solely in A were edit.exe and winnt256.bmp. I compared how many codewords were written to the file edit.exe between A and B, A wrote 167159, B wrote 83098. This seems to be a similar case to what was seen between the compression ratios of A and B/C/D in the best compression analysis; A ran out of codewords early on and was unable to create codewords for later patterns. I saw the same thing happen for winnt256.bmp; A wrote 106031, B wrote 35415; the potential to match more patterns existed but the ability for A to create codewords for more patterns did not. It is interesting to note that just resetting the dictionary when A ran out brought it down to 39660, giving it better compression than B, C or D with a ratio of 2.64.

**No Compression Analysis; A, B, C, D:**

There were 3 files that could not be compressed by any of the implementations: Lego-big.gif.temp, frosty.jpg and gone\_fishing.bmp.Z. This seems to be due to two things, image complexity and the fact that gone\_fishing.bmp.Z and Lego-big.gif.temp all three are already compressed. Taking a look at frosty.jpg it is easy to observe the many gradients and shadows in the image, since there are so many different colors/color patterns interacting it is probably hard to find larger repeating ones. Then factor in the fact that frosty.jpg and the rest are already compressed, the only patterns found are going to be small (~16 bits) and then they are getting replaced with similar sized sections while the smaller patterns (individual bits) are replaced with code words of a greater size (9 – 16 bits). Hence it makes sense that it might have a larger size after compression.