Compressing a LaTeX file

zrlr73

To test my algorithms on, I found two large (1.5MB and 1.3MB) TEX files with open-source licenses on GitHub: one was a syntax-heavy scientific report and the other a more textual set of NBA rules. I chose the inbuilt Python `pickle` library for serialising my structures to store on disk.

Idea 1: Huffman Coding

My first instinct was to get a 'baseline' to compare other results against. I chose to achieve this using the `dahuffman` library. I simply ran this library on the input file and pickled the tuple `(table, encoded)` where `table` was the translation table generated by `dahuffman` and `encoded` was the encoded output. On the decoder, this tuple was unpickled and decoded using the same library. This gained me a compression ratio of around 1.6 on both test documents.

Idea 2: Lempel-Ziv-Welch

I next decided to try and implement a more advanced encoding scheme, and chose LZW because I like the fact that it generates its decoding dictionary as it goes rather than having to store it with the output. I also chose it over LZ78 because its compressed output is simpler and cleaner thanks to the fact that it pre-fills its dictionary.

While the implementation of the LZW encoder went fairly smoothly, I did struggle with one thing: my decoder kept breaking, because it was looking for elements that did not exist in the dictionary yet. I struggled with this for a while, until I looked up the issue and found that it was in fact a quirk of LZW: if the encoder adds something to the dictionary and then immediately uses it in the next step, that item will not exist yet in the decoder's dictionary when it reaches that point. However, because this is the only time at which an error like this is encountered, it is possible to infer the situation and solve it simply by appending the first character of the last decoded value to the end of that same last decoded value.

Once this was resolved, LZW yielded a compression ratio of 1.8-1.9 on both test documents.

My next plan was to consider ways of exploiting predictable redundancies in the string, such that LZW could be left to work on more nuanced and document-specific redundancies.

Idea 3: Collapsing repeated characters

One of my test files contained some series of repeated characters. Indeed, a series of space characters was in the top 3 most common substrings at all reasonable lengths. I decided to write a function to detect these repeated characters, and then replace each run with a single instance of the character being repeated followed by a UTF-8 character signifying the number of repeats being represented. This character was chosen simply as the character at space 255 + `num_repeats` in Unicode. For instance, the string `‘xxxxxx’` would be replaced by `‘xą’` (where ą is the character with code $261 = 255 + 6$).

Preprocessing the string with this method yielded consistent compression ratio gains of 0.01-0.05.

Idea 4: Substring replacement: common substring search

My next idea was to find the most common substrings in the file and replace them with Unicode characters, with a translation table being sent alongside the original document content. I started writing some algorithms that generated counts of every substring in the file and then filtered them such that only the most useful ones remained.

I used a value of `(length_of_string - 1) * number_of_appearances_in_input` to score candidates, with the subtraction of 1 accounting for the character replacing the string. While I found this formula to be effective, I noticed that my filtering algorithms were failing to remove a lot of similar substrings, which would be difficult to robustly filter out to retrieve the most efficient of the bunch. For example, it might find `"\begin{ali"` and `"gin{align}"` as well as `"\begin{align}"`, the actual targeted string. For this reason, I abandoned this method and used what I'd learned to try a similar, but more targeted, approach.

Idea 5: Substring replacement: regex search

Following idea 5, I decided to take a more proactive approach in defining what repeated substrings I was after. I decided to use regular expressions to define likely repeating substrings and, finding that targeting multi-parameter LaTeX commands actually reduced compression ratios and (less surprisingly) increasing the number of matches and ignoring shorter words improved them, settled on the following:

- `(\\[a-zA-Z]+)`
- `[ \n({[]([a-zA-Z]{3,})[ .,:;!)}\]]`

I find all the matches of each regex in the input, produce counts, score each matching string and then sort each regex in descending order of score. The score used is `(length-2)*count` - a metric for the number of bytes saved by replacing that substring. I then produce an allocation of a fixed number of slots in a dictionary to maximise the total score, and replace every instance of each matched string with an allocated Unicode character. For this I have to adjust the lower bound of idea 3, essentially producing the following Unicode character allocation after doing some tinkering to find an appropriate size for this table:

- 0-255: Standard document characters
- 256-2047: Characters representing replaced common strings
- 2048+: Characters used to identify the counts of repeated characters

As stated before, I am assuming that the document only uses characters 0-255, but this scheme could theoretically also be used for extended character sets that include, for example, Nordic letters - the ranges would change, but the same techniques could still be applied. Indeed, the only change needed would be to adjust the value `maxCharacterCodeOfInput` in `encode_decode_config.py`.

The dictionary is then stored alongside the compressed output, making decoding trivial. The function has been written to work with whatever space and regexes are given in order to be easy to tinker with.

Applying this method in between idea 3 and LZW afforded me compression ratios of 2.2-2.5.

Idea 6: Character redistribution

My techniques so far had resulted in a number of character codes above 256 being added to the input during compression. I realised that this could be wasteful as these characters would require twice as much space to store. Therefore, I decided to try redistributing the characters so that they were stored with the most common characters assigned to the smallest character codes and the least common to the largest. This obviously required a list of the most common characters to be stored with the compressed data.

Attempting this gave me similar (but slightly reduced) compression ratios. Storing it without the table of common characters confirmed my suspicions: it did make a very slight improvement (changing one ratio from 2.202 to 2.206), but this was outweighed by the size of the dictionary, which subtracted approximately 0.014 from the aforementioned ratio, giving me 2.188 overall. Therefore I chose not to use it. My guess for this is that the inserted Unicode characters are already relatively few, meaning that redistribution does not change very much.

Idea 7: Results

In summary, I have produced an algorithm which:

- Collapses repeated characters by representing repeats with Unicode characters
- Replaces the most frequently-occurring substrings matching relevant regexes
- Performs LZW on the result, and
- Stores the encoded file to disk using the `pickle` library.

It achieves compression ratios of 2.28 for the scientific paper, and 2.46 for the more text-based NBA ruleset, and is quite easily-configurable and modular which I am proud of. Given more time, I would have tried some other 'standard' compression methods like PPM in comparison to my initial choice of LZW, but I decided to prioritise 'high-level' redundancies that I could derive from the context: ie, trying to model for $\TeX$ syntax and the known character set and language first.