1 LaTeX and English Dictionary

Before executing more traditional compression techniques, I felt it would be a waste not to take advantage of the knowledge given by the problem specification. It's known that the input file is a valid LaTeX file typeset in English. Therefore, the majority of the file will consist of LaTeX commands and English words. I created a dictionary consisting of 553 common LaTeX commands and 29,347 English words (all longer than 2 letters).

When compressing, the encoder creates a trie from the dictionary where each character contains a child for each character that appears after it in the dictionary (with the root being the empty character ϵ), nodes also contain a boolean stating whether they represent the end of a string in the dictionary. When encoding from a certain position, the encoder searches the trie to find the longest match in the dictionary from said position in the input (using the end booleans to avoid matching to prefixes). Once the longest match is found, the encoder replaces this string in the input with a reference to the dictionary before continuing from the next position after the string. If no match can be found, the character is printed as is and the encoder continues from the next position.

Dictionary references are a two byte integer, the first byte of each reference has an 0x80 (128) offset added to it. This is done as any ASCII file will contain only bytes in the range 0x0-0x7F (0-127), this is useful as the decoder knows that any byte outside of this range must be the first byte of a dictionary reference. Decoding thus consists of reading the encoded string byte by byte, detecting dictionary references, replacing them with the referenced string and leaving any non-reference bytes as they are. However, when initially developing this compression step I wasn't aware that the files would only contain ASCII bytes. My plan for this was to change the offset to 0x81 and use 0x80 bytes to flag any "natural" non-ASCII bytes in the input. In this way, this step of compression can be applied to files with arbitrary contents. All-in-all, my implementation allows for a dictionary containing $128 \times 256 = 32,768$ strings and achieves a reasonable compression on long LaTeX files, able to compress long commands to just two bytes.

2 Arithmetic PPMC

After experimenting with a lot of compression algorithms and different orders of application, I found that none were able to compete with applying PPM immediately after the first step of dictionary compression. Although it was quite difficult, I was able to create an implementation for arithmetic PPM. One thing I struggled with in particular was dealing with underflow from the adaptive arithmetic encoder and decoder, from the lecture and textbook the adaptation of the encoder (involving shifting bits and incrementing a counter etc.) was clear. But what was difficult to find was how this affected the decoder (it isn't mentioned at all in the textbook), after a decent amount of research as well as trial and error I found a method that worked, whenever the second bit of L is 1 and the second bit H is 0, the second bit of the input buffer is inverted and the third bit is shifted out.

As there are many different versions and implementations of PPM, I spent a long time experimenting to find the method that worked best. First, I compared using an arithmetic encoder and a Huffman encoder. I implemented both methods and ran them on several files to get a measure of average performance. The results below are measured post dictionary compression (as this is the performance that matters most) unlesss stated otherwise.

Method	Average Bits Per Character (bpc)
Arithmetic	2.073
Huffman	2.238

Method	Average bpc
PPMA	2.388
PPMB	2.162
PPMC	2.073

N	Average bpc
3	
4	2.073
5	
6	

The next things to test were the various methods of handling escape character frequency (tested with the Arithmetic PPM version). Finally, all that was left to determine was the context length to use. Interestingly, for uncompressed inputs $N = 5$ performed best but on inputs already compressed by the dictionary (Idea 1), $N = 4$ performed best. The likely explanation for this is that after the first

step of compression, any contextual redundancies in the original file have been shrunk so a shorter context bound is more appropriate.

3 PPM Additions

Beyond the above experiments, I also added some extensions in an attempt to make the compression ratio of PPM even better. The first and second techniques take advantage of the information about the next character implicitly transmitted by the escape characters. Whenever a context is escaped from, the decoder knows that the next character cannot be any character that has ever appeared after that context (as if it were, the context wouldn't have been escaped from), therefore these characters are considered "excluded" while encoding in lower contexts. Excluded characters are completely ignored when calculating character probabilities, offering a "free" improvement to compression ratio. As an example consider the dictionary below, suppose the current context is "abb" and the symbol being compresed is "a". The context "abb" would be escaped from (encoding *esc* with probability $\frac{2}{5}$) without exclusion "a" would be encoded from context "bb" with probability $\frac{1}{6}$. With exclusion, the escape character is encoded in the same way but when encoding "a" from context "bb" both "b" and "c" are excluded, meaning "a" can be encoded with probability $\frac{1}{4}$, a significant improvement. As an additional step, if a context has never been seen before (has no children in the trie) or all its children are excluded, no escape character is encoded (as escaping is the only option).

Context	Frequency
$abb \rightarrow c$	2
$abb \rightarrow b$	1
$abb \rightarrow esc$	2
$bb \rightarrow c$	2
$bb \rightarrow b$	1
$bb \rightarrow a$	1
$bb \rightarrow esc$	3

One limit of PPM is the issue of using larger contexts. One may think the larger the context the better the compression (as larger contexts will detect more niche patterns in the input). However this is usually not the case, large contexts (say $N = 15$) are very unlikely to appear in the same way followed by the same character meaning that many escape characters need to be printed to get down to a context short enough to encode the character. An idea I had to get around this was to encode a character for each context rather than encode escape characters. To do this I stored an $N + 2$ length array storing the frequencies with which each context is used ($N + 2$ to include order 0 and order -1), in this method a character's context could be encoded via a single call to the adaptive arithmetic encoder using this array as character frequencies. My theory behind this was that it would allow the use of larger contexts with little drawback, for example in the standard implementation if $N = 15$ and a character is encoded with order 1, 14 escape characters need to be encoded but in my implementation this can be done with one a single character. However in practice the compression ratio was slightly worse than the standard implementation. Although for large contexts it did perform better than the standard PPM implementation, it was never able to compete with standard PPM and small N.

4 Using a Static Initialiser Dictionary for PPM

One aspect of PPM and other adaptive dictionary based methods is that they become more effective as they process a file. For example early on in the file when certain characters or contexts have not been seen before, a lot of bits will be used for escape characters or order -1 contexts but later, on as patterns arise in the dictionary, the compression becomes very effective. In an attempt to achieve this behaviour from the start of a file I created an initial static dictionary. To create this dictionary, I used several latex files as well as regular text files and compounded the changes into a single dictionary

(note: the input files were first processed with the LaTeX dictionary encoder described in idea 1 as these are the contextual redundancies I wanted the dictionary to learn). The dictionary is then loaded at the start of encoding/decoding and from there the program functions as standard PPMC. The initial dictionary is still built on throught the encoding/decoding process so any patterns unique to the input file are still taken advantage of. This technique provides a surprising improvement over an uninitialised implementation and when tested on files not used to create the initial dictionary, encoded with an average of 2.018*bpc*.

5 Burrows-Wheeler Transform via Bit Stuffing

An interesting technique I experimented with was the use of the Burrows-Wheeler Transform (BWT) to create contextual redundancy in a file. When implementing the transform the first issue I an into was the inefficiency of the standard implementation, the algorithm uses $O(n^2)$ space and time which is not feasible for large files (due mostly to the memory usage), fortunately I found Chrochemore et al.'s paper [1] describing and $O(1)$ space algorithm for the BWT which made it feasible for me to implement. The second issue I ran into was that of the EOF byte. For the BWT to be reversible it is necessary that an EOF byte be present at the end of the file, the EOF byte must be strictly less than all other bytes in the file. As it only made sense to run the transform after the first step of LaTeX dictionary encoding, the input file could include any byte meaning that generating an EOF byte is no simple matter. To solve this I used bit stuffing, a technique I picked up from last year's Netowkr and Systems module. Every sequence of 7 1s in a row has a 0 appended to it, this is reversible and ensures that no sequences of 8 1s in a row ever appears, meaning that there are no 0xff bytes. Once this is achieved, all bytes can be incremented without overflow and 0x00 can be used as the EOF byte. I used bit stuffing over other techniques (such as using flag bytes) as the increase in file size it creates is next to nothing.

6 Other Compression Techniques

Beyond PPM I experimented with a multitude of other compression techniques as well as different orders of application. I experimented with several algorithms given in lectures as well as a few I found through my own research. In order to adequately take advantage of the contextual redundancy generated by the BWT, I implemented and compared LZ77, LZ78, LZW, LZFG, and the Run-Length Encoding technique used in MNP5.

Method	bpc	bpc (post BWT)
MNP5	5.065	3.305
LZ78	3.332	3.095
LZW	3.204	3.175
LZFG	3.535	3.152
PPMC	2.073	2.447

Interestingly, the BWT increased the effectiveness of every compression technique except for PPMC (which it actually made worse). Although BWT does create patches of repeated symbols, much of the document consists of blocks from which these symbols are taken meaning they are more random and are less compressable. For certain (less "aggressive") compression methods this still results in an improvement, but for more aggressive methods (such as PPM) the contextual redundancy created by these repeated patches is not enough to surpass the randomness of the regions described above. I feel this theory of less aggressive methods being able to take full advantage of BWT is substantiated by LZFG's significant increase in performance. LZFG is the least aggressive of the Lempel-Ziv variants I implemented, being able to print chunks of the input file as an uncompressed stream. This means that LZFG is able to take advantage of these repeated patches (as any good compression algorithm should be able to) but doesn't suffer too much when dealing with the more random areas.

Python Packages

All necessary packages can be installed directly through pip. All packages are listed below:

1. numpy
2. math
3. sys
4. pickle (install via $pip install pickle-mixin)
5. bitstring

References

[1] Maxime Crochemore et al. "Computing the Burrows–Wheeler transform in place and in small space". In: *Journal of Discrete Algorithms* 32 (2015), pp. 44–52.

[2] David Salomon. *Data compression: the complete reference*. Springer Science & Business Media, 2004.