Data Compression Class Project

This is a report for the Data Compression Class Project from the Topics on Algorithms Spring 2018 Class by Seok Won Lee and Chae Min Ahn.

Repository: https://github.com/ijleesw/data-compression

How to Run

Run `test.sh` in terminal. If malloc error occurs in tunstall, run `tunstall.sh`.

All code has been copied from existing github repositories and modified. The code for Tunstall Code, Golomb Code, Arithmetic Coding and LZ78 is implemented using C, whereas Static Huffman Code, Adaptive Huffman Code, LZ77, LZSS and LZW is implemented using Python 3.6.1. All code was tested on a macOS Sierra 10.12.5 operating system.

The file `test.sh` will execute a comprehensive run of all algorithms and compute compression ratios. Please check the contents of this file for commands to run individual algorithms.

Static Huffman Coding

1. Review of Algorithm

Static Huffman scans the text content to calculate the frequency of each symbols, then builds a binary tree according to these frequencies. A parent node has a value equal to the sum of its children's frequencies, and thus the root node has a value equal to the total frequency of all characters. Then each leaf node is represented with its path, as followed down from the root node, adding a '0' for a left child and '1' for a right child.

2. Description of Code

Static Huffman will execute a series of operations `s.make_frequency_dict()`, `s.make_heap()`, `s.merge_nodes()` and finally `s.make_codes()` to get the code representation of all symbols.

Adaptive Huffman Coding (FGK)

1. Review of Algorithm

Adaptive Huffman uses the same underlining idea, but builds the tree as each character is encountered. So instead of traversing the data twice, once for frequency calculation and once to convert each character to code, only one traversal is required.

2. Description of Code

Each character `s` is passed onto `get_code(s)` to get current path representation of the character, then the tree is updated, which may result in `insert()` or `swap_node()`.

Golomb Code

1. Review of Algorithm

Golomb code computes the probability *p* of zeros over all bit stream. Then using the notation from the lecture slides *m* is computed, and the quotient converted into unary representation. The remainder goes through the same process as on the page 22 on lecture 5 slides.

2. Description of Code

The probability *p* is computed using `num_set_bits()` in `golomb_encode()`. Then the text is encoded using `get_run_length_encoding()`, and each symbol is translated into unary code + minimal binary code.

Tunstall Code

1. Review of Algorithm

Given a symbol table and the probability of each symbols, the current leaf node with the highest probability is branched. This process is repeated until the number of leaf nodes exceed (2^k)-1, when using k-bit encoding. Then data is read from input source and corresponding lead node is returned.

2. Description of Code

A Tunstall Code instance is initiated first with `Tunstall t = Tunstall(8, 2)`, specifying the lookup table size and each encoded symbol be 8-bit. Then `t.getProbabilities()`, `t.createDecodingTables()`, `t.createEncodingTables()` are run in series to prepare lookup tables, followed by `t.compress()`.

Arithmetic Coding

1. Review of Algorithm

At each step interval [a, b) is divided according to each symbol's probability, and the interval associated with the next symbol is chosen for the next round. Starting from [0, 1), this process is repeated until the precision has reached its max limit.

2. Description of Code

Function `encode_BYTE` is in charge of whole compression process. Bytes are read until the end of input array, and `adaptive_model.encode_symbol` is run on each input. Within this function `adaptive_encoder.encode` takes place, which repeatedly divides 32 bit integers, sending each bit to `file_writer.send` to write on the output file.

LZ Family (LZ77/LZ78/LZSS/LZW)

1. Review of Algorithm

Lempel-Ziv algorithms encompass family of dictionary encoding techniques. Unlike Huffman Coding which attempts to reduce average amount of bits required to represent a symbol, LZ family algorithms attempt to replace a string of symbols with a reference to a dictionary location for the same string.

LZ77 encodes from a sliding window over previously seen characters, replacing reoccuring sequence with a pointer to its earlier occurrence. LZ78 builds a dictionary of phrases seen in the data, then replaces reoccuring pattern with its index number in the dictionary. LZSS is a variation of LZ78 algorithm, which ensures that the dictionary reference be shorter than the string it replaces. LZW keeps a dictionary between the longest encountered words and a list of code values.

2. Description of Code

LZ77 and LZSS is implemented very naively, and can take some time to run (half a second to a minute or two). `findSubstring()` is run with `searchBuffer` to look for previously seen patterns. LZ78 calls `wrapper_compress()` within `wrapper_exec()`, which then calls `wrapper_compress()` that compresses each byte from a fie and updates the dictionary. for each byte LZW looks up the codebook in `_encode_byte()`, and calls `_add_code(new_prefix + byte)` to update the codebook.

Dataset Preprocessing

For Tunstall code, Golomb code, arithmetic coding, LZ78 and LZW, datasets are treated as a bytestream. For other algorithms, datasets are treated as a stream of Latin-1 encoded characters.

Experimental Results

Each data and result set is listed with alphabet size, file size, and actual file size. For the provided datasets alphabet size is the supposed number of symbols provided on the project handout pdf, and the numbers inside the bracket indicate the actual number of symbols occuring in each of dnaby, englishby, xmlby files our team was given.

The actual file sizes were calculated using `file_size / 8 * ceil(log2(alphabet_size))`. Compression ratio is the percentage of compressed fie size over original file size. Algorithm to achieve the best compression ratio for each input file is marked with **boldface**.

Compression Algorithms :

		SD1	SD2	SD3	SD4	DNA	ENGLISH	XML
Dataset	alphabet size	255	64	16	4	16 (12)	239 (96)	97 (92)
	file size	32768	131072	524288	2097152	524288	4194304	1048576
	actual size	32768	98304	262144	524388	262144	2670016	917504
Tunstall	alphabet size	256	253	241	253	99	185	174
	file size	33289	125272	270677	526409	171596	3512202	982146
	actual size	33289	125272	270677	526409	150146	3512202	982146
	ratio	101.590	127.433	103.255	100.405	57.276	95.700	107.045
Golomb	alphabet size	252	255	228	100	84	241	234
	file size	38322	131322	437387	1310590	481347	4629271	1217329
	actual size	38322	131322	437387	1146766	421178	4629271	1217329
	ratio	116.949	133.588	166.850	218.728	160.667	126.138	132.678
Arithmetic	alphabet size	256	256	256	256	256	256	256
	file size	32850	98591	262535	524756	130571	2357989	684756
	actual size	32850	98591	262535	524756	130571	2357989	684756
	ratio	100.250	100.292	100.149	100.089	**49.809**	64.250	74.632
LZ77	alphabet size	256	65	17	11	21	105	101
	file size	98298	391575	1472910	3205596	736191	11523537	2783274
	actual size	98298	342628	920568	1602798	460119	10083094	2435364
	ratio	299.982	348.539	351.169	305.709	175.521	274.743	265.434
LZ78	alphabet size	256	256	256	256	256	256	256
	file size	46158	133283	319251	604808	145009	2112228	366741
	actual size	46158	133283	319251	604808	145009	2112228	366741
	ratio	140.863	135.582	121.785	115.358	55.317	**57.554**	**39.972**
LZSS	alphabet size	256	65	17	11	22	106	102
	file size	66816	280523	1245167	4011153	926897	9601214	2201309
	actual	66816	245457	778229	2005576	579310	8401062	1926145
	ratio	203.906	249.692	296.871	382.533	220.989	228.911	209.933
LZW	alphabet size	256	256	256	256	256	256	256
	file size	44833	136804	328359	624223	149529	2228515	412225
	actual size	44833	136804	328359	624223	149529	2228515	412225
	ratio	136.819	139.164	125.259	119.061	57.041	**60.722**	**44.929**
Static Huffman	alphabet size	256	256	256	256	216	256	256
	file size	32749	98306	262146	524290	144856	2374710	688748
	actual size	32749	98306	262146	524290	144856	2374710	688748
	ratio	**99.942**	**100.002**	**100.001**	**100.000**	55.258	64.706	75.068
Adaptive Huffman	alphabet size	n/a	n/a	n/a	n/a	n/a	n/a	n/a
	file size	33093	98636	266266	589817	144873	2374877	688900
	actual	33093	98636	266266	589817	144873	2374877	688900
	ratio	100.992	100.338	101.572	112.499	55.265	64.710	75.084

Out of all compression algorithms, the Static Huffman Code seems to work best with the randomly generated SD files. This is possibly due to the fact that the symbols in these files are uniformly distributed, so the Static Huffman is able to build a perfectly balanced binary tree with smallest possible average bit per symbol. The same applies to Adaptive Huffman Code, with close to 1:1 compression ratios. No other algorithm is able to achieve less than 100% compression ratio, not even the built-in compressors, since these files are fairly small and its distribution uniform with no obvious patterns.

As for DNA file, the Arithmetic Coding is an obvious winner with compression ratio 49.809. Arithmetic coding works best with data with biased distribution, and out of all 12 symbols that ever appear in the file the A, C, G, T occur much more frequently due to the special property of DNA data. This is also the best result including built-in compressors.

The LZ family algorithm generally perform well on ENGLISH and XML files, since these data tend to have abundance of reoccurring patterns. LZ78 achieves the best compression ratio on the two files, followed by LZW as the runner up. We have expected LZW to show better results, and are suspecting that LZ78 performs better than LZW in this case primarily due to the quality of implementation. For ENGLISH and XML data the built-in compressors are able to achieve much better compression ratios. These built-in compressors commonly used in Linux show solid performance on fairly large data with certain patterns.

Here are some further discussion on individual compression algorithms:

Since Tunstall code compresses text by expanding trees successively, and the end code has to fit within 1 byte=8 bit, it is more efficient when the number of symbols is small. (Since more expansion is possible) Tunstall doesn't perform well with xmlby or englishby where the alphabet size is over 90 or with SD2 where alphabet size is 64. But with SD3 (alphabet size = 16), SD4 (alphabet size = 4) Tunstall produces compression rate equal to or better than that of the built-in compressors.

Assuming p is the probability that a zero takes place, either a small or large p is preferable for effective compression. When p is near 0.5, meaning uniform distribution, almost no compression takes place. Sometimes the compressed files even get bigger. For example, the average bit rate from page 22 of lecture 5 computes up to 1.53 using p=0.5. Large p is needed for large m, and compression takes place only then.

Arithmetic coding works best with data with biased distribution and suffers from high number of symbols since the intervals divide into too tiny sections. Arithmetic shows better performance than the built-in compressors with dnaby, SD3 and SD4 where the alphabet size is small, but not with enlishby and xmlby (alphabet size > 90).

LZ family generally show terrible performance on uniformly distributed data, since they are pattern-dependent. They achieve the worst comperssion ratio recording up to 300%, and this is because each token is represented as either `(0, 0, a)` or `(0, a)`. Without reoccuring patterns they rather enlarge files, instead of reducing average bits needed to represent each symbol.

Builtin Compressors :

		SD1	SD2	SD3	SD4	DNA	ENGLISH	XML
Dataset	alphabet size	255	64	16	4	16 (12)	239 (96)	97 (92)
	file size	32768	131072	524288	2097152	524288	4194304	1048576
	actual size	32768	98304	262144	524388	262144	2670016	917504
gzip	alphabet size	256	256	256	256	256	256	256
	file size	32811	99286	299019	613511	150118	1612583	172550
	actual size	32811	99286	299019	613511	150118	1612583	172550
	ratio	100.131	100.999	114.067	117.018	57.265	43.939	18.806
bzip2	alphabet size	256	256	256	254	256	256	256
	file size	33231	999140	267044	571571	139452	1214487	118110
	actual size	33231	999140	267044	571571	139452	1214487	118110
	ratio	101.413	100.850	101.869	109.019	53.197	**33.092**	**12.873**
xz	alphabet size	256	256	256	256	256	256	256
	file size	32828	100456	272100	573220	132956	1267892	134852
	actual size	32828	100456	272100	573220	132956	1267892	134852
	ratio	100.183	102.189	103.798	109.333	**50.719**	34.547	14.698

References

- Arithmetic Coding : https://github.com/ldematte/arithmetic_coding
- Tunstall Code : https://github.com/cnr-isti-vclab/corto
- Golomb Code : https://github.com/anirudhvr/golomb-coding
- Static Huffman Code: https://github.com/bhrigu123/huffman-coding
- Adaptive Huffman Code: https://github.com/sh1r0/adaptive-huffman-coding
- LZ77, LZSS: https://github.com/laz08/dictionary-encoding
- LZ78 : https://github.com/evilaliv3/lz78
- LZW: https://github.com/joeatwork/python-lzw