Image Data Compression using LZSS dictionary method

Author: Pavel Kratochvíl

Introduction

This project implements the LZSS algorithm for lossless data compression and decompression. It leverages a sliding window and an efficient longest-match search strategy based on a hash table and linked list structures to replace redundant data sequences with compact coding and uncoding tokens.

Lempel-Ziv-Storer-Szymanski (LZSS) is a dictionary-based lossless data compression algorithm derived from LZ77. Its fundamental principle is to replace occurrences of data sequences with references to previous identical sequences found within a sliding window of limited length. When an input sequence is encountered that has appeared recently, it is encoded as a pair of values: an `offset` indicating how far back the previous occurrence starts, and a `length` indicating math length in bytes. If a sequence has not appeared recently (or the match is too short), the current character (byte) is outputted as uncoded. This process effectively reduces redundancy in the input data and when combined with preprocessing of the input sequence of bytes, it achieves significant results.

Program Arguments and Functioning Modes

On top of the mandatory arguments in the assignment, the submitted project also implements these optional but useful arguments for experimentation with encoding parameters without the need to recompile the project:

- `--block_size <N>`: (Optional, default=16, max=$2^{16} - 1$) Sets the block size to use in case of adaptive mode. Applies only in compression mode.
- `--offset_bits <N>`: (Optional, default=8, max=15) Sets the number of bits to use in the offset part of a coding token. Applies only in compression mode.
- `--length_bits <M>`: (Optional, default=10, max=15) Sets the number of bits to use in the length part of a coding token (maximum length of prefix match). Applies only in compression mode.
- `--help`: Prints information about the program and its arguments.

The default values can be altered in `common.hpp`. All of the aforementioned arguments are optional. The information about the image width, image height, block size (if adaptive mode is used), adaptive mode strategy for a block (if adaptive mode is used), offset bits, length bits, adaptive mode and model usage is saved into the encoded output, therefore all of the arguments apart from the input and output file arguments are ignored during decompression.
For example, for decoding file compressed with these arguments:

```
./lz_codec -c -i input.raw -o output.enc -w 512 -m -a -block_size 32 -offset_bits 6
```

These decompression arguments are sufficient:

```
./lz_codec -d -i output.enc -o output.dec
```

In `common.hpp`, the compile time variable `MTF` switches between delta encoding and MTF models for transformation. By default delta encoding is used (`MTF 0`), since the MTF model performed only marginally better at the cost of longer runtimes.

Table 1: Average Performance: Delta Encoding vs. MTF

	Delta Encoding	MTF
Average bits per pixel	2.1290	2.4175
Average compression time (s)	0.0916	0.0908s
Average decompression time (s)	0.0065	0.0068

Table 2: Average over all benchmark images and over all combinations of flags configuration (baseline, `-a`, `-m` and both `-a -m` for delta encoding and MTF models.

Algorithms and Data Structures

I opted for LZSS encoding algorithm for its effectivity for encoding repetitive sequences and adjustable parameters such as search buffer size and look-ahead buffer size, which can enhance the compression efficiency. For the prefix lookup, a **hash table** was implemented (class `HashTable` in `hashtable.cpp`). After advancing the sliding window, the first `MIN_CODED_LEN` (default=3) characters of new prefixes are hashed and saved into the hash table (default size=1<<12 entries) as a **linked list** node (struct `HashNode` in `hashtable.hpp`). The hashing function utilizes Knuth's Multiplicative hashing with the multiplicative constant being the fractional part of the golden ratio multiplied by 2^{32} [1].

The compression and decompression works for **images of arbitrary width and height**. The height is calculated as `total_size / width`, if the `total_size` is not divisible by the width, an error is thrown. Noteworthy consequence is that the program can be used for **compression and decompression of arbitrary binary file or text file** (up to 4096MB in size due to the use of `uint32_t` for width and height) by launching it with argument `width=1` (although it does not make sense to combine this with adaptive mode).

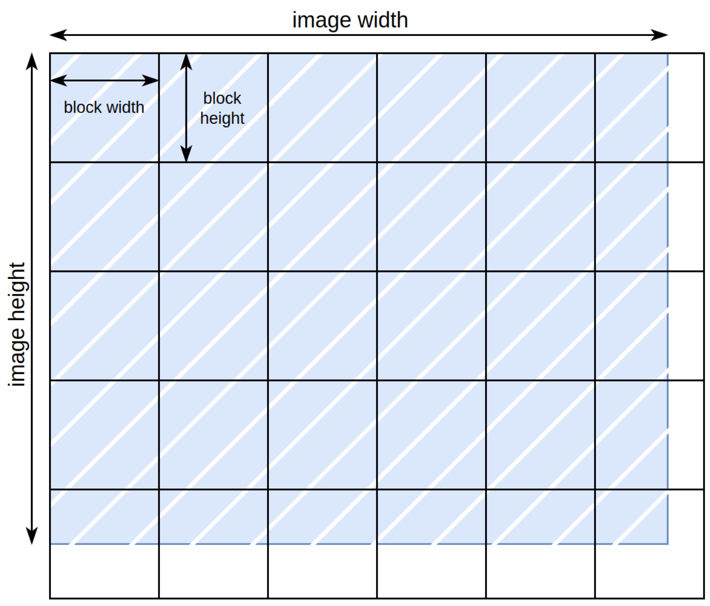

Figure 1: Division of image data into blocks. If the image height or width is not divisible by the block size, last blocks are not filled completely and the information about width and height of a block is saved and reconstructed during decompression.

Compression and Decompression steps

Order of Compression Steps:

1. **Reading input file:** The file is read into memory and the total size is checked with the supplied width (`-w` argument). If the total is size is not divisible by the width, an error is thrown.
2. **Division into blocks:** The blocks vector contains either a single element (non-adaptive mode) or can contain multiple blocks for larger input files (adaptive mode).
3. **Block serialization:** If the adaptive mode is enabled, every block is serialized horizontally and vertically.
4. **Model transformation:** If the model flag (`-m`) was used, the data in each block is transformed using the selected model.
5. **Block encoding:** All blocks are encoded using LZSS. If the adaptive mode is enabled, the best performing serialization strategy is picked.
6. **Decision based on the achieved compression:**
   - **If** the file size after compression if smaller than the original size, the file is written in compressed format (tokens) with the first byte indicating successful compression (`0x01`).

- **Otherwise** the original file is copied bit-to-bit with a single byte at the beginning indicating unsuccessful compression (`0x00`).

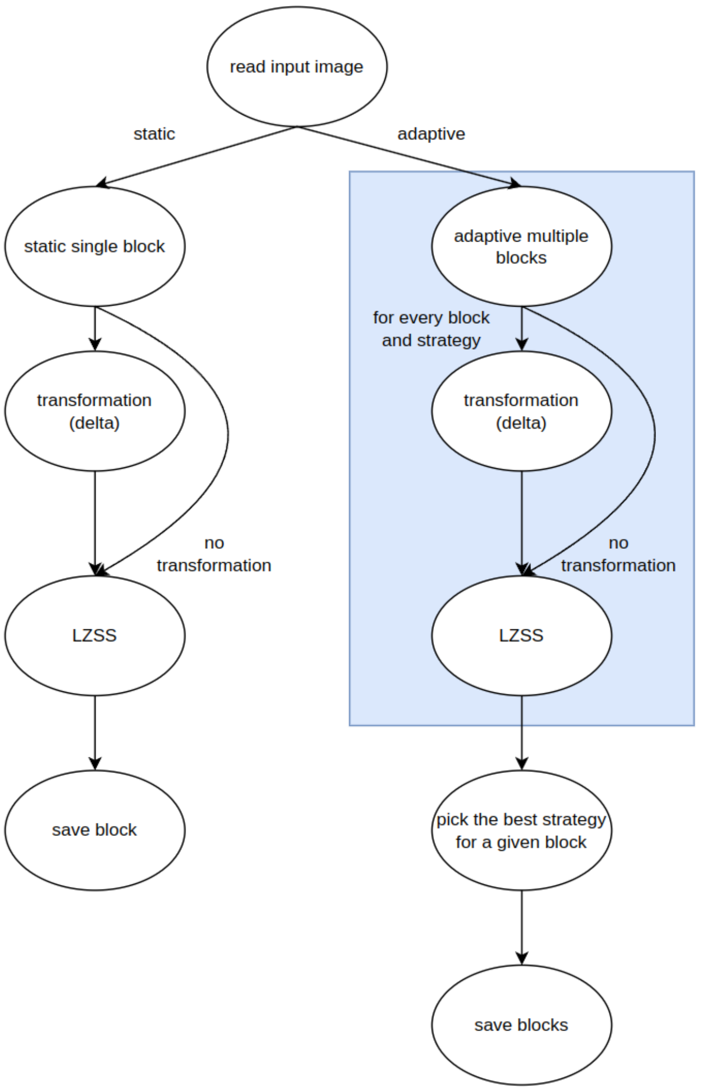

Figure 2: Visualization of the order of compression steps.

Order of Decompression Steps:

1. **Check for unsuccessful compression:** If the first byte of the input file indicates unsuccessful compression (`0x00`), the rest of the file is copied bit-to-bit to the output file and program exits.

2. **Reading input file**: The input file is read, the information about the width, height, offset bits, length bits, model and adaptive mode is parsed and saved. Then for each block, if the adaptive mode was used during encoding, serialization strategy is parsed from the input and saved to block along with tokens for the block.

3. **Block decoding:** Each block's contents (tokens) are decoded into bytes of data.

4. **Block de-serialization:** If the adaptive flag (`-a`) was used during compression, the data in each block is de-serialized using the strategy used during its compression.

5. **Image composing:** The final decoded, reverse transformed and de-serialized data from each block is composed into the output image.

6. **Image writing:** The data is saved into the decoded output file.

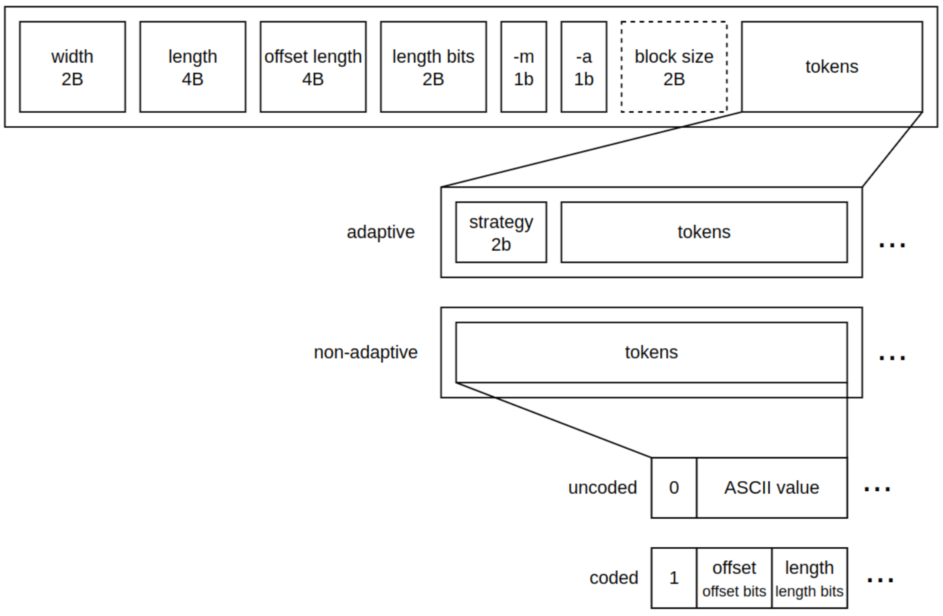

Figure 3: The structure of the compression output file, which contains the header with necessary information for decoding and blocks of tokens. The block size is present only if adaptive mode is enabled. Likewise, if adaptive mode is enabled, the serialization strategy of individual blocks is saved into each block. Each block then contains tokens which can be either in either coded or uncoded format. If the compression was not successful (compressed size $>$ original size), the first byte is zero (0x00) and the rest of the file is the original input file.

Results summary

Some of the added features added some computational and memory overhead but at on the other hand, the final implementation allows for playful experimentation with parameters and tuning the configuration to the best possible outcome for a particular file.

All of the results were measured on the reference machine (merlin.fit.vutbr.cz). Please note that the measurements vary by a large margin from run to run depending on the current utilization of the reference machine.

- **No Flags (Baseline):** Achieves high compression on some files (e.g., `cb.raw` and `df1v.raw` saving nearly all space), but fails on others (e.g., `df1h.raw`, `nk01.raw`). Average compression time is 0.0346s, and average decompression time is very fast at 0.0030s.
- **-a Adaptive Mode:** Often results in worse compression ratios than the baseline for highly compressible files (e.g., `cb.raw` 6156 bytes vs 703 bytes baseline), but successfully compresses `df1h.raw` unlike the baseline. Average compression time (0.0250s) is faster than baseline, while decompression (0.0042s) is slightly slower. Fails on `nk01.raw`.
- **`-m` Model Preprocessing:** Yields the best compression overall for most files, significantly reducing sizes (e.g., `cb.raw` 773 bytes, `df1h.raw` 907 bytes), but has the slowest average compression time (0.0756s). Decompression remains fast (avg 0.0034s), and `nk01.raw` still fails.
- **-a `-m` Adaptive Mode and Model Preprocessing:** Generally offers better compression than -a alone (e.g., `cb2.raw` 83% saved vs 73%), but is worse than `-m` alone for highly compressible files. Average compression time (0.0276s) is faster than `-m` and baseline, while decompression speed (avg 0.0045s) is similar to -a. Fails on `nk01.raw`.

Table 3: Entropy per File (bits per byte)

File	Entropy (bits per byte)
cb.raw	1.00
cb2.raw	6.90
df1h.raw	8.0
df1w.raw	8.0
df1hvx.raw	4.514
nk01.raw	6.4729
shp.raw	0.8675
shp1.raw	1.8961
shp2.raw	1.87

Table 4: Benchmark Results: Baseline vs. Adaptive (-a)

	Baseline			Adaptive (-a)		
File	Comp. (s)	Decomp (s)	bpp	Comp. (s)	Decomp (s)	bpp
cb.raw	0.176s	0.004s	0.02	0.038s	0.006s	0.18
cb2.raw	0.023s	0.008s	2.73	0.076s	0.010s	2.21
df1h.raw	0.019s	0.003s	8.00	0.059s	0.007s	0.64
df1v.raw	0.145s	0.004s	0.09	0.057s	0.007s	0.64
df1hvx.raw	0.044s	0.007s	1.10	0.052s	0.007s	1.59
nk01.raw	0.019s	0.003s	8.00	0.060s	0.003s	8.00
shp.raw	0.128s	0.004s	0.11	0.054s	0.008s	2.14
shp1.raw	0.068s	0.007s	2.06	0.058s	0.008s	2.02
shp2.raw	0.073s	0.008s	2.47	0.054s	0.008s	1.91

Table 5: Benchmark Results: Model (-m) vs. Adaptive + Model (-a -m)

	Model (-m)			Adaptive + Model (-a -m)		
File	Comp. (s)	Decomp (s)	bpp	Comp. (s)	Decomp (s)	bpp
cb.raw	0.340s	0.004s	0.02	0.039s	0.005s	0.20
cb2.raw	0.094s	0.007s	1.75	0.040s	0.006s	0.33
df1h.raw	0.510s	0.005s	0.01	0.045s	0.007s	0.32
df1v.raw	0.254s	0.005s	0.09	0.040s	0.007s	0.33
df1hvx.raw	0.063s	0.005s	0.76	0.052s	0.007s	1.17
nk01.raw	0.018s	0.003s	8.00	0.065s	0.003s	8.00
shp.raw	0.086s	0.005s	0.12	0.055s	0.009s	2.21
shp1.raw	0.052s	0.008s	2.41	0.069s	0.009s	2.08
shp2.raw	0.061s	0.009s	2.83	0.068s	0.010s	1.97

Table 6: Comparison of Compression Methods

	Baseline	Adaptive	Model	Adaptive+Model
Compression time (s)	0.0772s	0.0564s	0.1642s	0.0525s
Decompression time (s)	0.0053s	0.0071s	0.0056s	0.0070s
Bits per pixel (bpc)	2.7334	2.1522	1.7803	1.8497

References

[1] Thomas Wang. *Integer Hash Function*. Archive.org, 1997. URL: `https://web.archive.org/web/20071223173210/http://www.concentric.net/~Ttwang/tech/inthash.htm` (visited on 04/03/2025).