Lossless Compression and the PAQJP 8.2 Compressor with 256 Fully Reversible Transform; the code was done on 12 February, but the explanation was done on 16 February.

Introduction

Lossless data compression is a fundamental technique in computer science that reduces the size of data without losing any information. The original data can be perfectly reconstructed from the compressed representation. This is essential for applications where every bit matters—archiving, text compression, executable files, and many scientific datasets. Over the decades, numerous algorithms have been developed, from simple run-length encoding to sophisticated statistical models like those used in the PAQ family of compressors.

The Python script provided, PAQJP_8.2_LOSSLESS_256_TRANSFORMS, implements a unique approach: it applies one of 256 possible reversible transformations to the input data and then compresses the transformed data using either the PAQ compressor or Zstandard (zstd). The transformation that yields the smallest compressed output is chosen. All 256 transforms (numbered 0 through 255) are guaranteed to be 100% lossless—meaning that applying the transform and then its inverse restores the original byte sequence exactly. Transform 255 is the identity transform, which leaves the data unchanged, serving as a safe fallback.

This essay explains the concepts behind lossless transforms, details each of the 256 transforms used in PAQJP, describes the PAQ and Zstandard compression backends, and walks through the compression and decompression pipeline. It also discusses the self-test mechanism that verifies the lossless property for all transforms, and the practical implications of such a design.

The Role of Transforms in Lossless Compression

Before compressing data, it is often beneficial to apply a reversible transformation that rearranges or modifies the data to make it more amenable to compression. Classic examples include the Burrows-Wheeler Transform (BWT), which groups similar characters, and Move-To-Front (MTF) coding, which exploits local frequency. These transforms do not compress by themselves; they reorder the data so that subsequent compression algorithms (like Huffman coding or arithmetic coding) can achieve better ratios.

In PAQJP, the set of transforms is much simpler and more numerous. They are lightweight operations—XOR with constants, bit rotations, addition/subtraction of position-dependent values, substitution cyphers, and even a custom run-length encoding transform (transform 00). Each transform is designed to be easily invertible, and many are involutory (self-inverse). The idea is that for any given input, at least one of these transforms will “flatten” the data enough that the backend compressor (PAQ or zstd) can achieve a higher compression ratio than on the raw data.

Because the transforms are applied before compression, the compressed file must store a marker (0-255) indicating which transform was used, so that decompression can apply the inverse transform after decompressing the data. The marker is a single byte at the beginning of the compressed file.

Overview of PAQJP 8.2 Code Structure

The script defines a class PAQJPCompressor that contains:

· Helper methods for bit manipulation (_append_bits, _read_bits).

· A collection of 16 explicitly defined transforms (0-15) with custom logic.

· A dynamic transform generator for transforms 16-254, which simply XORs each byte with a seed derived from the transform index and the data length.

· Transform 255 as the identity.

· Optional backends: PAQ (if the paq module is installed) and Zstandard (if zstandard is installed). If neither is available, the backend falls back to storing the data raw (prefixed with 'N').

· A self-test method that verifies every transform on empty data, all 256 single-byte values, and random short data, as well as the full pipeline on 100 random inputs.

· Public methods compress and decompress that read/write files and handle errors.

The core compression logic is in compress_with_best: it iterates over all 256 transforms, applies each to the input data, compresses the result with the best available backend, and keeps the smallest compressed payload. The chosen transform’s marker is prepended.

Decompression reads the marker, extracts the payload, decompresses it with the backend, and then applies the inverse transform corresponding to the marker.

Detailed Examination of the 256 Transforms

Transforms 0-15: Custom, Explicitly Coded

These transforms are implemented with specific algorithms, each carefully crafted to be reversible.

· Transform 0 (multi-pass shift + compact RLE):

This transform attempts to find a sequence of additive shifts (0-255) that maximises runs of identical bytes. It applies up to 10 passes, each time choosing the shift that produces the longest runs (scored by the sum of squares of run lengths). After shifting, it encodes the data with a custom run-length encoding (RLE) that uses variable-length codes: 2-bit prefixes indicate short runs (1, 2-5, 6-12) and a special 4-bit code (1111) indicates longer runs (13-268). The encoder also stores the shift value for each pass. The decoder reads the RLE stream bit-by-bit, reconstructs the runs, and then subtracts the shifts in reverse order. The code includes careful bit-level reading to avoid “out of bits” errors, making it fully lossless.

· Transform 1 (involutory XOR with primes):

Iterates over a list of prime numbers less than 256, computes a derived XOR value for each prime, and repeatedly XORs every third byte with that value. Because XOR is its own inverse, applying the same transform again restores the original.

· Transform 2 (XOR with pattern):

Uses a pattern derived from the data length and checksum to XOR every fourth byte (starting at index 1). The pattern index is stored as the first byte, allowing the decoder to regenerate the same pattern.

· Transform 3 (bit rotation):

Rotates the bits of every fifth byte (starting at index 2) by a rotation amount (1-7) derived from the data length and sum. The rotation amount is stored as the first byte.

· Transform 4 (subtract/add position):

Repeatedly subtracts i % 256 from each byte (with modulo 256). The inverse repeatedly adds the same values.

· Transform 5 (rotate each byte):

Rotates every byte by a fixed shift (3 bits by default). The inverse rotates back.

· Transform 6 (substitution cypher):

Generates a random permutation of 0-255 using a fixed seed (42), then substitutes each byte. The inverse uses the inverse permutation.

· Transform 7 (XOR with length and π digits):

XORs each byte with the data length modulo 256, then repeatedly XORs with rotated digits of π (79, 17, 111). The same operations restore the original because XOR is involutory.

· Transform 8 (XOR with prime and π digits):

Similar to 7, but first XORs with a prime near the data length modulo 256.

· Transform 9 (XOR with prime, seed, π digits, and position):

Combines several sources: a prime, a seed from an internal table, rotated π digits, and the byte index. All XOR operations are self-inverse.

· Transform 10 (XOR with count of “X1” substrings):

Counts occurrences of the two-byte sequence b'X1', derives a value n, and XORs every byte with n repeatedly. The value n is stored as the first byte.

· Transform 11 (reverse order with Fibonacci and previous byte):

A more complex transform that processes bytes in forward order using Fibonacci numbers, position, and the previous transformed byte. The inverse processes in reverse order to undo the dependency.

· Transform 12 (XOR with Fibonacci numbers):

XORs each byte with a Fibonacci number (mod 256). Involutory.

· Transform 13 (XOR with nearest prime):

Computes several repeats (based on data length and sum), then iteratively finds nearest primes starting from the data length modulo 256. The final prime is XORed with all bytes. The repeat count is stored.

· Transform 14 (append metadata):

Appends a byte that encodes the number of bits to add (0-8) and a value derived from the last byte. The inverse simply removes the last byte. This is lossless because the appended byte is part of the transformed data and is later stripped.

· Transform 15 (add pattern to every third byte):

Adds a pattern (generated from an index based on data length) to every third byte. The index is stored, and the inverse subtracts the same pattern.

All these transforms are implemented with careful attention to reversibility. For example, transforms that use addition or subtraction take care to wrap modulo 256. Those that store parameters (like shift amounts, pattern indices) embed them as prefix bytes, ensuring the decoder has all the needed information.

Transforms 16-254: Dynamic XOR with Seed

For transforms 16 through 254, the code uses a generic approach:

seed = self.get_seed(n % len(self.seed_tables), len(data))

Then each byte is XORed with this seed. The get_seed method looks up a precomputed table of 126 rows (each 256 bytes) seeded with a fixed random seed. The seed value depends on the transform index and the data length, ensuring that different transforms produce different XOR masks. Because XOR is its own inverse, the forward and reverse functions are identical. This makes all 239 transforms trivially lossless.

Transform 255: Identity

The identity transform simply returns the data unchanged. It serves as a baseline: if no other transform yields a smaller compressed size, the compressor can fall back to storing the raw data (after backend compression). Note that there is no separate "raw" marker; transform 255 fills that role.

Compression Backends: PAQ and Zstandard

After applying a transform, the data is passed to a compression backend. The script supports two optional libraries:

· PAQ: A family of high-compression algorithms based on context mixing. PAQ achieves some of the best compression ratios in the world, but it is extremely slow. The paq Python module (if installed) provides bindings to the PAQ8o8 variant. Because PAQ is so slow, it is only practical for small files or when maximum compression is required.

· Zstandard (zstd): A modern compression algorithm developed by Facebook. It offers a wide range of speed/compression trade-offs. The script uses the highest compression level (22) via zstandard.ZstdCompressor(level=22). Zstd is much faster than PAQ and still provides excellent compression.

The backend selection is done in _compress_backend: it attempts to compress with both PAQ and zstd (if available) and chooses the smaller result. If neither is available, it simply stores the data raw, prefixed with b'N'. This ensures that the compressor always produces valid output, even without external libraries.

During decompression, _decompress_backend reads the first byte to determine which backend was used (L for PAQ, Z for zstd, N for raw) and invokes the appropriate decompressor.

The Compression Pipeline

1. Input: The user provides a file to compress.

2. Data reading: The entire file is read into memory.

3. Transform loop: For each transform marker from 0 to 255:

   · The corresponding forward transform function is called on the data.

   · The transformed data is passed to _compress_backend, which returns a compressed byte string (including backend marker).

   · The length of the compressed output is recorded.

   · If this length is smaller than the current best, the compressed payload and marker are saved.

4. Output: The chosen marker (one byte) is prepended to the best compressed payload, and the result is written to the output file.

Because all transforms are tested, the algorithm guarantees that the best among them (in terms of compressed size) is selected for that particular input. This is a brute-force approach, but given that transforms are cheap and there are only 256 of them, it is feasible for small to medium files. For large files, the overhead of 256 transformations and compressions might be significant, especially if PAQ is used.

Decompression Pipeline

1. Input: The compressed file is read.
2. Marker extraction: The first byte indicates which transform was used.
3. Backend decompression: The remaining bytes are passed to _decompress_backend, which returns the transformed data (the output of the forward transform).
4. Inverse transform: The inverse transform corresponding to the marker is applied to recover the original data.
5. Output: The original data is written to the specified output file.

The decompressor does not need to know which backend was used; that information is encapsulated in the payload (the backend's own marker byte). This design keeps the format simple.

Self-Test and Verification

A critical feature of PAQJP is the self-test routine (self_test). It verifies that every transform (0-255) is indeed lossless for:

· Empty input (except transform 0, which has special handling).

· Every single byte value (0-255) individually.

· Five random short sequences (1-100 bytes) per transform.

· The full compression/decompression pipeline on 100 random inputs (1-500 bytes).

If any test fails, the program reports the failure and exits. This rigorous testing ensures that no transform introduces bugs that would corrupt data. The identity transform (255) is also tested, confirming that it does nothing.

The self-test also demonstrates the importance of bit-exact reversibility. For example, the transform 0’s RLE decoder had to be carefully written to avoid off-by-one errors in bit consumption. The test catches such issues.

Why 256 Transforms? The Identity as Fallback

The choice of 256 transforms (0-255) is natural because a single byte can represent the marker. Having the identity transform (255) as one of them simplifies the design: there is no special case for “no transform”. The compressor always selects one of the 256, and 255 is always available. In practice, if none of the transforms improves compression, the identity transform will be chosen (unless a transform accidentally makes the data larger, in which case the minimal size will still be the raw data after backend compression). Since the backend itself may add overhead (e.g., the b'N' marker for raw storage), the identity transform might still produce a smaller payload if the backend compresses the raw data well.

Advantages and Limitations

Advantages:

· Guaranteed lossless: The self-test ensures that data integrity is never compromised.

· Adaptive: By trying all transforms, the compressor can adapt to different data patterns. For example, text files might benefit from transforms that add positional XORs, while binary files might respond well to bit rotations or RLE.

· Modular: Adding a new transform is straightforward; it must be included in the list and its inverse defined.

· Backend flexibility: Users can install PAQ, zstd, both, or neither, and the compressor still works.

Limitations:

· Speed: Testing 256 transforms means compressing the data 256 times (with the backend). For large files, this is prohibitively slow, especially with PAQ. The script is best suited for small files (kilobytes to a few megabytes).

· Memory usage: The entire file is read into memory and transformed multiple times. For very large files, this could exceed available RAM.

· Transform simplicity: Most transforms are very simple (XOR, add, rotate). They may not capture complex patterns as effectively as more sophisticated transforms like BWT or LZ. However, the brute-force approach compensates by sheer number.

· No streaming: The implementation processes the whole file at once, which is not suitable for streaming data.

Comparison with Other Preprocessors

Many compression tools employ preprocessing to improve compression. For example:

· gzip can optionally apply the --best flag, but it doesn't try multiple transforms.

· bzip2 uses the Burrows-Wheeler transform followed by MTF and Huffman coding.

· LZ4 and zstd have built-in dictionaries and can use training data.

PAQJP's approach is more akin to "meta-compression": it tries many simple transforms and picks the best. This is reminiscent of algorithms like PAQ8 itself, which uses a large number of models (contexts) to predict the next bit. However, PAQJP separates transformation from compression, allowing the backend to focus on statistical modelling.

Real-World Use Cases

Given its speed limitations, PAQJP is not intended for everyday use on large files. However, it could be useful for:

· Competitions: In lossless compression challenges where compression ratio is the only metric, trying many transforms might yield a slight edge.

· Research: Experimenting with different preprocessing techniques and understanding how they interact with strong backends like PAQ.

· Small critical data: When compressing tiny files (e.g., configuration files, short messages) where the overhead of trying all transforms is negligible, and every byte saved matters.

Conclusion

PAQJP 8.2 demonstrates an elegant idea: combine a large set of simple reversible transforms with a powerful backend compressor, and let the data choose the best transform. By ensuring every transform is lossless through exhaustive self-testing, the tool guarantees data integrity. The inclusion of the identity transform (255) provides a safe baseline. The optional use of PAQ and Zstandard gives users flexibility in speed/compression trade-offs.

While not practical for large-scale compression due to speed, the code serves as an educational example of how preprocessing can boost compression, and how to systematically verify reversibility. It also highlights the importance of bit-level care when implementing custom encodings (like the RLE in transform 0). For those interested in lossless compression, studying PAQJP's transforms and pipeline offers valuable insights into the building blocks of more sophisticated compressors.

Word count: approximately 3200.