Project compression Overview Quantum PAQJP_6.1

Already working and lossless

PAQJP_6 is designed to compress and decompress files efficiently by applying a series of reversible transformations to the input data before compressing it with either PAQ, zlib, or Huffman coding, depending on which yields the smallest output. The system supports multiple file types (JPEG, TEXT, EXE, DEFAULT) and uses two modes: **fast** (transformations 1, 3, 4, 5, 6, 7, 8, 9) and **slow** (all transformations 1–255). The transformations manipulate the input data to make it more compressible, and the system embeds a marker and file creation time in the compressed output for proper decompression.

The system uses the first three base-10 digits of π (3, 1, 4), mapped to base-256 (85, 28, 113), in some transformations to introduce pseudo-randomness and enhance compressibility. The project also includes a logging system for debugging and performance tracking.

Transformations (1–11, 12–255)

The transformations are a core component of the PAQJP_6 system, designed to preprocess the input data to improve compression ratios. Each transformation is reversible, ensuring lossless compression. Below is a detailed explanation of transformations 1–11 and the generalized approach for transformations 12–255.

Transformations 1–11

1. **Transform_01 (Prime XOR Every 3 Bytes)**:

   - **Description**: XORs every third byte of the input data with a value derived from prime numbers in the range [2, 255]. The XOR value is either the prime itself (for prime = 2) or `max(1, ceil(prime * 4096 / 28672))`.

   - **Parameters**: `repeat=100` (number of iterations).

- **Purpose**: Introduces variability using prime numbers to disrupt patterns, potentially making data more compressible.

- **Reverse**: Identical to the forward transformation (since XOR is its own inverse).

- **Example**: For input `data = b'abcdef'`, XOR every third byte (a, d) with a prime-derived value, repeated 100 times.

2. **Transform_03 (Pattern Chunk XOR)**:

- **Description**: Divides the input data into chunks of 4 bytes and XORs each byte in the chunk with 0xFF.

- **Parameters**: `chunk_size=4`.

- **Purpose**: Inverts bits in fixed-size chunks to alter data distribution, potentially aiding compression for certain patterns.

- **Reverse**: Identical to the forward transformation (XOR with 0xFF is self-inverse).

- **Example**: For `data = b'1234'`, output is `b'\xCC\xCD\xCE\xCB'` (each byte XORed with 0xFF).

3. **Transform_04 (Position-Based Subtraction)**:

- **Description**: Subtracts the index modulo 256 from each byte, repeated 100 times.

- **Parameters**: `repeat=100`.

- **Purpose**: Introduces position-dependent changes to break repetitive patterns.

- **Reverse**: Adds the index modulo 256 to each byte, repeated 100 times.

- **Example**: For `data = b'\x01\x02'`, first iteration yields `b'\x01\x01'` (subtract 0 from first, 1 from second).

4. **Transform_05 (Bit Rotation)**:

- **Description**: Rotates each byte left by 3 bits, with overflow bits wrapping around.

- **Parameters**: `shift=3`.

- **Purpose**: Shifts bit patterns to potentially align repetitive structures for better compression.

- **Reverse**: Rotates each byte right by 3 bits.

- **Example**: For `data = b'\x01'`, left rotate by 3 gives `b'\x08'`.

5. **Transform_06 (Random Substitution)**:

- **Description**: Applies a byte substitution based on a shuffled table generated with a fixed seed (42).

- **Parameters**: `seed=42`.

- **Purpose**: Randomizes byte values to disrupt local patterns while maintaining reversibility.

- **Reverse**: Applies the inverse substitution table.

- **Example**: For `data = b'\x01'`, maps to a random byte in the substitution table.

6. **Transform_07 (Pi and Size-Based XOR)**:

- **Description**: XORs each byte with a size-based value (data length modulo 256) and then with the mapped π digits (cycled through the data), with the π digits shifted based on data length modulo π length. The number of cycles is `min(10, max(1, int(data_size_kb)))`.

- **Parameters**: `repeat=100`.

- **Purpose**: Uses π digits and data size to introduce pseudo-random changes tailored to the input size.

- **Reverse**: Reverses the XOR operations and restores the π digit shift.

- **Example**: For `data = b'abc'`, XOR with size modulo 256, then with cycled π digits [85, 28, 113].

7. **Transform_08 (Pi and Nearest Prime XOR)**:

- **Description**: Similar to Transform_07, but uses the nearest prime to the data length modulo 256 instead of the size itself, followed by XOR with cycled π digits.

- **Parameters**: `repeat=100`.

- **Purpose**: Enhances randomness by incorporating prime numbers and π digits.

- **Reverse**: Reverses the XOR operations and π digit shift.

- **Example**: For `data = b'abc'`, XOR with nearest prime to length % 256, then π digits.

8. **Transform_09 (Pi, Prime, and Seed-Based XOR)**:

   - **Description**: XORs each byte with a combination of the nearest prime to the data length modulo 256, a seed value from a precomputed table, and cycled π digits XORed with the index modulo 256.

   - **Parameters**: `repeat=100`.

   - **Purpose**: Combines multiple sources of pseudo-randomness for complex data transformation.

   - **Reverse**: Reverses the XOR operations and π digit shift.

   - **Example**: For `data = b'abc'`, XOR with prime, seed value, and π digits XORed with index.

9. **Transform_10 (X1 Sequence-Based XOR)**:

   - **Description**: Counts occurrences of the byte sequence `b'X1'` (0x58, 0x31) in the input, computes `n = (((count * SQUARE_OF_ROOT) + ADD_NUMBERS) // 3) * MULTIPLY % 256`, and XORs each byte with `n`. Prepends `n` to the output.

   - **Parameters**: `repeat=100`, `SQUARE_OF_ROOT=2`, `ADD_NUMBERS=1`, `MULTIPLY=3`.

   - **Purpose**: Tailors the transformation to specific byte patterns in the data.

   - **Reverse**: Reads `n` from the first byte, XORs the remaining bytes with `n`.

   - **Example**: For `data = b'X1ab'`, counts one `X1`, computes `n`, XORs data with `n`.

10. **Transform_11 (Optimal y Addition)**:

    - **Description**: Tests values of `y` from 1 to 255, adding `(y + 1)` modulo 256 to each byte, repeated 100 times, and selects the `y` that yields the smallest PAQ-compressed output. Prepends the chosen `y` to the compressed data.

    - **Parameters**: `repeat=100`.

    - **Purpose**: Optimizes transformation by selecting the best `y` for compression.

    - **Reverse**: Reads `y` from the first byte, decompresses the data, and subtracts `(y + 1)` modulo 256.

    - **Example**: For `data = b'abc'`, tests multiple `y`, selects best, outputs `y` + compressed data.

Transformations 12–255

- **Description**: These transformations are dynamically generated and apply a size-based XOR operation. Each byte is XORed with `(size_mod + (i % 256)) % 256`, where `size_mod = (data_size % scale_factor) % 256` and `scale_factor` ranges from 2000 to 256000 based on data size.

- **Parameters**: `repeat=1000`.

- **Purpose**: Provides a large set of transformations to explore different data manipulations, increasing the chance of finding a highly compressible form.

- **Reverse**: Identical to the forward transformation (XOR is self-inverse).

- **Example**: For `data = b'abc'`, XOR each byte with `(size_mod + (i % 256)) % 256`, repeated 1000 times.

Transformation Selection

- **Fast Mode**: Uses transformations 1, 3, 4, 5, 6, 7, 8, 9 for quicker processing.

- **Slow Mode**: Uses all transformations (1–255) for maximum compression potential.

- **Filetype Prioritization**: For JPEG and TEXT files, transformations 7, 8, 9, 10, 11 (and 12–255 in slow mode) are prioritized due to their effectiveness on these file types, as they leverage π digits, primes, and pattern-based transformations that align well with image and text data structures.

Huffman Coding

Huffman coding is used as one of the compression methods, particularly effective for small files (below `HUFFMAN_THRESHOLD = 1024` bytes). It is implemented as follows:

Huffman Compression

- **Process**:

  1. **Convert to Binary String**: The input data is converted to a binary string (e.g., `b'abc'` becomes a binary representation of its byte values).

  2. **Calculate Frequencies**: Counts the frequency of '0' and '1' in the binary string.

  3. **Build Huffman Tree**: Creates a binary tree where leaves represent bits ('0' or '1'), and paths to leaves define variable-length codes based on frequency (more frequent bits get shorter codes).

  4. **Generate Codes**: Assigns Huffman codes to each bit.

  5. **Compress**: Replaces each bit in the binary string with its Huffman code, producing a compressed binary string.

  6. **Convert to Bytes**: Converts the compressed binary string to bytes for storage.

- **Marker**: Uses marker `4` to indicate Huffman compression.

- **Example**: For `data = b'a'`, binary is `01100001`. If '0' is more frequent, it might get code `0`, and '1' gets `10`, reducing the output size.

Huffman Decompression

- **Process**:

  1. **Read Compressed Data**: Converts the input bytes back to a binary string.

  2. **Rebuild Huffman Tree**: Uses the frequency of bits in the compressed data to reconstruct the Huffman tree.

  3. **Decode**: Traverses the Huffman tree for each bit in the compressed string to recover the original binary string.

  4. **Convert to Bytes**: Converts the binary string back to bytes.

- **Challenges**: If the input is empty or the Huffman tree cannot be rebuilt, it returns an empty string or logs an error.

Usage

- Huffman coding is tested alongside PAQ and zlib for files smaller than 1024 bytes. If it produces the smallest output, it is selected, and the marker `4` is prepended to the output.

Dictionary-Based Approach (Seed Tables)

The **seed tables** are a dictionary-like mechanism used in **Transform_09** to introduce additional randomness based on data size. Here's how they work:

- **Generation**:
  - Creates 126 tables, each containing 256 random integers between 5 and 255, seeded with a fixed value (42) for reproducibility.
  - **Code**: `random.seed(seed); tables.append([random.randint(min_val, max_val) for _ in range(table_size)])`.
- **Usage in Transform_09**:
  - Selects a table index based on `data_length % num_tables`.
  - Retrieves a seed value using `get_seed(table_idx, value)`, where `value` is the data length.
  - XORs each byte with the seed value combined with other factors (prime and π digits).
- **Purpose**: Acts as a precomputed dictionary to add controlled randomness, making the data more suitable for compression by breaking predictable patterns.
- **Example**: For `data_length = 1000`, table index is `1000 % 126 = 10`, seed value is retrieved from `seed_tables[10][1000 % 256]`, and used in XOR operations.

System Workflow

Compression

1. **Read Input**: Reads the input file as bytes.
2. **Detect Filetype**: Determines if the file is JPEG, TEXT, or DEFAULT based on extension.
3. **Get Creation Time**: Retrieves the file's creation time for embedding.
4. **Apply Transformations**:

- Tests transformations based on mode (fast or slow) and filetype.

- Prioritizes transformations 7, 8, 9, 10, 11 for JPEG/TEXT in slow mode.

5. **Compress**:

- Applies PAQ and zlib compression to each transformed output.

- For files < 1024 bytes, also tests Huffman coding.

- Selects the method with the smallest output size.

6. **Output Format**: Prepends a marker byte (indicating the transformation/compression method) and 6 bytes of encoded datetime, followed by the compressed data.

7. **Write Output**: Saves the result to the output file and logs the compression ratio and zero-byte count.

Decompression

1. **Read Input**: Reads the compressed file.

2. **Parse Header**:

- Extracts the marker byte and datetime bytes.

- Decodes the datetime for setting the output file's creation time.

3. **Decompress**:

- If marker is 4, uses Huffman decompression.

- Otherwise, tries PAQ decompression; if it fails, attempts zlib.

4. **Reverse Transformation**: Applies the reverse transformation corresponding to the marker (1–255).

5. **Write Output**: Saves the decompressed data and sets the file's creation time.

6. **Logging**: Reports the decompressed size and zero-byte count.

Key Features

- **Pi-Based Transformations**: Uses the first three base-10 digits of π (mapped to base-256) in transformations 7, 8, 9 to introduce pseudo-randomness.

- **Dynamic Transformations**: Generates transformations 12–255 on-the-fly for extensive experimentation.

- **Filetype Optimization**: Prioritizes transformations for JPEG and TEXT files to leverage their structural properties.

- **Huffman for Small Files**: Efficiently handles small files with Huffman coding.

- **Seed Tables**: Acts as a dictionary to add controlled randomness in Transform_09.

- **Datetime Embedding**: Embeds file creation time in the compressed output for metadata preservation.

- **Fast and Slow Modes**: Balances speed and compression efficiency.

- **Robust Error Handling**: Logs warnings/errors for empty inputs, invalid data, or compression failures.

Example Usage

Compression

```bash

PAQJP_6 Compression System with Base-10 Pi Transformation (3 digits)

Created by Jurijus Pacalovas.

Options:

1 - Compress file (PAQJP_6 with transformations and datetime)

2 - Decompress file (PAQJP_6 with transformations and datetime)

Enter 1 or 2: 1

Enter compression mode (1 for fast, 2 for slow): 2

Input file name: image.jpg

Output file name: image.paq
```

```

- **Action**: Compresses `image.jpg` using slow mode, testing all transformations (1–255), prioritizing 7, 8, 9, 10, 11. Outputs `image.paq` with marker, datetime, and compressed data.

### Decompression

```bash

Enter 1 or 2: 2

Input file name: image.paq

Output file name: image_decompressed.jpg

```

- **Action**: Reads `image.paq`, extracts marker and datetime, decompresses using the appropriate method, applies the reverse transformation, and sets the output file's creation time.

Implementation Notes

- **Pi Digits**: Stored in `pi_digits.txt` and loaded/generated as needed. The digits 3, 1, 4 are mapped to 85, 28, 113 in base-256.

- **Logging**: Comprehensive logging tracks transformation cycles, compression sizes, and errors.

- **Error Handling**: Handles empty files, invalid inputs, and compression failures gracefully.

- **Dependencies**: Requires `paq`, `zlib`, `mpmath` for π digit generation, and standard Python libraries.

- **Limitations**: Transformation 11 is computationally intensive in slow mode due to testing 255 `y` values. Huffman coding may fail for highly skewed bit distributions.
```

Future Improvements

- **Optimize Transform_11**: Reduce the number of `y` values tested or use heuristics to select promising candidates.

- **Parallel Processing**: Parallelize transformation and compression tests for faster slow-mode operation.

- **Adaptive Transformations**: Dynamically select transformations based on data analysis (e.g., entropy, byte frequency).

- **Extended Pi Digits**: Allow more π digits for larger files to enhance transformation complexity.

- **Custom Huffman**: Optimize Huffman coding for specific file types or bit patterns.

Conclusion

The **PAQJP_6 Compression System** represents a significant advancement in lossless compression technology, combining a diverse set of transformations (1–11 and 12–255) with multiple compression methods (PAQ, zlib, Huffman) to achieve optimal compression ratios. By leveraging the first three base-10 digits of π, prime numbers, and seed tables, the system introduces controlled randomness to enhance compressibility, particularly for JPEG and TEXT files. The inclusion of fast and slow modes allows users to balance speed and compression efficiency, while robust error handling and datetime embedding ensure reliability and metadata preservation. The dynamic generation of transformations 12–255 and the optimization of transformation 11 demonstrate the system's flexibility and potential for high compression performance. Future enhancements, such as parallel processing and adaptive transformation selection, could further improve its efficiency and applicability. Overall, PAQJP_6 is a versatile and powerful tool for compressing a wide range of file types, making it a valuable contribution to the field of data compression.

When done:

date and time is 01:25 PM IST on Tuesday, August 05, 2025.