Quantum PAQJP_4.3.5 already working and lossless

Author Jurijus Pacalovas

Project Overview

The PAQJP_4 system, created by Jurijus Pacalovas, is designed to compress and decompress files with enhanced efficiency by applying a series of transformations before compression. It integrates multiple compression methods (PAQ, zlib, and Huffman coding) and selects the best method based on the smallest output size. The system also embeds file metadata, such as creation time, into the compressed output and uses a custom set of transformations to preprocess data, making it more amenable to compression. Key features include:

- **Filetype Detection**: Automatically detects file types (e.g., JPEG, TEXT, DEFAULT) based on extensions to tailor transformations.
- **Transformations**: A set of nine transformations (01-09) preprocess the data to enhance compressibility.
- **Compression Methods**: Supports PAQ, zlib, and Huffman coding, with automatic selection of the best method.
- **Pi-Based Transformations**: Uses the first three base-10 digits of π (3, 1, 4), mapped to 0-255, in transformations 07, 08, and 09.
- **Datetime Encoding**: Embeds file creation time in the compressed output for metadata preservation.
- **Reversibility**: All transformations are reversible to ensure lossless compression.

Transformations 01-09: Detailed Explanation

The system defines nine transformations (labeled 01-09) that preprocess the input data before compression and reverse-transform the data after decompression. Each transformation modifies the input byte stream to improve compression efficiency by altering patterns or reducing entropy. Below is a detailed breakdown of each transformation and its reverse operation:

Transformation 01: Prime XOR Every 3 Bytes

- **Description**: Applies an XOR operation with prime numbers (from a precomputed list of primes between 2 and 255) to every third byte of the input data. The XOR value is either the prime itself (for prime = 2) or a scaled value (`ceil(prime * 4096 / 28672)`). The operation is repeated 100 times by default.
- **Purpose**: Introduces randomness using prime numbers to break repetitive patterns, potentially making the data more compressible.
- **Implementation**:

```python
def transform_with_prime_xor_every_3_bytes(data, repeat=100):
    transformed = bytearray(data)
    for prime in PRIMES:
        xor_val = prime if prime == 2 else max(1, math.ceil(prime * 4096 / 28672))
        for _ in range(repeat):
            for i in range(0, len(transformed), 3):
                transformed[i] ^= xor_val
    return bytes(transformed)
```

- **Reverse Transformation**: Since XOR is self-inverse (`a ^ b ^ b = a`), the reverse transformation is identical to the forward transformation.
- **Usage**: Applied to all file types but prioritized lower for JPEG and TEXT files.

Transformation 02: Alias for Transformation 01

- **Description**: Identical to Transformation 01 in the provided code (uses `transform_with_prime_xor_every_3_bytes`).
- **Purpose**: Likely included for flexibility or future differentiation but currently redundant.
- **Reverse Transformation**: Same as Transformation 01.
- **Note**: This appears to be a placeholder or duplicate in the current implementation.

Transformation 03: Pattern Chunk Inversion

- **Description**: Divides the input data into chunks of 4 bytes and applies a bitwise XOR with 0xFF (inverts all bits) to each byte in the chunk.
- **Purpose**: Inverts bit patterns to potentially create more uniform or compressible data.
- **Implementation**:

```python
def transform_with_pattern_chunk(data, chunk_size=4):
    transformed = bytearray()
    for i in range(0, len(data), chunk_size):
        chunk = data[i:i + chunk_size]
        transformed.extend([b ^ 0xFF for b in chunk])
    return bytes(transformed)
```

- **Reverse Transformation**: Identical to the forward transformation since XOR with 0xFF is self-inverse.
- **Usage**: Applied to all file types, with lower priority for JPEG and TEXT files.

Transformation 04: Position-Based Subtraction

- **Description**: Subtracts the index modulo 256 from each byte, repeated 100 times by default.
- **Purpose**: Introduces a position-dependent transformation to disrupt repetitive patterns.
- **Implementation**:

```python
def transform_04(self, data, repeat=100):
    transformed = bytearray(data)
    for _ in range(repeat):
        for i in range(len(transformed)):
```

transformed[i] = (transformed[i] - (i % 256)) % 256

return bytes(transformed)

```

- **Reverse Transformation**: Adds the index modulo 256 back to each byte, repeated the same number of times.

```python def reverse_transform_04(self, data,

repeat=100):

transformed = bytearray(data)

for _ in range(repeat): for i in

range(len(transformed)):

transformed[i] = (transformed[i] + (i % 256)) % 256

return bytes(transformed)

```

- **Usage**: Applied to all file types, with lower priority for JPEG and TEXT files.

Transformation 05: Bit Rotation

- **Description**: Performs a left circular shift of 3 bits on each byte.

- **Purpose**: Reorganizes bit patterns to potentially improve compressibility by aligning bits differently.

- **Implementation**:

```python def transform_05(self,

data, shift=3):

transformed = bytearray(data)

for i in range(len(transformed)):

transformed[i] = ((transformed[i] << shift) | (transformed[i] >> (8 - shift))) & 0xFF return bytes(transformed)

```

- **Reverse Transformation**: Performs a right circular shift of 3 bits.
```

```python   def reverse_transform_05(self,
data, shift=3):
    transformed = bytearray(data)
for i in range(len(transformed)):
        transformed[i] = ((transformed[i] >> shift) | (transformed[i] << (8 - shift))) & 0xFF
return bytes(transformed)
```

- **Usage**: Applied to all file types, with lower priority for JPEG and TEXT files.

Transformation 06: Random Substitution

- **Description**: Applies a random substitution cipher using a seed (default 42) to shuffle the 0-255 byte range.
- **Purpose**: Randomizes byte values to break patterns, potentially aiding compression.
- **Implementation**:

```python   def transform_06(self,
data, seed=42):
    random.seed(seed)
substitution = list(range(256))
random.shuffle(substitution)
transformed = bytearray(data)
for i in range(len(transformed)):
        transformed[i] = substitution[transformed[i]]
return bytes(transformed)
```

- **Reverse Transformation**: Uses the inverse of the substitution table to restore original bytes.

```python   def reverse_transform_06(self,
data, seed=42):
```

```
    random.seed(seed)
substitution = list(range(256))
random.shuffle(substitution)
reverse_substitution = [0] * 256
for i, v in enumerate(substitution):
        reverse_substitution[v] = i
transformed = bytearray(data)      for
i in range(len(transformed)):
        transformed[i] = reverse_substitution[transformed[i]]
return bytes(transformed)
```

- **Usage**: Applied to all file types, with lower priority for JPEG and TEXT files.

Transformation 07: Pi-Based XOR with File Size

- **Description**: Uses the first three base-10 digits of π (3, 1, 4, mapped to 0-255) to XOR each byte, with a circular shift based on file size modulo 3. Additionally, XORs each byte with the file size modulo 256. The number of cycles is based on file size (in KB), capped at 10.
- **Purpose**: Leverages π digits and file size to introduce controlled randomness, tailored to file characteristics.
- **Implementation**:

```python   def transform_07(self, data,
repeat=100):
    transformed = bytearray(data)
pi_length = len(self.PI_DIGITS)
data_size_kb = len(data) / 1024      cycles =
min(10, max(1, int(data_size_kb)))      shift =
len(data) % pi_length
```

```
    self.PI_DIGITS = self.PI_DIGITS[shift:] + self.PI_DIGITS[:shift]



    size_byte = len(data) % 256 for i in
range(len(transformed)):
transformed[i] ^= size_byte      for _ in
range(cycles * repeat // 10):         for i
in range(len(transformed)):
            pi_digit = self.PI_DIGITS[i % pi_length]
transformed[i] ^= pi_digit      return
bytes(transformed)
```

- **Reverse Transformation**: Identical to the forward transformation (XOR is self-inverse), with the π digits shifted back.
- **Usage**: Prioritized for JPEG and TEXT files due to its effectiveness on structured data.
- **Pi Integration**: The first three digits of π (3, 1, 4) are mapped to 0-255 range (e.g., `(d * 255 // 9) % 256`), resulting in values like `[85, 28, 113]`.

Transformation 08: Pi-Based XOR with Prime-Based Pre-Transformation

- **Description**: Similar to Transformation 07, but pre-XORs each byte with the nearest prime number to the file size modulo 256. Uses π digits with a circular shift.
- **Purpose**: Combines prime number properties with π-based XOR for enhanced randomization.
- **Implementation**:

```python   def transform_08(self, data,
repeat=100):
    transformed = bytearray(data)
pi_length = len(self.PI_DIGITS)
data_size_kb = len(data) / 1024      cycles =
```

```
    self.PI_DIGITS = self.PI_DIGITS[shift:] + self.PI_DIGITS[:shift]


min(10, max(1, int(data_size_kb)))      shift =
len(data) % pi_length

    size_prime = find_nearest_prime_around(len(data) % 256)
for i in range(len(transformed)):          transformed[i] ^=
size_prime      for _ in range(cycles * repeat // 10):          for i
in range(len(transformed)):
            pi_digit = self.PI_DIGITS[i % pi_length]
transformed[i] ^= pi_digit      return
bytes(transformed)
```

- **Reverse Transformation**: Identical to the forward transformation, with π digits shifted back.
- **Usage**: Prioritized for JPEG and TEXT files.
- **Pi Integration**: Same as Transformation 07.

Transformation 09: Pi-Based XOR with Seed Table Modulation

- **Description**: Extends Transformation 08 by XORing each byte with a value from a randomly generated seed table (based on file size modulo the number of tables) and the nearest prime to the file size modulo 256. Applies π-based XOR with a circular shift.
- **Purpose**: Adds an additional layer of randomization using seed tables for complex data patterns.
- **Implementation**:

```python   def transform_09(self, data,
repeat=100):
    transformed = bytearray(data)
pi_length = len(self.PI_DIGITS)
```

```
    self.PI_DIGITS = self.PI_DIGITS[shift:] + self.PI_DIGITS[:shift]


data_size_kb = len(data) / 1024      cycles =
min(10, max(1, int(data_size_kb)))      shift =
len(data) % pi_length

    size_prime = find_nearest_prime_around(len(data) % 256)
seed_idx = len(data) % len(self.seed_tables)      seed_value =
self.get_seed(seed_idx, len(data))      for i in
range(len(transformed)):
        transformed[i] ^= size_prime ^ seed_value
for _ in range(cycles * repeat // 10):          for i
in range(len(transformed)):
            pi_digit = self.PI_DIGITS[i % pi_length]
transformed[i] ^= pi_digit ^ (i % 256)      return
bytes(transformed)
```

- **Reverse Transformation**: Identical to the forward transformation, with π digits shifted back.
- **Usage**: Prioritized for JPEG and TEXT files.
- **Pi Integration**: Same as Transformation 07.

Key Features of Transformations

- **Pi Digits**: Transformations 07, 08, and 09 use the first three base-10 digits of π (3, 1, 4), mapped to 0-255 using `(d * 255 // 9) % 256`. These digits are stored in a file (`pi_digits.txt`) or generated if the file is absent. The mapping ensures values fit within a byte range.

```
    self.PI_DIGITS = self.PI_DIGITS[shift:] + self.PI_DIGITS[:shift]
```

- **File Size Dependency**: Transformations 07, 08, and 09 adjust the number of cycles based on file size (in KB) and use file size modulo 256 or its nearest prime for XOR operations.
- **Reversibility**: All transformations are designed to be reversible, ensuring lossless compression. XOR-based transformations (01, 03, 07, 08, 09) are self-inverse, while others (04, 05, 06) have explicit reverse functions.

- **Priority for File Types**: Transformations 07, 08, and 09 are prioritized for JPEG and TEXT files, as they are likely optimized for structured or repetitive data.

Compression and Decompression Process

Compression

1. **Filetype Detection**: Determines the file type (JPEG, TEXT, DEFAULT) based on the extension.
2. **Transformation Selection**: Applies transformations (01-09), with 07-09 prioritized for JPEG and TEXT files.
3. **Compression Methods**: Tests PAQ, zlib, and Huffman (for files < 1024 bytes) after each transformation, selecting the method and transformation yielding the smallest output.
4. **Datetime Encoding**: Embeds the file's creation time (6 bytes) using a custom encoding scheme.
5. **Output Format**: The compressed file starts with a 1-byte marker indicating the transformation used, followed by 6 bytes of datetime data, and then the compressed data.
6. **Logging**: Reports compression ratio and method used.

Decompression

1. **Marker Extraction**: Reads the transformation marker (first byte) to determine the reverse transformation.
2. **Datetime Decoding**: Extracts and decodes the 6-byte datetime to restore the file's creation time.
3. **Decompression**: Attempts PAQ decompression first; if it fails, tries zlib. For marker 4, uses Huffman decompression.
4. **Reverse Transformation**: Applies the corresponding reverse transformation (01-09) based on the marker.
5. **Output**: Writes the decompressed data to the output file and sets its creation time to the decoded datetime.

Additional Components

- **Pi Digit Generation**: Uses the `mpmath` library to generate π digits or loads them from a file. Fallbacks to [3, 1, 4] if generation fails.
- **Seed Tables**: Transformation 09 uses 126 random tables (256 values each) for additional randomization.
- **State Table**: A `StateTable` class with a predefined table is included but not used in the transformations, suggesting potential for future context modeling or arithmetic coding.
- **Huffman Coding**: Used for small files (< 1024 bytes) with a custom Huffman tree implementation.
- **Datetime Handling**: Encodes and decodes file creation time to preserve metadata, using a compact 6-byte format.

Strengths and Weaknesses

Strengths

- **Flexible Transformations**: Multiple transformations cater to different data types, with π-based transformations (07-09) optimized for structured files.
- **Best Method Selection**: Automatically chooses the most effective compression method, improving efficiency.
- **Lossless**: All transformations and compression methods are reversible, ensuring no data loss.
- **Metadata Preservation**: Embeds file creation time, useful for archiving.

Weaknesses

- **Complexity**: The multitude of transformations and compression methods increases computational overhead.

- **Redundancy**: Transformation 02 is identical to 01, indicating potential code optimization.
- **Limited Pi Digits**: Uses only three π digits, which may limit the transformation's effectiveness for large files.
- **Error Handling**: While robust, errors in one compression method may cascade, though fallbacks (e.g., zlib after PAQ) mitigate this.

Conclusion

The PAQJP_4 Compression System is a versatile and innovative tool that leverages mathematical constants (π digits), prime numbers, and multiple compression algorithms to achieve efficient, lossless compression. The transformations 01-09 provide a range of preprocessing techniques, with 07-09 being particularly notable for their use of π digits and file size-based adjustments. The system's ability to select the best compression method and preserve metadata makes it suitable for various file types, especially JPEG and TEXT files for all types files can use.