A DNA-based Data Storage Method using Reed-Solomon Algorithm

Site Fan
12111624@mail.sustech.edu.cn
SUSTech, China

Jiachen Xiao
12112012@mail.sustech.edu.cn
SUSTech, China

Jan 1, 2025

Abstract

DNA storage is a cutting-edge approach to data storage, offering remarkable benefits like high storage density, energy efficiency, and longevity. Our project implements a decoding system that converts raw DNA sequence data into usable digital information, specifically by decoding DNA sequences that encode image files. The decoding backend is implemented in C++ and uses Reed-Solomon error correction to ensure data integrity. We also utilize Python2 for the frontend of the decoding process using its PRNG to recover the *The Great Wave off Kanagawa*. Our custom C++ encoder works seamlessly with the C++ full decoding backend. The final image file is successfully recovered, demonstrating the potential of DNA storage systems for high-density data retrieval.

1 Introduction

In this project, we focus on decoding an image file from DNA sequences, which consist of four bases: A, T, G, and C. This work is inspired by prior research (Erlich and Zielinski 2017), and aims to recover the encoded image by implementing a decoding system that handles errors introduced during DNA sequencing. The system is built in stages, including preprocessing, droplet recovery, and segment inference, all of which contribute to the final recovered image.

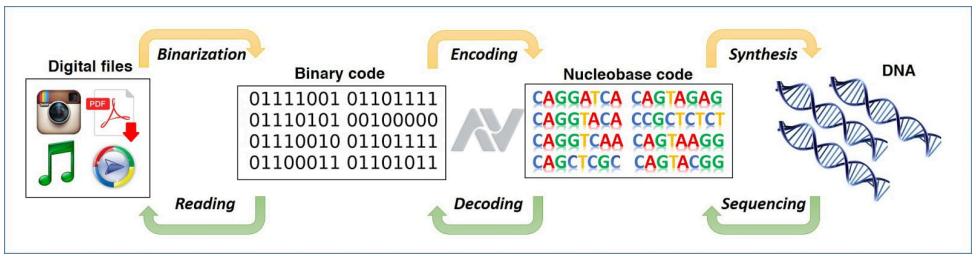


Figure 1: An Overview of DNA Storage Workflow

1.1 Overview of DNA Storage System

DNA data storage has emerged as a promising solution for addressing the exponential growth of data. It leverages the high storage density, durability, and low maintenance costs of DNA molecules to store vast amounts of information(Jo et al. 2024). Recent advancements

in high-throughput DNA synthesis and error correction techniques have further enhanced the feasibility of DNA storage. Researchers have explored various encoding strategies, preservation methods, and retrieval processes to optimize DNA data storage systems(Jo et al. 2024).

Despite the challenges such as synthesis errors and data retrieval efficiency, DNA storage holds great potential for revolutionizing data storage technologies. DNA-based storage offers numerous advantages over traditional storage methods(Erlich and Zielinski 2017):

- **High storage density**: DNA can store vast amounts of data in a tiny physical space.
- **Low energy consumption**: Unlike electronic storage systems, DNA does not require power to maintain its stored information.
- **Long lifespan**: DNA is a stable medium that can last for thousands of years if stored properly.

However, challenges persist, notably the error rates in DNA sequencing, which can range from 10% to 30% or even higher. These errors include:

- **Substitution errors**: Incorrect base substitutions during sequencing.
- **Insertion errors**: Extra bases being inserted into the sequence.
- **Deletion errors**: Loss of bases during sequencing.

This project addresses these challenges through error-correction techniques, such as Reed-Solomon encoding.

1.2 Luby Transform Code

LT (Luby Transform) codes are a type of erasure code used for reliable data transmission over unreliable or lossy networks. They are the first practical realization of fountain codes, which are rateless erasure codes. LT codes are designed to generate potentially infinite numbers of encoded symbols from a given set of source symbols, allowing the receiver to recover the original data from any subset of the encoded symbols that is slightly larger than the original set(Luby 2002).

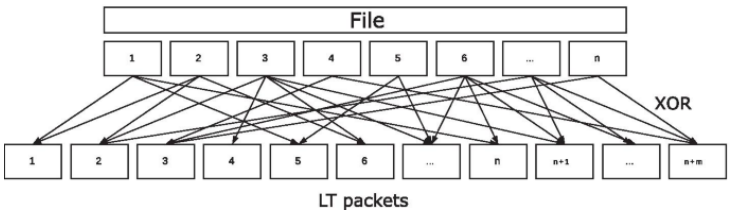


Figure 2: The Encoding Process of LT Codes

1.2.1 Encoding

The encoding process begins by dividing the uncoded message into n chunks of roughly equal length. Encoded packets are then produced with the help of a pseudorandom number generator.

1. **Degree Distribution**: Select a degree d from a predefined degree distribution.
2. **Random Selection**: Randomly choose d source symbols from the original data.

3. **XOR Operation**: XOR the selected source symbols to produce an encoded symbol.

If M_i is the i-th block of the message, the data portion of the next packet is computed as:

$$M_{i_1} \oplus M_{i_2} \oplus \cdots \oplus M_{i_d} \tag{1}$$

where $\{i_1, i_2, ..., i_d\}$ are the randomly chosen indices for the d chunks included in this packet.

A prefix is appended to the encoded packet, defining how many chunks n are in the message, how many chunks d have been XORed into the data portion of this packet, and the list of indices $\{i_1, i_2, ..., i_d\}$. Finally, an error-detecting code is applied to the packet, and the packet is transmitted.

1.2.2 Decoding

The decoding process uses the XOR operation to retrieve the encoded message. The procedure works as follows:

1. If the current packet replicates a packet already processed, it is discarded.
2. If the current clean packet is of degree $d > 1$, it is processed against all fully decoded chunks in the message queuing area, then stored in a buffer area if its reduced degree is greater than 1.
3. When a new, clean packet of degree $d = 1$ is received, it is moved to the message queuing area and matched against all packets of degree $d > 1$ in the buffer. It is XORed into the data portion of any buffered packet that was encoded using M_i, the degree of that matching packet is decremented, and the list of indices for that packet is adjusted.
4. When this process unlocks a block of degree $d = 2$ in the buffer, that block is reduced to degree 1 and processed against the packets remaining in the buffer.
5. When all n chunks of the message have been moved to the message queuing area, the receiver signals that the message has been successfully decoded.

This decoding procedure works because $A \oplus A = 0$ for any bit string A. After $d - 1$ distinct chunks have been XORed into a packet of degree d, the original unencoded content of the unmatched block is all that remains.

The degree distribution used in LT codes is crucial for their performance. A common choice is the Robust Soliton Distribution, which ensures that the decoding process is efficient and reliable(Chen et al. 2013).

1.3 Reed-Solomon Code

Reed-Solomon (RS) codes are a type of error-correcting code that is widely used in digital communications and storage. They are block codes that can detect and correct multiple symbol errors within a block of data. RS codes are particularly effective for correcting burst errors, where multiple consecutive symbols are corrupted.

1.3.1 Encoding

The encoding process of RS codes involves the following steps:

1. **Polynomial Representation**: Represent the data as a polynomial over a finite field.

2. **Parity Symbols**: Generate parity symbols by evaluating the polynomial at additional points.
3. **Codeword Construction**: Construct the codeword by appending the parity symbols to the original data.

If $i(x)$ is the information polynomial and $g(x)$ is the generator polynomial, the codeword $c(x)$ is constructed as:

$$c(x) = i(x)c \cdot x^{\{n-k\}} + r(x) \tag{2}$$

where $r(x)$ is the remainder when $i(x)c \cdot x^{\{n-k\}}$ is divided by $g(x)$.

1.3.2 Decoding

The decoding process involves the following steps:

1. **Syndrome Calculation**: Compute the syndromes by evaluating the received polynomial at the same points used for generating the parity symbols.
2. **Error Locator Polynomial**: Use the syndromes to construct the error locator polynomial.
3. **Error Correction**: Find the roots of the error locator polynomial to identify the error positions and magnitudes, and correct the errors.

The syndromes S_i are calculated as:

$$S_i = r(\alpha^i) \tag{3}$$

where $r(x)$ is the received polynomial and α is a primitive element of the finite field.

The error locator polynomial $\sigma(x)$ is constructed using the Berlekamp-Massey algorithm or Euclid's algorithm. The roots of $\sigma(x)$ are found using the Chien search algorithm, and the error values are calculated using the Forney algorithm.

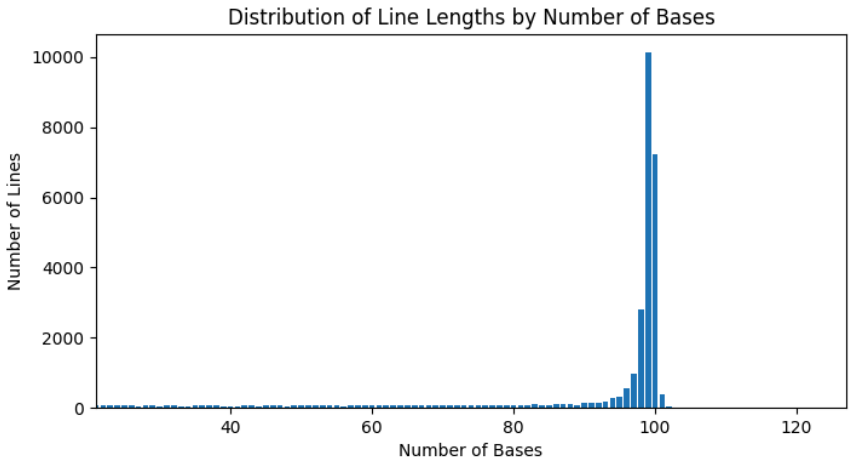
The Reed-Solomon code used in this project is capable of correcting up to 10 substitution errors per droplet, ensuring the integrity of the recovered data.

2 Methodology

Our decoding system is built in two primary components: the encoder and the decoder. The encoder, implemented in C++, generates the DNA-encoded file from the original image. The decoder, implemented in a combination of C++ and Python2, recovers the image by decoding the DNA sequence data, correcting errors, and reconstructing the original file.

2.1 Data Analysis

The project provides a sequencing result file called "50-SF.txt", which includes 27,726 sequencing sequences that contain all the information about The Great Wave off Kanagawa. The DNA sequences in the file are divided into different lengths, ranging from 21 to 127 bases. Meanwhile, there are insertion, deletion, and substitution errors in the sequencing results.



The figure below illustrates that although most of the DNA fragments in the file are between 98-101 in length, there are still significant base sequencing errors that cannot be ignored. If the length of DNA sequence is far away from 100, it should be discarded. Besides, for those duplicated DNA sequences after the preprocessing, only one of them should be kept for simplicity.

2.2 Encoding Scheme

The image used in this project is "*The Great Wave off Kanagawa*," a binary file that is divided into segments for DNA encoding. Each segment consists of:

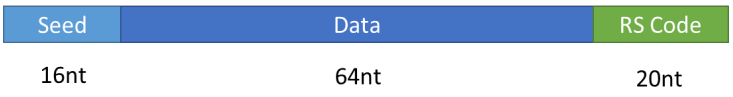


Figure 4: The Encoding Scheme in Our Project

- Address: 16 bases, 4 bytes
- Data: 64 bases, 16 bytes
- Reed-Solomon Code: 20 bases, 5 bytes

This encoding scheme ensures that we need at least 2025 sequences for reliable decoding, each capable of detecting up to 20 substitution errors and correcting up to 10 errors.

Once the DNA is encoded, the resulting file is sequenced and stored in a DNA storage system. The `50-SF.txt` file, a sequence of 27,726 DNA reads, is the one used for decoding in this project.

2.3 Decoding Scheme

Our decoding scheme follows a staged process of preprocessing, droplet recovery, and segment inference.

2.3.1 Preprocessing

In this project, based on the massive number of given DNA droplets compared to the relatively small number of healthy droplets required to recover the information, we first filtered the

DNA droplets by the number of bases. We first select droplets with a complete number of bases, even though they might have substitution errors.

2.3.2 Droplet Recovery

After de-serializing these droplets from base-chain to a list of bytes, we apply the Reed-Solomon algorithm to try to recover these droplets if any error exists.

Droplet recovery involves translating DNA sequences from base pairs `(A, C, G, T)` to binary `(0x00, 0x01, 0x10, 0x11)` and then decoding them using Reed-Solomon (RS) error correction. The recovery process includes the following steps:

- **Base mapping**: Convert DNA bases to binary pairs.
- **Error correction**: Use the Reed-Solomon code to attempt error correction on the sequences.
- **Extraction**: After correction, extract the seed and data from the sequence.

We use the Reed-Solomon codec (RSCodec) to detect and correct errors. If we cannot recover a droplet from errors, the droplet is discarded. Otherwise, the seed and data are extracted and stored into the intermediate results.

By Reed-Solomon, we are able to recover droplets with less than 10 base errors, which are sufficient for recovering the image.

2.3.3 Segment Inference

Segment inference is a process of identifying and recovering the encoded chunks from the droplets. This is done using a message-passing algorithm:

- **Generate identifiers**: Each droplet is assigned segment identifiers, which represent chunks of the original data.
- **XOR operation**: Droplets that contain inferred segments are updated by XORing with known segments.
- **Propagate information**: Once a segment is inferred, it is propagated recursively to all droplets that contain it.

This algorithm ensures that once a segment is recovered, it can be used to help recover other related segments until all segments are decoded.

2.4 Implementation

The backend of our decoding system is implemented in C++, with the main logic for droplet recovery and segment inference handled by the Python2 frontend. Due to limitations in Python2's PRNG, which affects the deterministic mapping between seed and chunk IDs, we were unable to fully migrate the decoding process to C++ for the given encoded file `50-SF.txt`.

However, we implemented a custom C++ encoder that works seamlessly with our C++ decoding backend, demonstrating the potential for a fully integrated system. It is worth noting that our encoder follows a more deterministic approach of mapping seeds to chunk IDs, which is essential for further optimization and integration with the decoding process.

The project structure is as follows:

```
Kanagawa_DNA_Storage
├── 50-SF.txt
├── build
│   ├── decoder
│   └── encoder
├── CMakeLists.txt
├── finalize.py
├── include
│   ├── GaloisField.hpp
│   ├── Polynomial.hpp
│   └── ReedSolomon.hpp
├── src
│   ├── Decoder.cpp
│   ├── Encoder.cpp
│   ├── GaloisField.cpp
│   ├── Polynomial.cpp
│   └── ReedSolomon.cpp
└── utils
    └── // Python Packages
```

In the `src` directory, `Decoder.cpp` contains the main decoding logic, while `Encoder.cpp` implements the encoding process. The `include` directory contains the necessary header files for the project. The `utils` directory contains Python packages used for the frontend of the decoding process.

2.5 Final Decoder Process

The overall decoding workflow is structured as follows:

- **Initialization**: Read the DNA sequence file and initialize variables.
- **Droplet Recovery**: For each DNA sequence, apply droplet recovery and attempt segment inference.
- **Completion Check**: The process terminates once a sufficient number of chunks are recovered.
- **Image Reconstruction**: Once all chunks are decoded, the data is reconstructed into the original image.

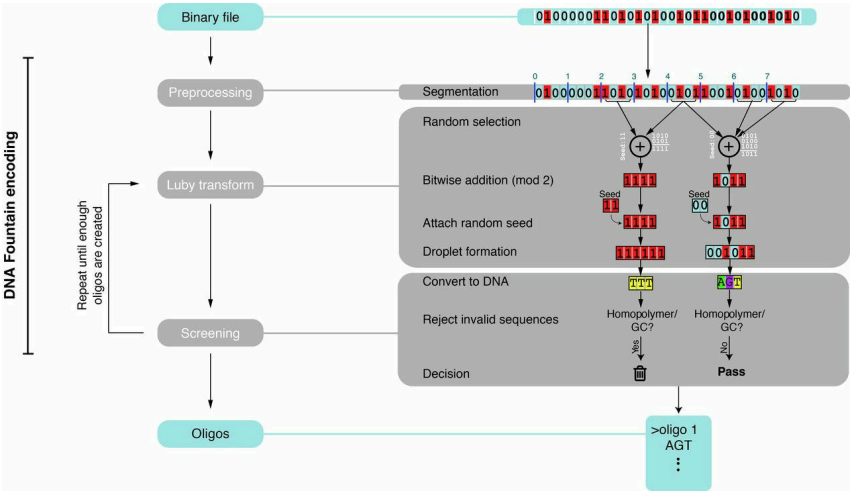


Figure 5: The Decoding Workflow

Overall, the decoding process follows the DNA fountain coding pattern, involving a combination of error correction, droplet recovery, and segment inference to reconstruct the original image from the DNA sequences.

3 Results

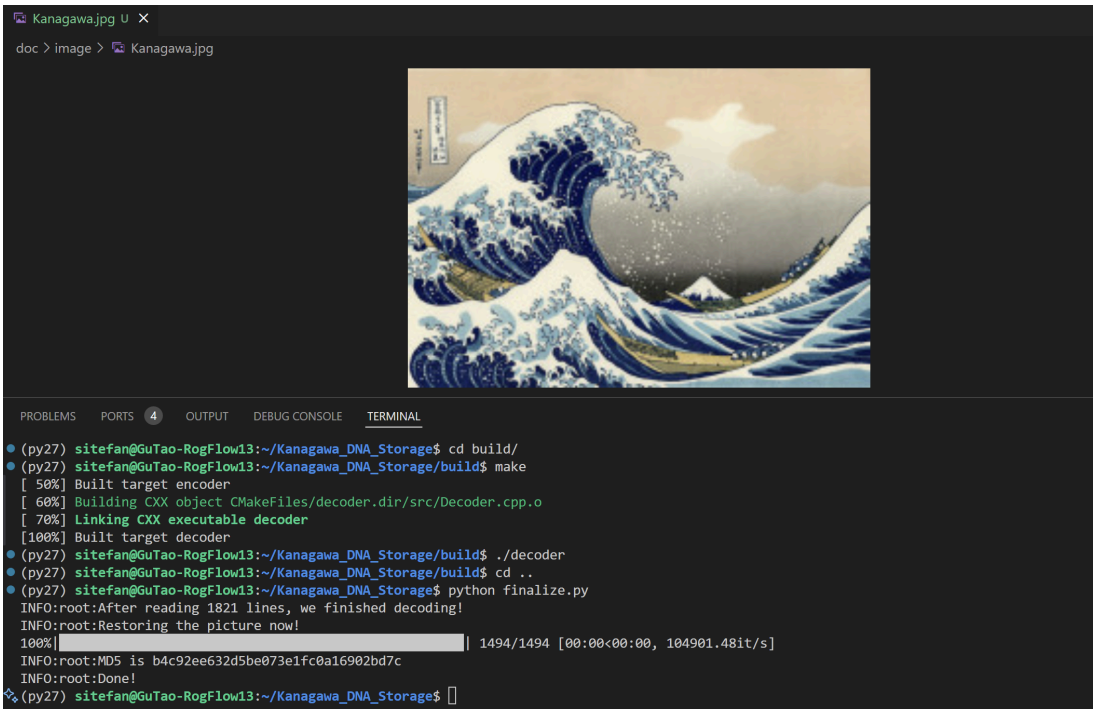


Figure 6: The Recovered *The Great Wave off Kanagawa*

By running the decoding process of backend and frontend on the provided DNA sequence file `50-SF.txt`, we successfully recovered the encoded image "*The Great Wave off Kanagawa*" using 1821 lines, among which 1494 distinct droplets involved decoding. This number is relatively smaller than the total number of droplets, demonstrating the efficiency of the decoding system, and indicating that the redundancy in the encoding scheme is sufficient for error correction.

The final image file is identical to the original image, demonstrating the effectiveness of our decoding system.

4 Conclusion

Our project makes the following contributions:

- Implementation of a encoding/decoding system for DNA-encoded data, specifically for image files.
- Utilization of LT Code and Reed-Solomon error correction to ensure data integrity by redundancy during decoding.
- Integration of C++ backend with Python2 frontend for efficient decoding, along with a C++ encoder optimized with deterministic mapping.
- Successful recovery of "*The Great Wave off Kanagawa*" image from DNA sequences.

Bibliography

[1] Y. Erlich and D. Zielinski, "DNA Fountain enables a robust and efficient storage architecture," *Science*, vol. 355, no. 6328, pp. 950–954, 2017.

[2] S. Jo, H. Shin, S.-y. Joe, D. Baek, C. Park, and H. Chun, "Recent progress in DNA data storage based on high-throughput DNA synthesis," *Biomedical Engineering Letters*, vol. 14, pp. 993–1009, 2024.

[3] M. Luby, "LT codes," in *Proceedings of the 43rd Annual IEEE Symposium on Foundations of Computer Science*, 2002, p. 271.

[4] C.-M. Chen, Y.-P. Chen, T.-C. Shen, and J. K. Zao, "A Practical Optimization Framework for the Degree Distribution in LT Codes," *IEEE Transactions on Communications*, 2013.

APPENDIX A Contributors

In this project, the following contributors made significant contributions:

- Site Fan: Responsible for the backend implementation, including the C++ encoder and decoder logic; documentation.
- Jiachen Xiao: Responsible for the frontend implementation, including the Python2 decoding process; documentation.