EE411 FINAL PROJECT REPORT: DECODING OF DNA STORAGE

Peijia Qin
Southern University of Science and Technology
12111327@mail.sustech.edu.cn

Zubin Zheng
Southern University of Science and Technology
12112328@mail.sustech.edu.cn

ABSTRACT

DNA storage systems, emerging as a novel approach to information storage, offer advantages such as high density, low energy consumption, and long lifespan. This project aims to decode an image file from a DNA sequence, utilizing an algorithm primarily inspired by [1]. We implement a staged decoding scheme, including preprocessing, droplet recovery, and segment inference, using Python 2.7.18. MD5 of decoded picture file 50-SF.jpg **b4c92ee632d5be073e1fc0a16902bd7c**. The successful decoding results demonstrate the potential application of DNA storage systems in information retrieval. However, challenges such as base errors in DNA sequencing need to be addressed for practical engineering applications.

1 Introduction

In this project, we aim to decode an image file from a sequence of DNA, which consists of only four characters A, T, G, C. The algorithm adopted is mainly from [1].

Overview of DNA Storage Systems

Compared to traditional storage systems, DNA storage systems offer several advantages:

1. High storage density
2. Low energy consumption
3. Long storage life, and so on

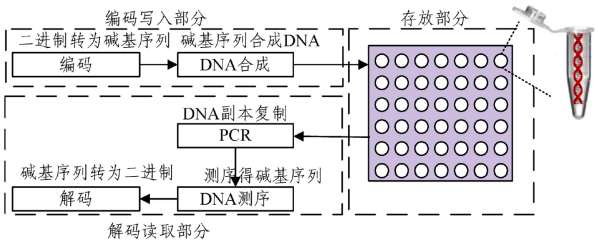

图 2　DNA 存储流程图

Figure 1: Overview of the DNA storage pipeline

Issues to be addressed in future engineering applications—Base errors in DNA sequencing:

1. DNA sequencing is prone to base sequencing errors (10%-30% or even higher error rates).
2. Base sequencing errors include:
   (a) Substitution errors (one or more bases in the DNA sequencing sequence are replaced with other bases),
   (b) Insertion errors (one or more non-existent bases are inserted into the DNA sequencing sequence),
   (c) Deletion errors (one or more bases in the DNA sequencing sequence are deleted).

2 Methodology

The core methodology in this project contains encode scheme and decode scheme. All we need to do is to implement the decoding part.

2.1 Encode Scheme

The file used in this project is the image called *Kanagawa Surfing*, which can be viewed simply as a binary file.

Figure 2: Image file used in this project

To encode the whole file, all the data is divided into 2504 sequences with a length of 100 bases each. For each sequence, the address uses 16 bases, data uses 64 bases, and the remaining 20 bases are used for Reed-Solomon. The whiloe file is eventually split into 1494 encoded segments.

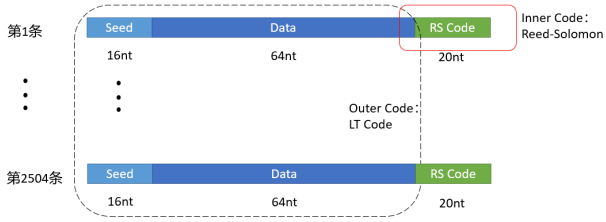

Figure 3: The used encode scheme. 1 "nt" denotes a base.

The distinctive feature of this coding is that it only requires 2025 sequences for decoding, with each sequence capable of detecting 20 substitution errors and correcting 10 substitution errors.

Here, the inner code used is Reed-Solomon code, while the outer code is LT Code [2].

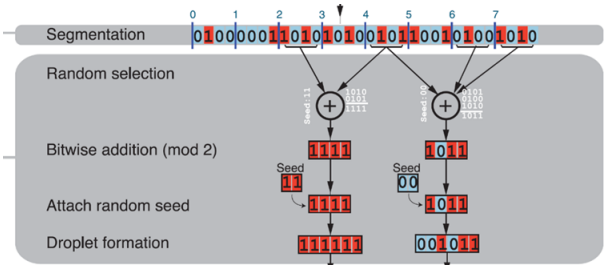

Figure 4: LT-code

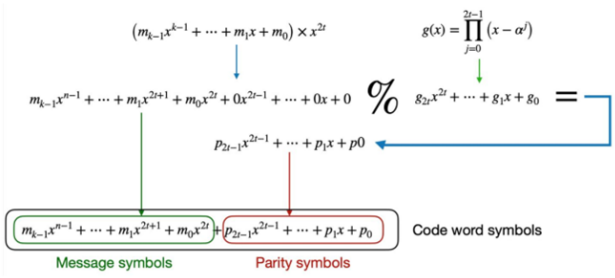

Figure 5: Reed-Solomon code

To get the final DNA segment, the encoded data was sequenced. After sequencing the DNA stored in the DNA storage system, we obtained the file '50-SF', which comprises 27, 726 sequencing sequences. These sequences contain all the information about the Kanagawa Surfing. Besides, the sequencing results have insertion, deletion, and substitution errors.

2.2 Decode Scheme

The decode scheme is divided into 3 steps: preprocessing, droplet recovery, and segment inference, we introduce all of them step by step in the following parts. We have decoded the 50-SF.txt file using

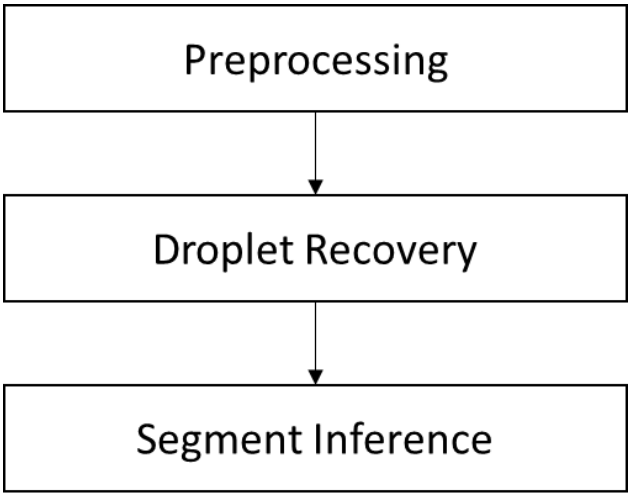

Figure 6: Overview of decode scheme

Python, and the code implementation mainly refers to the following https://github.com/jeter1112/dna-fountain-simplified/blob/main/py2ImagDecode/decode.py.

2.2.1 Preprocessing

The ideal preprocessing process should be divided into the following steps

1. Stitch the paired-end reads using PEAR
2. Retain only sequences whose length is 60/100nt
3. Collapse identical sequence and store the collapsed sequence and number of occurrences in the data
4. Sort the sequences based their abundance

The project provides a sequencing result file called "50-SF.txt", which includes 27726 sequencing sequences that contain all the information about Kanagawa surfing. Meanwhile, there are insertion, deletion, and replacement errors in the sequencing results. Table 1 shows the corresponding count of DNA of different lengths in the file "50-SF.txt".

Figure 7 illustrates more intuitively 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 removed. In addition, duplicate DNA sequences should be removed and only unique DNA sequences should be retained.

Table 1: Statistics of DNA length and count in '50-SF.txt' file

length	**count**	**length**	**count**	**length**	**count**	**length**	**count**	**length**	**count**	**length**	**count**
21	57	41	48	61	63	81	75	101	386	121	-
22	70	42	58	62	74	82	66	102	19	122	-
23	59	43	52	63	72	83	95	103	4	123	-
24	65	44	48	64	71	84	79	104	1	124	1
25	53	45	63	65	76	85	85	105	2	125	-
26	59	46	56	66	72	86	94	106	3	126	-
27	45	47	59	67	58	87	96	107	-	127	1
28	53	48	46	68	63	88	111	108	1	128	-
29	54	49	53	69	69	89	86	109	-	129	-
30	49	50	53	70	52	90	124	110	-	130	-
31	53	51	55	71	76	91	142	111	-	131	-
32	52	52	65	72	57	92	146	112	-	132	-
33	48	53	72	73	73	93	180	113	-	133	-
34	47	54	65	74	74	94	280	114	1	134	-
35	54	55	74	75	62	95	328	115	-	135	-
36	70	56	45	76	67	96	544	116	-	136	-
37	65	57	66	77	54	97	965	117	1	137	-
38	53	58	59	78	86	98	2812	118	-	138	-
39	51	59	77	79	63	99	10147	119	-	139	-
40	47	60	54	80	70	100	7228	120	-	140	-

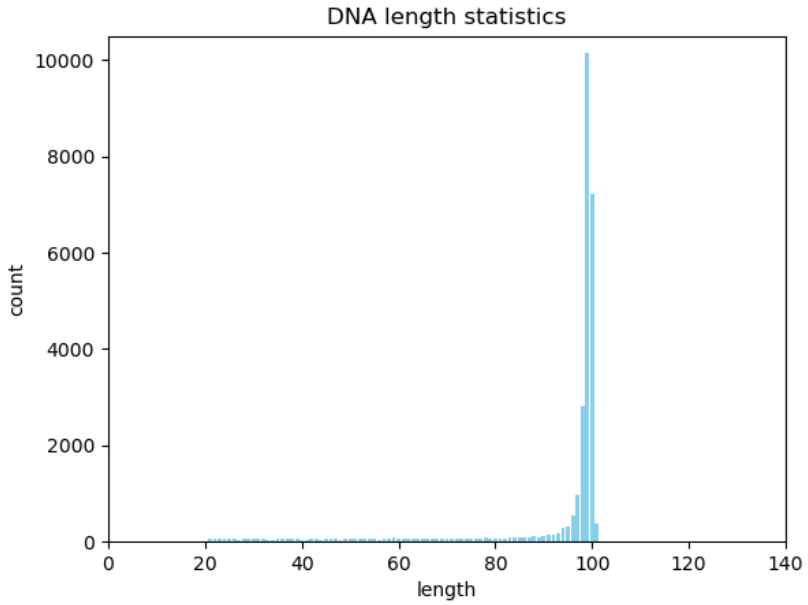

Figure 7: DNA length statistics

However, it is not necessary for us to implement the ideal preprocessing process in our desiged decodeing algorithm for this EE411 final project. PEAR is a fast and accurate paired-end read merger that uses a scoring matrix to penalize

mismatches and reward matches. It merges reads by maximizing the assembly score of the read overlap[3]. But we did not apply this technology. Collapsing identical sequence and storing the collapsed sequence and number of occurrences in the data, sorting the sequences based their abundance may be time-consuming. Since Kanagawa surfing files were cut into 1494 encoding segments(chunks), our algorithm only reads one line of DNA sequence from the file "50-SF.txt" each time and track the number of recovered chunks in real-time. If the number of recovered chunks reaches 1494, that is, decoding of all encoding fragments is completed, the decoding process is terminated. In our test, after reading 1821 lines, we finished decoding.

Listing 1 shows the initialization of some variables, where the meaning of 'encoded_file', 'decoded_file', 'seed_size', 'data_size', 'rscode_size' are the same as their literal meaning, 'chunk_num' is the total number of encoding chunks, 'count' is used for recording the number of segments that has be recovered, 'rs' implements Reed Solomon error correction encoding for detecting and correcting errors, 'chunks' stores all encoded chunks, 'prng' is a pseudo-random number generator implemented in utils/robust_solition.py, 'droplets' stores all droplets recovered from input DNA sequences, 'done_segments' stores all decoded chunks, 'chunk_to_droplets' maintains the droplets that contain the corresponding chunk.

```
encoded_file = '50-SF.txt'
decoded_file = '50-SF.jpg'
seed_size = 4
data_size = 16
rscode_size = 5
chunk_num = 1494
count=0
rs = RSCodec(rscode_size)
chunks = [None] * chunk_num
prng = PRNG(K=chunk_num, delta=0.05, c=0.1, np=False)
droplets = set()
done_segments = set()
chunk_to_droplets = defaultdict(set)
```

Listing 1: Initialization

2.2.2 Droplet Recovery

This section introduces the specific details of using **Python** to implement Droplet Recovery:

1. Translating $\{A, C, G, T\}$ to $\{0, 1, 2, 3\}$:
   Firstly, we replace the bases in the DNA sequence as follows: replace 'A' with '0', replace 'C' with '1', replace 'G' with '2', and replace 'T' with '3'. After the overall replacement, the string sequence is converted into a binary representation sequence. The four bases require a two bit binary representation, expanding '0' to '00', expanding '1' to '01', expanding '2' to '10', and expanding '3' to '11'. In this way, each base is uniquely determined by two bits of 01 data, in accordance with coding principles. Then, because the normal DNA sequence length is 100 bases, which includes address 16 bases, data 64 bases, and Reed Solomon encodes 20 bases, we divide the DNA sequence into 1 group for every 4 bases (or 8-bit 01 data), so address occupies 4 groups, data occupies 16 groups, and Reed Solomon occupies 5 groups.
2. Attempt to correct the substitution error with RS code:
   In Python, RSCodec is a library that implements Reed Solomon error correction encoding for detecting and correcting errors during data transmission or storage. Reed Solomon encoding is a powerful error correction technique commonly used in data communication and storage systems, especially in media damage or noisy environments. It allows for the detection and correction of a certain number of errors in data. We created an RSCodec object, specifying encoding and error correction parameters: rs=RSCodec (rscode_size), where rscode_size represents error correction capability, which can correct rscode_size group-errors.
3. Exclude the sequence with error, which is founded by RS code:
   Here we call the function rs.decode(byte_groups), which functions to decode byte_groups using the Reed Solomon algorithm. This method restores the original data by correcting possible errors. The parameter byte_groups passed to rs.decode should be the data encoded by the RS algorithm, and the decoded data should return the original byte_groups data. If rs.decode(byte_groups) fails to successfully correct potential errors, it will return an error or failure flag, usually indicating that the original data cannot be fully recovered, i.e. the program throws an exception, which is considered as the signal that we should exclude the sequence with error.
4. Extract seed, data payload from the sequence:

We extract seed and convert it from 32-bit binary to decimal, extract data payload from the corrected sequence. Since the RS code is used for correcting sequences, we have already used it in the previous steps and will not extract it here, as the RS code will not be needed in the future.

The Python implement of the process of Droplet Recovery is shown in Listing 2, where the class Droplet is implemented in utils/droplet.py:

```python
def droplet_recovery(dna):
    # Translating {A,C,G,T} to {0,1,2,3}
    base_mapping = {'A': '00', 'C': '01', 'G': '10', 'T': '11'}
    binary_string = ''.join(base_mapping[base] for base in dna)
    byte_groups = [int(group, 2) for group in textwrap.wrap(binary_string, 8)]
    try:
        # Attempt to correct the substitution error with RS code
        corrected_byte_groups = rs.decode(byte_groups)[0]
    except:
        # Exclude the sequence with error, which is founded by RS code
        return None, -1
    # Extract Seed, data payload from the sequence
    seed_bytes = ''.join(chr(byte) for byte in corrected_byte_groups[:seed_size])
    seed = sum(ord(char) << (8 * i) for i, char in enumerate(seed_bytes[::-1]))
    data = corrected_byte_groups[seed_size:]
    droplet = Droplet(data, seed, [])
    return droplet
```

Listing 2: Droplet Recovery Function

2.2.3 Segment Inference

First, we generate a list of segment identifiers, then run a message passing algorithm, which works as follows:

1. If the droplet contains inferred segments, the algorithm will XOR these segments from the droplet and remove them from the identity list of droplet
2. If the droplet has only one segment left in the list, the algorithm will set the segment to the droplet's data payload
3. Recursively propagate the new inferred segment to all previous droplets until no more updates are made
4. If the file is not recovered, the decoder will move to the next sequence in the file and execute the droplet recovery and segment inference

For the Python implement of Segment Inference, we divide the function into two parts: Identifiers Generate and Message Pass, as shown in Listing 3.

```python
def segment_inference(droplet):
    # Generate a list of segment identifiers
    identifiers_generate(droplet)
    # one round of message passing
    message_pass(droplet)
```

Listing 3: Segment Inference Function

In Identifiers Generate, we invoke prng.get_src_blocks_wrap() to get a list of segment identifiers which are indexes of encoded chunks. Then we add the droplet into droplets, update chunk_to_droplets. In Message Pass part, the process is described in 1, 2, 3. Listing 4 shows the Python implement details:

```python
def identifiers_generate(droplet):
    prng.set_seed(droplet.seed)
    droplet.num_chunks=set(prng.get_src_blocks_wrap()[1])
    droplets.add(droplet)
    for chunk_num in droplet.num_chunks:
        chunk_to_droplets[chunk_num].add(droplet)

def message_pass(droplet):
    # If the droplet contains inferred segments, XOR them,
```

```
    # and remove them from the identity list of droplet
    for chunk_num in (droplet.num_chunks & done_segments):
        droplet.data = map(operator.xor, droplet.data, chunks[chunk_num])
        droplet.num_chunks.remove(chunk_num)
        chunk_to_droplets[chunk_num].discard(droplet)
    # If the droplet has only one segment left,
    # set the segment to the droplet's data payload
    if len(droplet.num_chunks) == 1:
        lone_chunk = droplet.num_chunks.pop()
        chunks[lone_chunk] = droplet.data
        done_segments.add(lone_chunk)
        droplets.discard(droplet)
        chunk_to_droplets[lone_chunk].discard(droplet)
        # Recursively propagate the new inferred segment to all previous droplets
        # until no more updates are made
        for other_droplet in chunk_to_droplets[lone_chunk].copy():
            message_pass(other_droplet)
```

Listing 4: Identifiers Generate and Message Pass Function

If the file cannot be recovered, the decoder will proceed to the next sequence in the file, executing the droplet recovery and segment inference. The complete framework of the 50-SF Decoder is illustrated in Listing 5:

```
def decoder():
    with open(encoded_file, 'r') as f:
        for count, dna in enumerate(f, start=1):
            dna = dna.rstrip('\n')
            segment_inference(droplet_recovery(dna))
            if len(done_segments) >= chunk_num:
                break
    logging.info("After reading %d lines, we finished decoding!", count)
    outstring = ''
    logging.info("Restoring the picture now!")
    for x in tqdm(chunks):
        outstring += ''.join(map(chr, x))
    with open(decoded_file, 'wb') as f:
        f.write(outstring)
        logging.info("MD5 is %s", fp.get_md5(outstring))
    logging.info("Done!")
```

Listing 5: Framework of 50-SF Decoder

3 Result

We run EE411-Final-Project/50-SF_decoder.py in Linux 5.15.0-91-generic (buildd@lcy02-amd64-061) with python 2.7.18. We recover the image successfully, and check the md5 of both two file to verify the result. The md5 of decoded picture file 50-SF.jpg is **b4c92ee632d5be073e1fc0a16902bd7c**. Decoding results and some details is shown in Figure 8.

4 Work Division

Peijia Qin: Analysis of DNA storage encoding and decoding, Writing report
Zubin Zheng: Python implement of DNA storage decoding, Writing report

Acknowledgments

This project is supported in part by teaching team of EE411 in SUSTech at 2023 Fall.

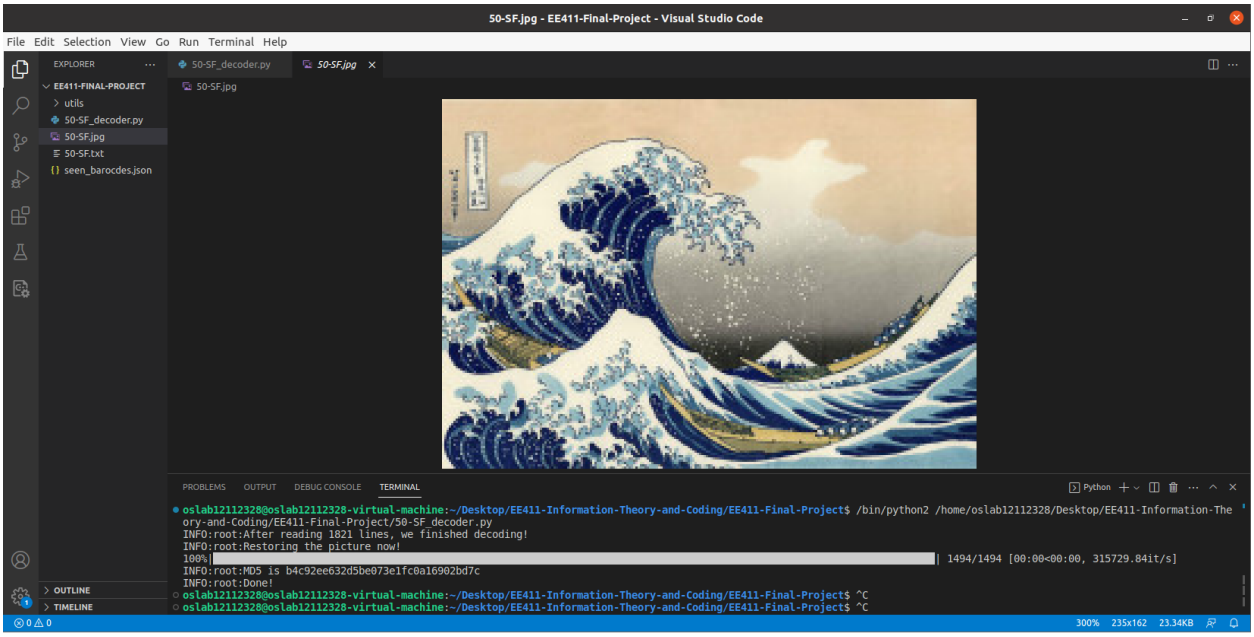

Figure 8: Decoding results

References

[1] Yaniv Erlich and Dina Zielinski. Dna fountain enables a robust and efficient storage architecture. *science*, 355(6328):950–954, 2017.

[2] Michael Luby. Lt codes. In *The 43rd Annual IEEE Symposium on Foundations of Computer Science, 2002. Proceedings.*, pages 271–271. IEEE Computer Society, 2002.

[3] Jiajie Zhang, Kassian Kobert, Tomáš Flouri, and Alexandros Stamatakis. Pear: a fast and accurate illumina paired-end read merger. *Bioinformatics*, 30(5):614–620, 2014.