PROJECT REPORT

Composite DNA Reconstruction using Hedges Error Correcting Code

Aviv Maayan, Nitai Kluger

With guidance from Omer Sabary, from the group of Prof. Eitan Yaakobi

Table of Contents

1. Abstract

Composite DNA letters were introduced in a recent paper [2] [4] [2.2]. The research presented synthesis and sequencing methods that exploit the built-in information redundancy of DNA data archiving. These methods involve using a logical enlarged alphabet, rather than the pure DNA bases only, and take advantage of the multiplication of DNA strands in current synthesis technologies to use fewer synthesis cycles per unit of data. In this project, we propose a new approach for DNA reconstruction [2.1] over composite DNA alphabets. Based on another recent paper "Hedges" [1] [2.3], we implemented a Hashed based Error Correcting Code (HECC) for encoding and decoding binary messages to and from composite alphabets. Our HECC handles all three basic types of DNA errors: substitutions, insertions, and deletions, up to an error rate of 2% while synthesizing only 40 copies of the original strand. When handling substitutions only, our HECC succeeds in decoding up to an error rate of 15% while synthesizing only 20 copies of the original strand.

We first present a short introduction to both the Composite and the Hedges papers. Secondly, we offer a detailed explanation of our encoding-decoding scheme and of the specific methods that we implemented. We then present our results and discuss some future possible optimizations to our algorithm.

2. Introduction

2.1. DNA Reconstruction

In DNA storage systems, the DNA reconstruction problem arises when a strand passes through a DNA-storage channel, introducing errors of deletions, insertions, or substitutions. The channel generates multiple noisy copies of the strand. A DNA reconstruction algorithm receives as input those copies and aims to produce an estimation of the original strand.

2.2. Composite DNA Letters

As described in a recent study [2], the inherent information redundancy in DNA-based storage systems could be used to extend the standard DNA alphabet to a larger "composite" DNA alphabet.

A composite DNA alphabet is a set of letters, where each letter is composed of a combination of one or more pure DNA bases (A, C, G, T), according to some predetermined ratio. Each letter is defined by a vector of size four that essentially represents a distribution over the DNA bases.

The synthesis of a composite DNA letter is done by using a mixture of the DNA bases according to the letter's distribution vector. Given that n copies are synthesized, each letter is written by n DNA bases. The distribution of those DNA bases reflects the letter's predetermined ratios.

2.3. Hedges

Hedges (Hash Encoded, Decoded by Greedy Exhaustive Search) is an error-correcting code for the pure DNA bases. Hedges was presented in a 2020 paper [1] and can correct all three DNA-storage channel errors with up to 10% error probability, while using a single copy of the original strand.

Hedges implements a "tree code" which integrates certain attributes tailored for the DNA channel. Decoding is accomplished using a heap algorithm that assigns costs to insertions, deletions, and substitutions. The success of the algorithm is probabilistic.

3. This Work

3.1. Combining Composite and Hedges

As explained above, the objective of this project is to reconstruct noisy copies of DNA strands generated utilizing a composite alphabet. To achieve this objective, we have opted to implement a modification of the tree code methodology used in Hedges.

Adapting the encoding phase of the Hedges framework required minor adjustments to accommodate our composite-alphabet configuration. However, the decoding phase demanded a comprehensive reassessment of the entire algorithm to effectively operate within the context of a composite alphabet and across multiple replicated sequences, as opposed to solely handling an individual strand.

The subsequent sections [3.2] [3.3] go into the details of the encoding and decoding techniques that are implemented in this project. Then, the "Two-sided encoding-decoding" approach is depicted [3.4]. This approach is an optimization for the encoding-decoding process that improved our results significantly.

3.2. Encoding

The encoding process receives a *message* of b bits to encode. For the sake of simplicity, it is assumed $b = 2t$, implying an even length, although the algorithm accommodates odd lengths easily (by adding a zero bit at the end of the message, for example). The primary objective of this phase is to generate an encoded strand, manifested as a vector containing composite letters derived from a predetermined *alphabet* denoted by Σ, where L signifies the size of the alphabet, expressed as $L = |\Sigma|$. In our implementation, an alphabet of size $L = 8$ is employed. The selection of alphabet size and constituent letters is deliberated in the Tuning section [5.1].

The encoding occurs sequentially, processing k_{in} bits at each step. Each k_{in} bits are translated into k_{out} composite letters using a process called "Hashed encoding" [3.2.2] which incorporates redundancy to facilitate error correction. Consequently, independent of the composite alphabet size, the length of the resultant strand is precisely $\frac{k_{out}}{k_{in}} \cdot b$ composite letters. In this work these parameters are set to $k_{in} = 2, k_{out} = 1$. In other words, every two bits are encoded to a single composite letter. Therefore, the output of encoding is half the length of the message, hence the simplicity of utilizing an even length.

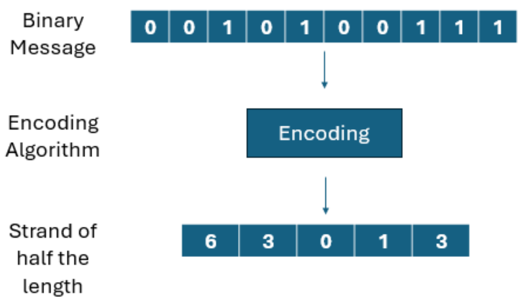
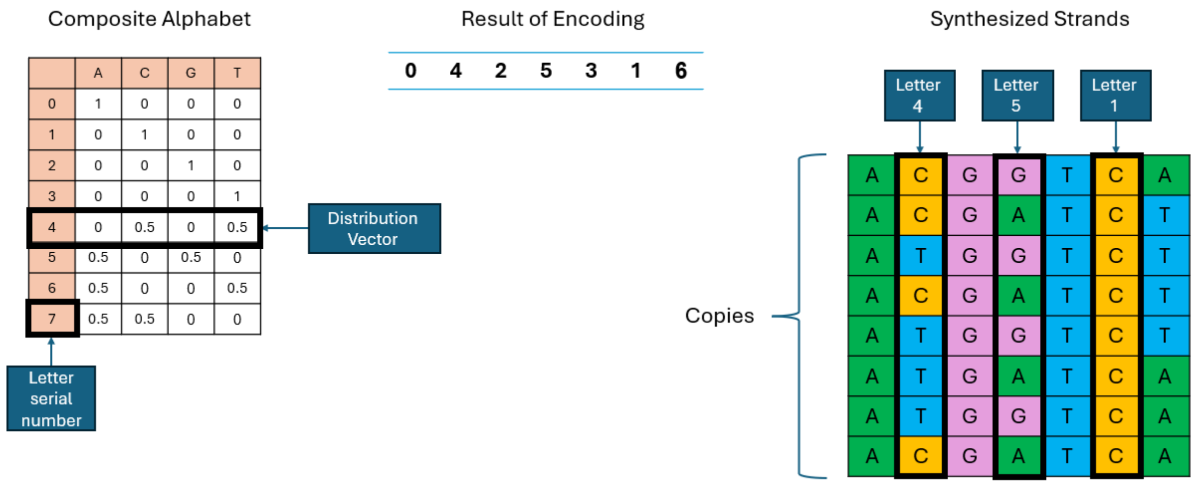


Illustration of the encoding input and output. The input is a binary message of an even length. Every two bits are encoded to a single composite letter. The output strand is composed of composite letters and is exactly half the length of the input message.

3.2.1. *Composite Letter Representation*

A composite letter $l \in \Sigma$ is defined as a four-dimensional vector, denoted as $l = (c_A, c_C, c_G, c_T)$, where each component c_i represents the weight of the DNA base i in the composition of the letter. For instance, the composite letter $l_{0,1} = (1,1,0,0)$ signifies a 50% probability of being A and a 50% probability of being C. These probabilities come to work during synthesis; a composite letter like $l_{0,1}$ synthesized x times, approximates a distribution of $\frac{1}{2}x$ bases of A and $\frac{1}{2}x$ bases of C. Notably, composite letters can be viewed as random variables following a multinomial distribution. In the case of $l_{0,1}$ it follows the distribution $l_{0,1} \sim Multi\left(x, \left(\frac{1}{2}, \frac{1}{2}, 0,0\right)\right)$. Another illustrative example is the composite letter $l_0 = (1,0,0,0)$, which represents the pure DNA letter A.

Each letter within the composite alphabet is assigned a serial number ranging from 0 to $L - 1$. This group is denoted by $[L] = \{0,1, \ldots, L-1\}$. Hence, after the encoding phase: $strand \in [L]^t$.



	A	C	G	T
0	1	0	0	0
1	0	1	0	0
2	0	0	1	0
3	0	0	0	1
4	0	0.5	0	0.5
5	0.5	0	0.5	0
6	0.5	0	0	0.5
7	0.5	0.5	0	0

A	C	G	G	T	C	A
A	C	G	A	T	C	T
A	T	G	G	T	C	T
A	C	G	A	T	C	T
A	T	G	G	T	C	T
A	T	G	A	T	C	A
A	T	G	G	T	C	A
A	C	G	A	T	C	A

The "Composite Alphabet" where each letter is assigned with a distribution vector. In this example, there are eight letters. The first four letters are the pure DNA bases, and the others are composed of two DNA bases each. The "Result of Encoding" is an example of the output of the encoding phase, composed of the letters' serial numbers. The "Synthesized Strands" are the copies of the encoding result (without any errors in this example). Each row is a copy, and each column is composed of synthesized DNA bases distributed according to the letter's multinomial distribution.

3.2.2. *Hashed Encoding*

Hashed Encoding is the process of encoding two bits (a "bit pair") into a single composite letter. The hashed encoding is done similarly to the process presented in Hedges. For each bit pair $p_i \in \{00,01,10,11\}$, a pseudorandom character $K_i \in \{0,1, \ldots, L-1\}$ is generated where each K_i depends, by a hash function, on the current position index i and a fixed number of previous message bits B_i:

$$K_i = hash(B_i, i)$$

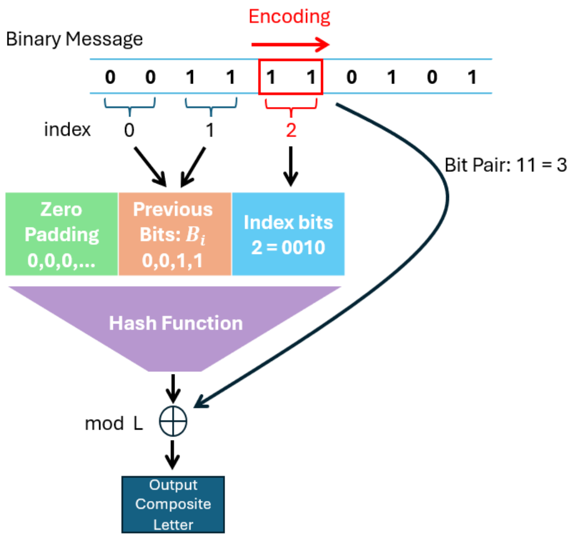
The number of previous bits utilized for hashing is ten, a parameter that remains unchanged throughout this project. Rather than adjusting this parameter, we have decided to maintain consistency with the Hedges algorithm, which reached notable results with this configuration. Regarding the first indexes that are below ten and do not have enough previous bits, we padded their B_i with zeros. We then emit a character C_i with:

$$C_i = K_i + p_i \quad (mod\ L)$$

The addition is performed modulo L so $C_i \in [L]$. The output DNA strand is composed from all C_i:

$$strand = (C_0, C_1, \ldots, C_{t-1})$$

This strand is always completely (pseudo)random because the hash is pseudorandom.



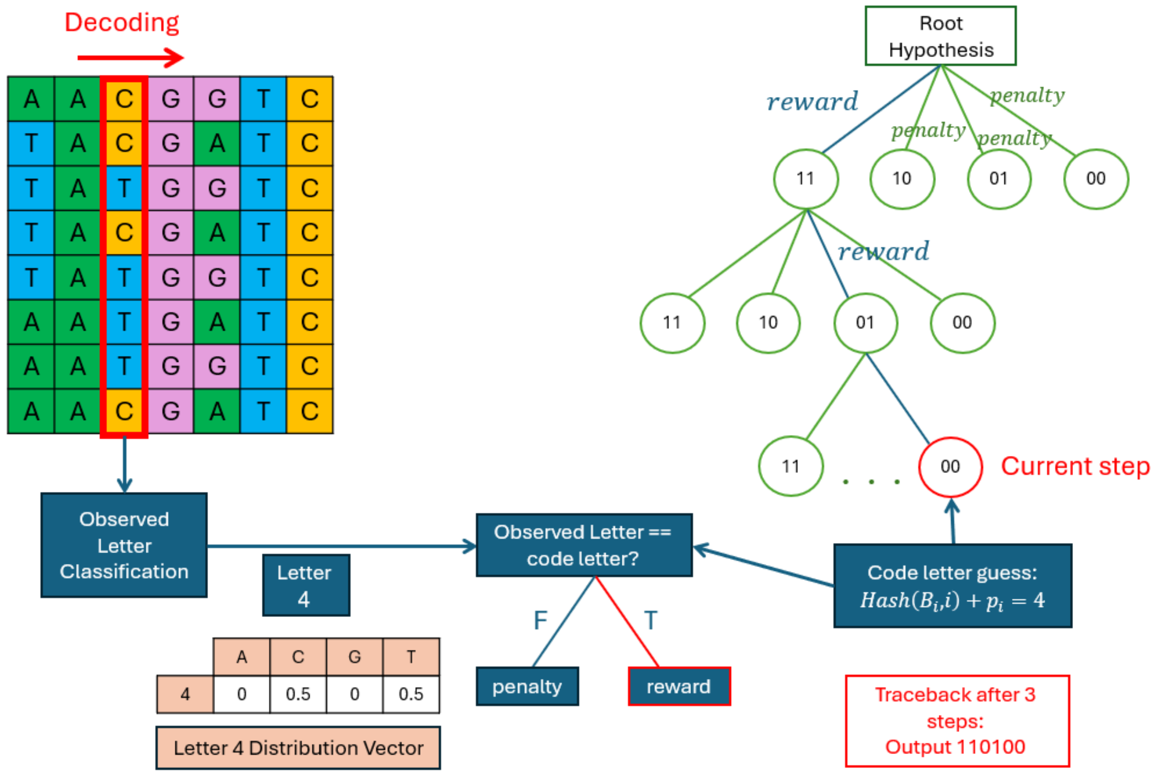
Example of the encoding of a single bit pair, in this case "11". When encoding this bit pair, indexed 2, there are only four previous bits. Therefore, we pad with zeros before using the hash function. The result of the hashing is added (mod L) to the bit pair value. The final number is the composite letter serial number.

3.3. Decoding

The decoding phase involves processing a set of n noisy copies, all originating from the same underlying strand. The number of copies is a tunable parameter, and our decoding algorithm has been tested across n values ranging from 10 to 60. The primary objective of the decoding phase is to reconstruct a singular strand from the disparate noisy copies.

Symmetrically mirroring the encoding process, the algorithm traverses the copies letter by letter, attempting to deduce the bit pair from which each letter originated. In each step, all four possible bit pairs $p_i \in \{00, 01, 10, 11\}$ are considered. For every bit pair guess, the algorithm compares its calculated hash-encoding result with the actual "observed" letter depicted by the copies. If there is a match, the bit pair is assumed correct, and the guess receives a "reward". Otherwise, it receives a "penalty". The observed letter is calculated in a process called "Observed letter classification" [4.2]. The sequential deductions and scores form chains of "Hypotheses", as each chain aims to predict the subsequent bits in the original strand. All chains are scored with a heap structure employed to manage these scores. The score of a chain is the summary of the scores of all the hypotheses that compose that chain.

In practice, the search for the best chain is a variant of the A* algorithm [3]. The decoding process performs an exhaustive search over the hypotheses tree, with a greedy heuristic of extending the search only for the best scoring chain at each step. This heuristic prevents the heap from growing exponentially. To further limit this potential growth, we have implemented a *limit* configurable variable that dictates the maximum size of the heap.



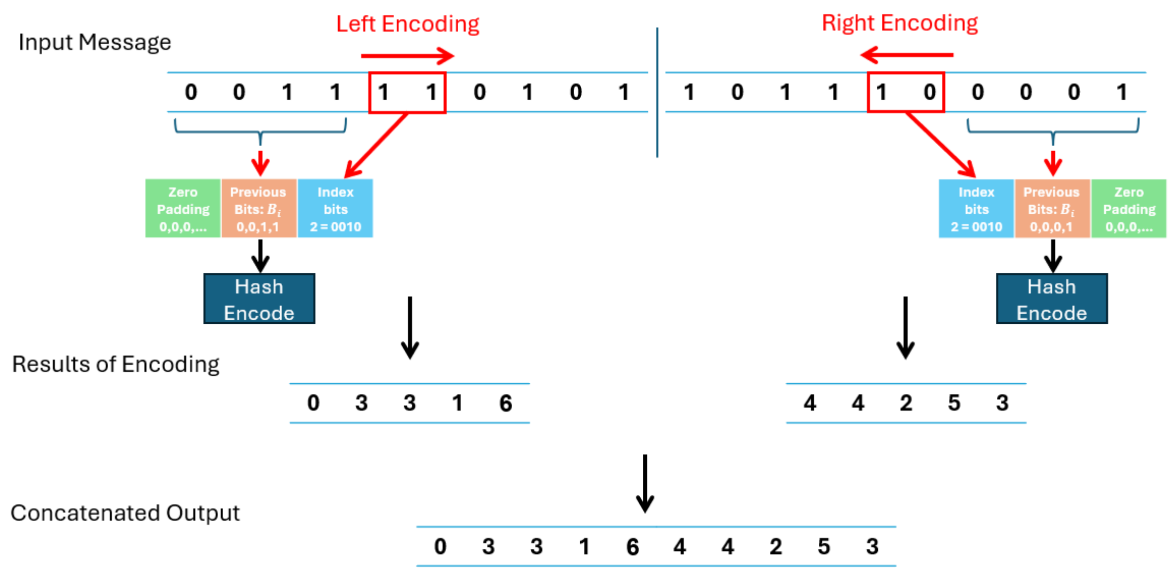
The decoding processes. The noisy copies are the colored array on the top left. Each letter is analyzed by the "observed letter classification" algorithm that determines the empirical composite letter imposed in each index. This letter corresponds with a unique distribution vector. The tree to the right illustrates the chains of hypotheses. At each step, the most rewarded chain of hypotheses is chosen to be extended with all 4 possible bit pair guesses. The algorithm calculates the code letter from the hash encoding result of this chain and assigns a reward or penalty to each successor. If the code letter and the observed letter are identical, the new hypothesis receives a reward. Otherwise, it is penalized.

The decoding algorithm ceases decoding for one of three reasons:

a. The algorithm reaches the final index. This means that decoding is finished successfully.
b. When all the hypotheses in the heap have lower scores than the original root hypothesis, this means that they had received much more penalties than rewards. In these scenarios the decoding algorithm signifies a failure and stops decoding.
c. If the heap of hypotheses extends the limit for its size (for this project, 10^6 was used) the decoding algorithm signifies a failure in decoding. This failure might be fixable through increasing the limit of the heap size and retrying.

3.4. Two-sided Encoding-Decoding

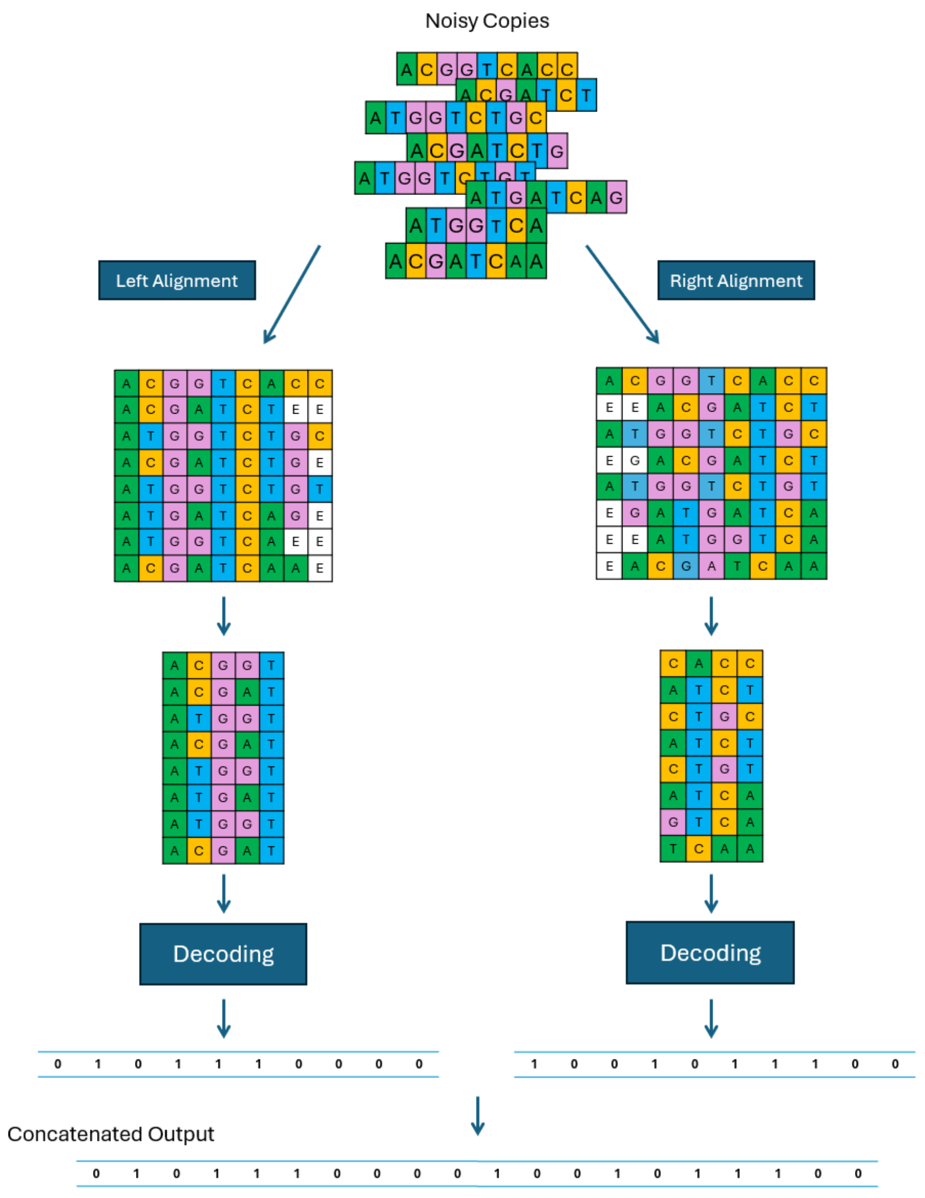
To improve our results, we implemented a dual application of the hashed encoding technique for each encoded message. Initially, the original message is partitioned into two distinct segments: the first half of the message (to the left), and the second one (to the right). Then, the encoding algorithm is employed independently on each segment. Notably, the first half undergoes encoding from left to right, while the second is encoded from right to left. Following this encoding process, we concatenate both encoded segments to obtain the ultimate encoded output.



Two-sided encoding illustration. The input message is divided in half and is encoded separately. The left side is encoded from left to right and the right side is encoded from right to left. The resulting output from each segment is concatenated to produce the final output.

To decode a message that was encoded via two-sided encoding, the n copies must first be divided to two segments. Given that the longest copy is of length l, The first $\left\lceil \frac{l}{2} \right\rceil$ indexes from all the copies are taken to create a "left" two-dimensional array. Similarly, the last $l - \left\lceil \frac{l}{2} \right\rceil$ indexes are taken to create the "right" array. The decoding algorithm is then applied over each array separately such that the left array is aligned and decoded from left to right, and the right array is aligned and decoded from right to left. The result of each decoding process is concatenated to provide the final output. For simplicity, we assume that l is a multiplication of 4 and hence: $\left\lceil \frac{l}{2} \right\rceil = \frac{l}{2} = l - \frac{l}{2} = l - \left\lceil \frac{l}{2} \right\rceil$ because l is even. Also, it therefore holds that both $\frac{l}{2}$ and $l - \frac{l}{2}$ are also even, maintaining the original assumption from the encoding phase.

Two-sided encoding decoding improved the algorithm's success rate significantly. With the two-sided approach, the algorithm managed to correct strands of up to two times the length it could beforehand, when applying the same error rate. Two-sided encoding decoding is useful against strand drifting and presents efficiency improvements.



Two-sided decoding illustration. The noisy copies are aligned to the left and to the right and divided in half accordingly. The left and right segments go through the decoding algorithm separately. The left segment undergoes decoding from left to right and the right segment undergoes decoding from right to left. The resulting output is the concatenation of the separate results.

3.4.1. Drifting of Strands

Drifting in DNA strands, arising from insertions and deletions, poses a well-known challenge. A strand is deemed "drifted" if the number of insertions differs from the number of deletions it has experienced. Consequently, the strand's alignment becomes disrupted, indicating drift.

Our decoding algorithm is prone to errors when confronted with drifted strands. Sampling the observed letter index by index becomes particularly problematic in such cases, as drifted strands significantly impact the sampling process negatively.

To handle the adverse effects of drifting, the approach of encoding only half of the original length at a time, as accomplished in the two-sided encoding decoding scheme, proves to be useful. The two-sided strategy enhances the likelihood of encountering fewer drifted strands. Aligning the strands both to the left and to the right ensures that the beginnings of each segment (left and right) are more likely to remain unaffected by drifting, thereby minimizing its impact on the decoding process. For that reason, the two-sided algorithm manages to decode longer binary messages correctly.

3.4.2. Performance and Efficiency

The decoding process described in [3.3] does not exhibit linearity efficiency with respect to the length of the strand. This is because the size of the heap of hypotheses may grow exponentially the longer the strand is. Therefore, simply shortening the strand by half and decoding each part as two-sided decoding does leads to improved efficiency, rather than decoding the entire original longer strand once.

Moreover, the two segments of the two-sided encoding (and decoding) technique are independent of each other. This allows for the possibility of executing both concurrently or in parallel, in both encoding and decoding phase. This parallel execution enables simultaneous decoding of both segments, thereby reducing the total processing time and improving overall throughput.

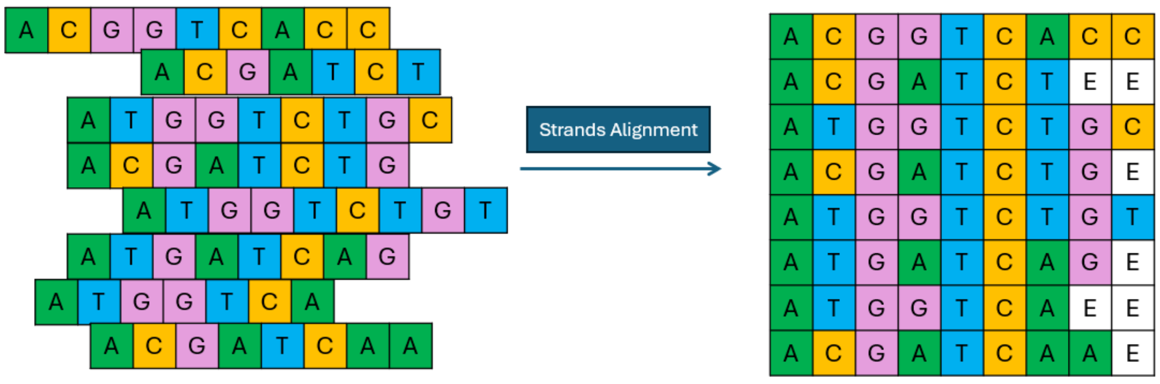
4. Methods

4.1. Strands Alignment Algorithm

Input: List of noisy copies

Output: Two-dimensional array of the noisy copies

To facilitate operations across the copies, the copies are aligned to conform to a two-dimensional array structure. Each row in this array corresponds to a noisy copy, and the length of the array is determined by the longest copy. Any shorter copies are padded with a null token. The strands are generally padded from left to right. When using two-sided decoding, the alignment is done both to the left and to the right, as explained in [3.4].



Strands Alignment process. In this example the strands are aligned to the left. Each strand is padded with a null token according to the longest copy to fit in a two-dimensional array.

4.2. Observed Letter Classification Algorithm

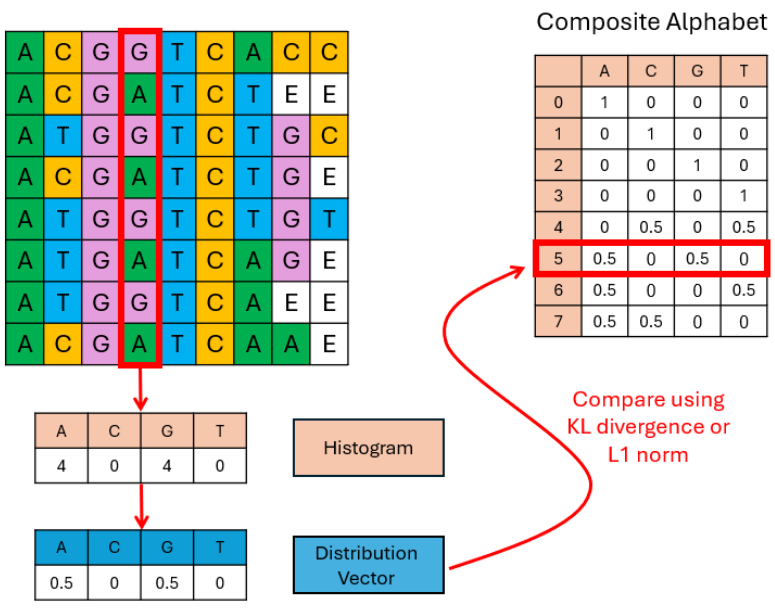
Input:

- Two-dimensional $array$ of noisy copies
- $index$ to examine

Output: Composite letter $\in \Sigma$

Observed Letter classification is the process of deducing which composite letter from the alphabet was synthesized in some index of the noisy copies array.

First, a histogram for the DNA bases present in column $index$ of the $array$ is generated. This histogram is then normalized to create a distribution vector across the pure DNA bases. The distribution vector is compared with the other distribution vectors of the letters from the alphabet. Using metrics such as Kullback-Leibler (KL) divergence or the L1 norm, the closest letter from the alphabet is inferred as the observed letter.



A	C	G	T
4	0	4	0

A	C	G	T
0.5	0	0.5	0

	A	C	G	T
0	1	0	0	0
1	0	1	0	0
2	0	0	1	0
3	0	0	0	1
4	0	0.5	0	0.5
5	0.5	0	0.5	0
6	0.5	0	0	0.5
7	0.5	0.5	0	0

General scheme of the "Observed Letter Classification" algorithm. First, creating a histogram from the relevant column of the noisy copies. The histogram is then normalized to a distribution vector and compared with each of the composite letters from the alphabet. The closest letter from the alphabet is the output of the algorithm.

Due to the presence of insertions and deletions in the noisy copies, relying solely on creating a histogram and sampling index by index may lead to suboptimal results. To address this challenge, we use a method that incorporates information from neighboring indexes to account for insertions or deletions, called "Multi-Strand Letter Sampling".

4.2.1. Multi-Strand Letter Sampling

Input:

- Two-dimensional $array$ of noisy copies $= S_{n\times l}$
- $index$ to examine
- $variance = \sigma^2$

- $mean = \mu$

Output: Composite letter $\in \Sigma$

The algorithm creates a histogram matrix $H_{4\times l}$ from S. Every column i of H is a four-dimensional vector that corresponds to the pure DNA bases' histogram from column i of S.

Then, we create the weights vector $\overline{w}_{index}$. It is a normalized column vector with size $l \times 1$ entries, and each entry holds a weight between zero and one.

The weight vector is dependent on the given $index$ and thus consists of "heavier" weights around the $index$ row. Multiplying the histogram matrix with $\overline{w}_{index}$ to classify the observed letter effectively enables the influence of nearby indexes on the calculation process. The result of this multiplication is a four-dimensional vector that imputes a distribution among DNA bases:

$$H \cdot \overline{w}_{index} = (\alpha_A, \alpha_C, \alpha_G, \alpha_T)^T$$

To set the weights of $\overline{w}_{index}$, we have implemented a discrete Gaussian kernel with the given $variance$ and $mean$ parameters. To limit the effect from distinct indexes, only close indexes are given a non-zero weight. For some $\epsilon \in \mathbb{N}$ $s.t$ $\epsilon << \frac{l}{2}$, the group of indexes that has non-zero weights in $\overline{w}_{index}$ is:

$$M = [index - \epsilon, index + \epsilon]$$

We have implemented ϵ to be a configurable parameter and used $\epsilon = 3$ for most of our tests. The entries $w_{index}^{i} \in \overline{w}_{index}$ are defined as such:

$$\forall i \in [l]: \quad w_{index}^{i} = \begin{cases} \Phi_{\mu,\sigma}\left(i + \frac{1}{2}\right) - \Phi_{\mu,\sigma}\left(i - \frac{1}{2}\right), & i \in (index - \epsilon, index + \epsilon) \\ \Phi_{\mu,\sigma}\left(i + \frac{1}{2}\right), & i = index - \epsilon \\ 1 - \Phi_{\mu,\sigma}\left(i - \frac{1}{2}\right), & i = index + \epsilon \\ 0, & otherwise \end{cases}$$

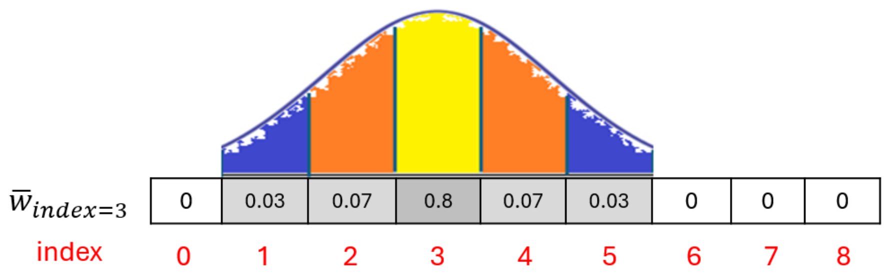
When $\Phi_{\mu,\sigma}$ is the CDF of the normal distribution with the given $mean$ and $variance$.

Therefore, it holds that the weights vector is normalized:

$$\sum_{i=1}^{l} w_i = \sum_{i \in M} w_i = \sum_{i=index-\epsilon}^{index+\epsilon} w_i = \Phi_{\mu,\sigma}\left(index - \epsilon + \frac{1}{2}\right) + \sum_{i=index-\epsilon+1}^{index+\epsilon-1} w_i + 1 - \Phi_{\mu,\sigma}\left(index + \epsilon - \frac{1}{2}\right) =$$

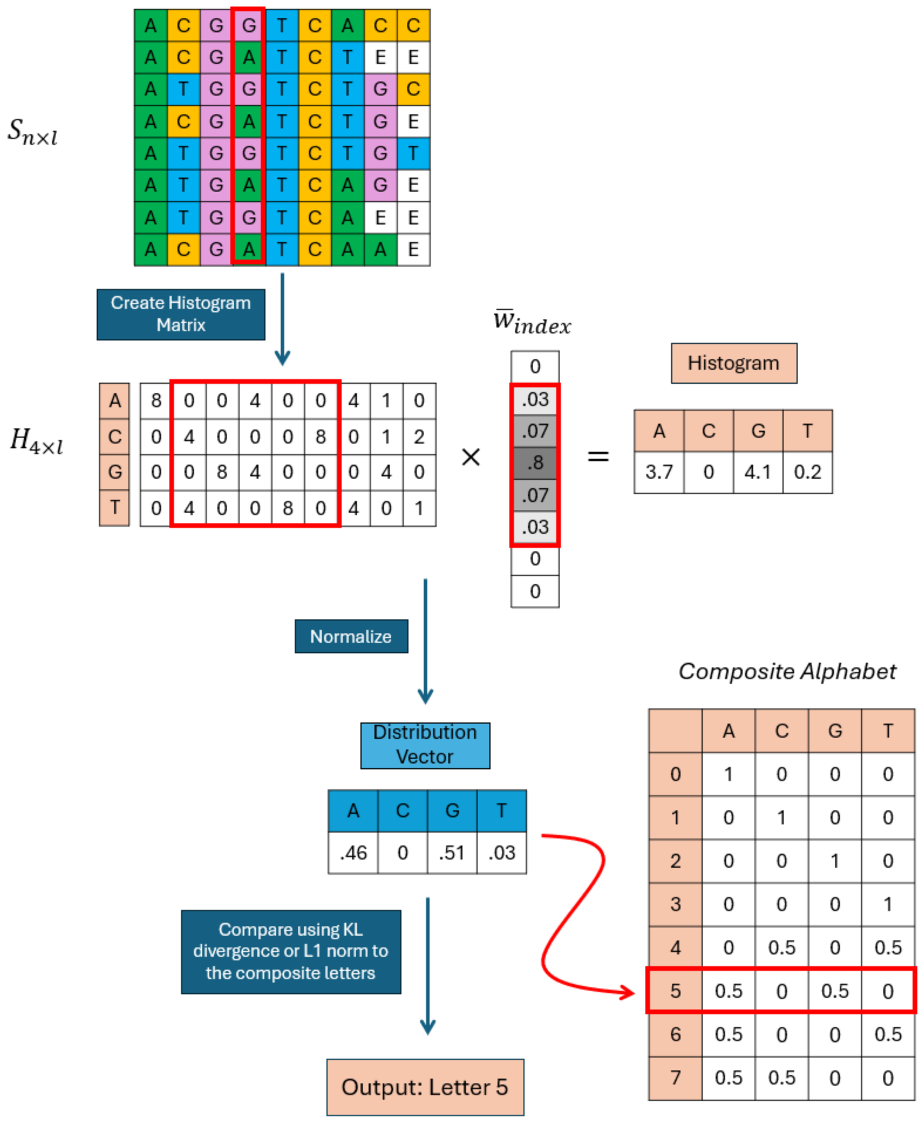
$$= \Phi_{\mu,\sigma}\left(index - \epsilon + \frac{1}{2}\right) + \sum_{i=index-\epsilon+1}^{index+\epsilon-1} \left(\Phi_{\mu,\sigma}\left(i + \frac{1}{2}\right) - \Phi_{\mu,\sigma}\left(i - \frac{1}{2}\right)\right) + 1 - \Phi_{\mu,\sigma}\left(index + \epsilon - \frac{1}{2}\right) =$$

$$=_{telescopic\ sum} 1$$



index	0	1	2	3	4	5	6	7	8
$\overline{w}_{index=3}$	0	0.03	0.07	0.8	0.07	0.03	0	0	0

The weights vector $\overline{w}_{index}$ for $index = 3,\ \mu = 3,\ l = 9$. The weights of each entry in the vector are heavier the closer the index is to the mean, which is three. Notably, the summary of all weights is one. In this example there are five weights that are larger than zero, hence: $M = [1, 5] = [3 - 2, 3 + 2]$ and thus $\epsilon = 2$. This illustration depicts $\overline{w}_{index}$ as a row vector for simplicity, but in practice it is a column vector as explained above.



$S_{n\times l}$

A	C	G	G	T	C	A	C	C
A	C	G	A	T	C	T	E	E
A	T	G	G	T	C	T	G	C
A	C	G	A	T	C	T	G	E
A	T	G	G	T	C	T	G	T
A	T	G	A	T	C	A	G	E
A	T	G	G	T	C	A	E	E
A	C	G	A	T	C	A	A	E

$H_{4\times l}$

A	8	0	0	4	0	0	4	1	0
C	0	4	0	0	0	8	0	1	2
G	0	0	8	4	0	0	0	4	0
T	0	4	0	0	8	0	4	0	1

$\overline{w}_{index}$

0
.03
.07
.8
.07
.03
0
0

Histogram

A	C	G	T
3.7	0	4.1	0.2

Distribution Vector

A	C	G	T
.46	0	.51	.03

Composite Alphabet

	A	C	G	T
0	1	0	0	0
1	0	1	0	0
2	0	0	1	0
3	0	0	0	1
4	0	0.5	0	0.5
5	0.5	0	0.5	0
6	0.5	0	0	0.5
7	0.5	0.5	0	0

Multi-Strand Letter Sampling process. In this example, the observed letter classified is the letter at index three. The histogram matrix H is created from the copies. H is multiplied with $\overline{w}_{index}$, in this case $index = 3$. The resulting vector is a histogram of the DNA bases and is affected by multiple adjacent indexes from all copies. This vector is normalized to create a distribution vector. It is then compared with all the distribution vectors of the composite alphabet, using KL *divergence or L1 norm. The observed letter that's returned as an output is the closest letter from the composite alphabet.*

5. Tuning

5.1. Composite Parameters

5.1.1. Alphabet Size

Tuning the alphabet size involves a significant trade-off. On one hand, a larger alphabet increases the code's internal redundancy. This is because with the hashed-encoding technique, the probability that a random hypotheses chain produces the same output C_i as the "correct" chain is inversely related to the size of the hash function's codomain. Expanding the composite alphabet size enlarges the codomain, thereby enhancing error correction capabilities.

However, expanding the alphabet can also make the letters' distribution vectors more similar, complicating the classification of observed letters. This increased similarity can negatively impact the decoding process by causing rewards and penalties to be incorrectly assigned to hypotheses.

Choosing a composite alphabet larger than eight letters proved to be too difficult for our observed letter classification algorithm. The letters were too similar, and we encountered many errors in classification. Conversely, an alphabet smaller or equal to six was too small. This is because each hypothesis guesses among four different bit pairs, which results in four different composite letters. In practice, this means every hypothesis guesses four out of six (or less) possible letters in every step, which was almost as exhaustive as simply trying all possible letters. This made the algorithm assign rewards and penalties almost at random. Eventually, the alphabets of sizes seven and eight were the most adequate. To increase the error correcting redundancy, we chose size eight.

5.1.2. Letter Selection

Using the notation from the Composite paper [2], we limit the possible alphabets using a resolution parameter. First, we define: $\Sigma_k = \{\sigma = (c_A, c_C, c_G, c_T) \mid k = c_A + c_C + c_G + c_T\}$. The size of Σ_k is calculated as the number of possible combinations of inserting k balls into four bins, which is:

$$|\Sigma_k| = \binom{k+3}{k}$$

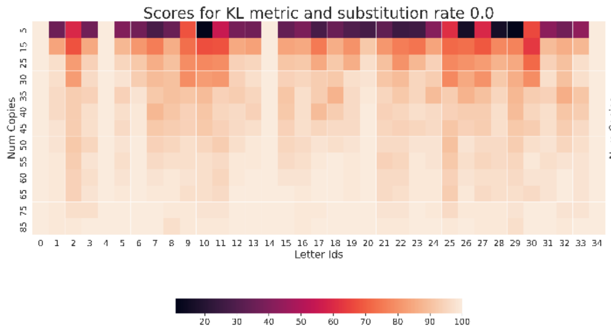
We have decided that our alphabet Σ will be composed as $\Sigma \subseteq \Sigma_k$ for some $k \geq 2$. After initial testing we decided that choosing $k = 4$ entailed set of letters large enough to choose from but not too large for testing runtime efficiency:

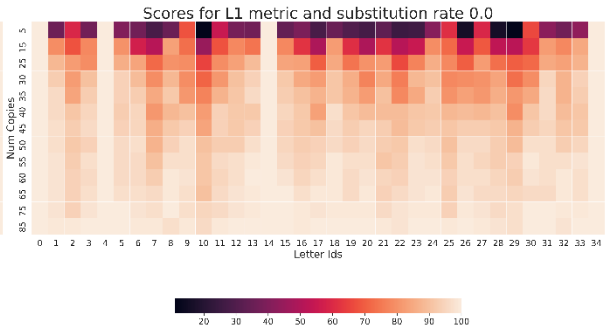
$$|\Sigma_4| = \binom{4+3}{4} = \binom{7}{4} = 35$$

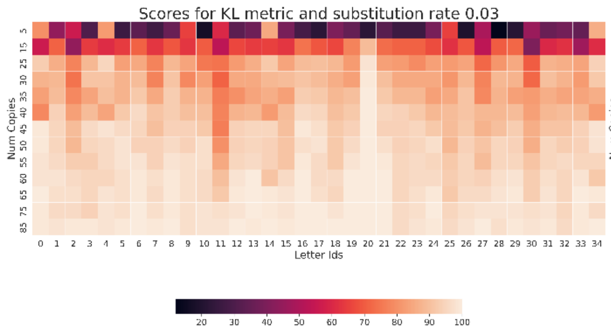
The initial tuning test conducted involved the classification of individual letters from the set Σ_4, utilizing both KL divergence and L1 norm metrics. Each letter was synthesized across varying numbers of copies, allowing the classification algorithm to determine the most similar letter from Σ_4. This test was repeated multiple times with different substitution error rates.

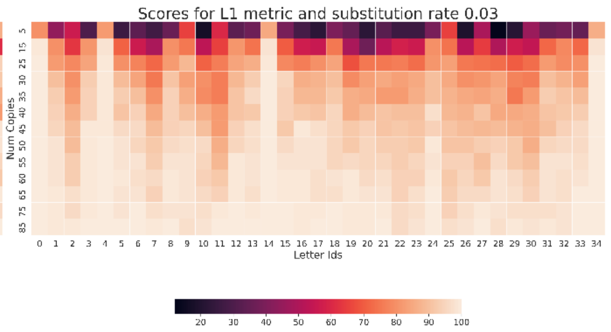
Separate tests were conducted for the L1 norm and KL divergence. An intriguing characteristic of KL divergence is its handling of zero probabilities, which is not optimal. The conventional approach to address this issue, as outlined in the Composite paper, involves employing a form of Laplace smoothing. This technique involves replacing each zero probability with a small value denoted as ϵ_0. Despite the incorporation of ϵ_0 enhancing the classification success rate, it still yielded inferior scores compared to L1 norm when large error rates were introduced.

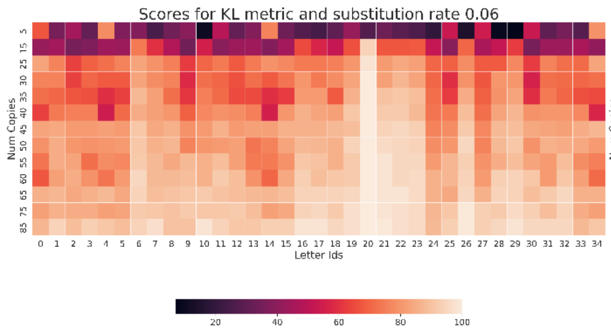
The results of this test are depicted below. The x-axis represents the sequential numbering of letters in Σ_4, ranging from 0 to 34. The y-axis indicates the number of synthesized copies. For each (x, y) pair, the classification algorithm was executed 100 times. The color coding within each square denotes the number of successful classifications of the letter. The headline of each graph denotes the introduced error rate.

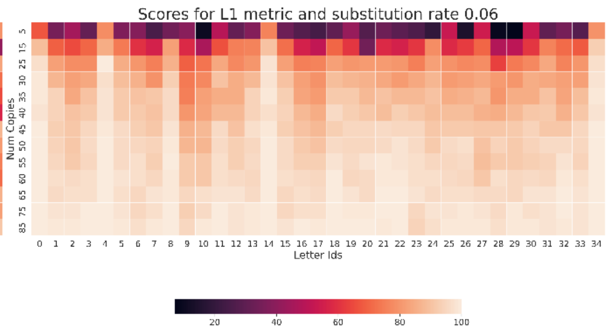












Notably, for low error rates KL was better than L1 and for high error rate it was L1 that showed better results.

After conducting this test, we employed a heuristic strategy based on distance metrics between letter pairs to determine the best eight-letter subset. We exhaustively evaluated all possible subsets to find the optimal solution, first by maximizing the minimal distance between all letter pairs and then by maximizing the sum of distances between all letter pairs.

We chose the set of eight letters that yielded the best results. The four best scoring letters were the four pure DNA bases. The subsequent four letters were chosen from the "pair" letters, defined as those having a distribution vector consisting solely of two primary DNA bases with equal probabilities. For instance, this includes $l_{0,1}$ that was described above [3.2.1].

Therefore, the chosen composite alphabet for this project is:

$$\Sigma = \{(4,0,0,0), (0,4,0,0), (0,0,4,0), (0,0,0,4), (2,2,0,0), (0,0,2,2), (2,0,0,2), (0,2,0,2)\}$$

And the resultant distribution vectors are:

$$\{(1,0,0,0), (0,1,0,0), (0,0,1,0), (0,0,0,1), (0.5,0.5,0,0), (0,0,0.5,0.5), (0.5,0,0,0.5), (0,0.5,0,0.5)\}$$

5.2. Hedges Parameters

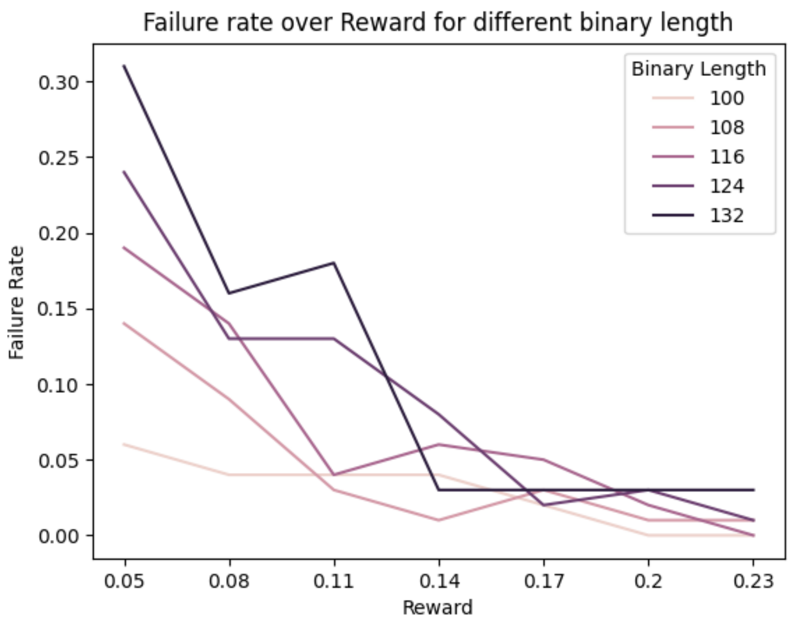
5.2.1. Reward and Penalty

At each iteration of the decoding algorithm, it assigns rewards and penalties to the hypotheses it examines. This determines the subsequent hypotheses to be extended. Hence, the heap of hypotheses topology within the decoding algorithm heavily relies on the sizes of rewards and penalties.

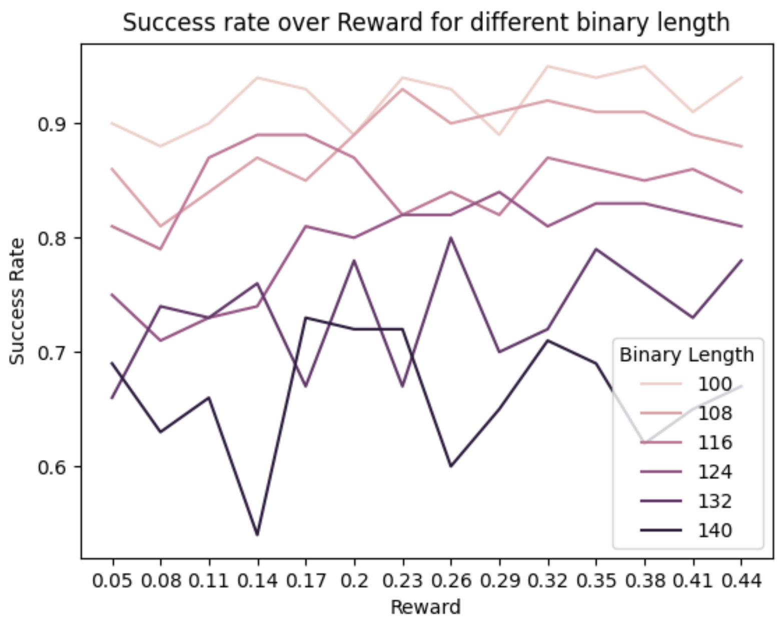
A situation where the reward is deemed "small" implies that despite receiving multiple rewards, hypotheses would exhibit minimal enhancements in their overall scores, as a few penalties could significantly diminish their standing.

Conversely, a scenario characterized by a "large" reward suggests that even a small number of rewards could elevate a hypothesis to a high status, with numerous penalties insufficient to substantially lower its score.

We have examined these parameters by setting the penalty to a fixed value of minus two and testing our algorithm with a reward ranging from 0.05 to 0.44. This test was performed multiple times under varied environmental conditions, changing parameters such as the number of copies and error rates.



Failure rate by Reward. The failure rate is defined as the percentage of decoding runs that ended with a failure, i.e. there was no output from the decoder. The colors are the lengths of the *encoded binary message. As expected, the higher the reward is the less likely the decoding fails since the standings of the hypotheses is very high.*



Success rate by Reward. The success rate is defined as the percentage of decoding runs that ended with a success, i.e. the message was correctly decoded. From this test it is noticeable that the success rate is generally similar for different rewards.

From these results we concluded that the tradeoff of reward tuning is between "failures" and "mistakes". A failure is when decoding has not reached the end of the message and there is no return value. A mistake is when decoding returned a value with some substitution errors. Since substitution errors could be fixable by adding an ECC to our algorithm, we decided to set a value of reward that reached a relatively small number of failures. Eventually, reward of size 0.3 was chosen.

6. Optimizations

6.1. Second-best Letter Reward

Our decoding algorithm extends hypotheses based on a comparison between the calculated code letter and the empirical observed letter. This approach assumes the observed letter is always accurate, rewarding only the bit pair that yields a match.

An optimization to this process involves enhancing the observed letter classification algorithm to return not just the most similar letter, but also the second most similar letter from the composite alphabet. Subsequently, the comparison rewards a "best match" and a "second best match" between the code letter and the two observed letters. The size of the reward for the second match is a configurable parameter.

This optimization does not eliminate any potential hypotheses but alters the scoring of others. Consequently, the algorithm remains at least as robust as before and potentially much stronger. Upon implementation, this feature showed an improvement in results. However, it should be noted that having more hypotheses with higher scores extends the algorithm's computational time. The parameter governing the size of the reward for the second match was not tuned during our experimentation and is left as a subject for future investigation.

6.2. Mean Shifting

The observed letter classification algorithm relies on multiple indices as explained in Multi-Strand Letter Sampling [4.2.1]. Central to this approach is the employment of a Gaussian kernel characterized by a fixed $variance$, and a $mean$ value spanning the spectrum of all index values. However, this method eventually wears out when the strands are heavily drifted.

To further optimize this process, an alternative strategy called "Mean Shifting" can be employed. Instead of exclusively sampling the observed letter at index i with a mean equal to i, an additional shift parameter can be introduced, so the mean equals to $i + shift$.

This concept comes into play during the decoding phase when extending a hypothesis with new successors. Let δ be a (configurable) small increment. The shifts for the successors are derived from the hypothesis's shift, with three options: maintaining the predecessor shift value, increasing the shift value by δ, or decreasing it by δ. Hence, upon sampling the observed letter at index i, each new child hypotheses uses a different mean value, spanning the range: $[i - i \cdot \delta, i + i \cdot \delta]$.

The introduction of mean shifting results in the creation of four successors for each mean shift, yielding a total of twelve child hypotheses. Although this augmentation substantially increases both the heap size and computation time, it tends to yield slightly improved outcomes. This is because Mean Shifting allows sampling multiple copies while considering that they may drift from their original index, hence potentially improving the accuracy.

Future investigations in this domain hold promise for resembling the +1 and −1 shifts employed by Hedges. This optimization could improve the success rate of decoding significantly.

6.3. Reed-Solomon and other ECC

In this project, the incorporation of supplementary Error Correction Codes (ECC), such as Reed-Solomon, was not executed. These codes could readily be applied to the initial binary message prior to hash-encoding, and subsequently decoded from the binary data after completion of the decoding process.

This augmentation of ECC can correct numerous substitution errors that the decoding process could not resolve. We leave the addition of ECC as further work in this topic.

7. Results

7.1. Testing Framework

The project is executed using the Python programming language. Pytest, a testing library, is utilized due to its simple syntax that is easy to understand, and since it helps in testing various scenarios without writing repetitive code.

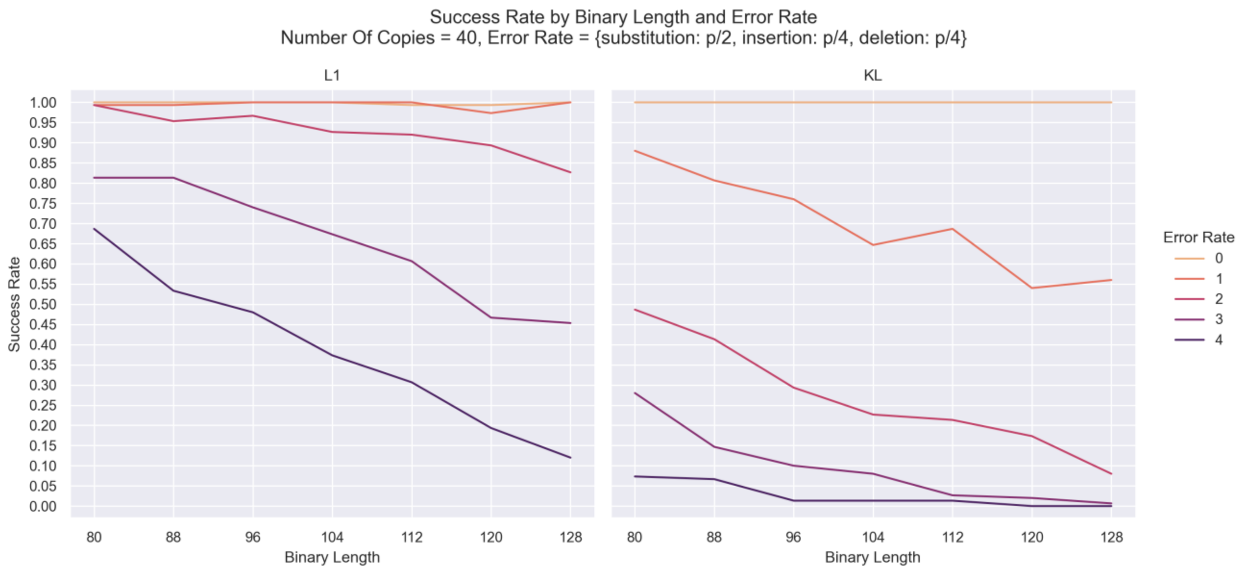
All binary messages subjected to encoding are randomly generated. After encoding phase, the encoded strand is transmitted through an error channel implemented for this purpose. This error channel first synthesizes the requested number of copies based on the distribution vector associated with each letter. Then, errors are introduced independently at each index of every copy, according to the error rate of the given test.

7.2. Results by Environment Parameters

In all the tests, the decoding algorithm is subjected to three changing parameters at most – number of copies synthesized, original message binary length, and error rate probabilities. The headline of each graph describes which of those parameters are constant and which are changing. The y-axis is the success rate of the algorithm, where success means perfect decoding.

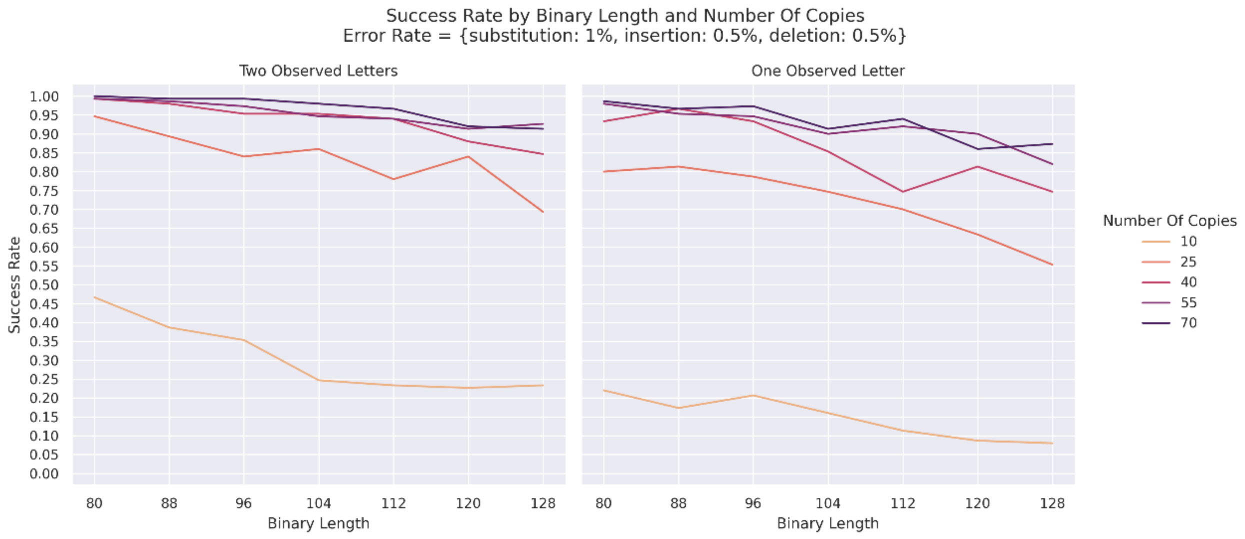
7.2.1. Comparison of KL vs. L1

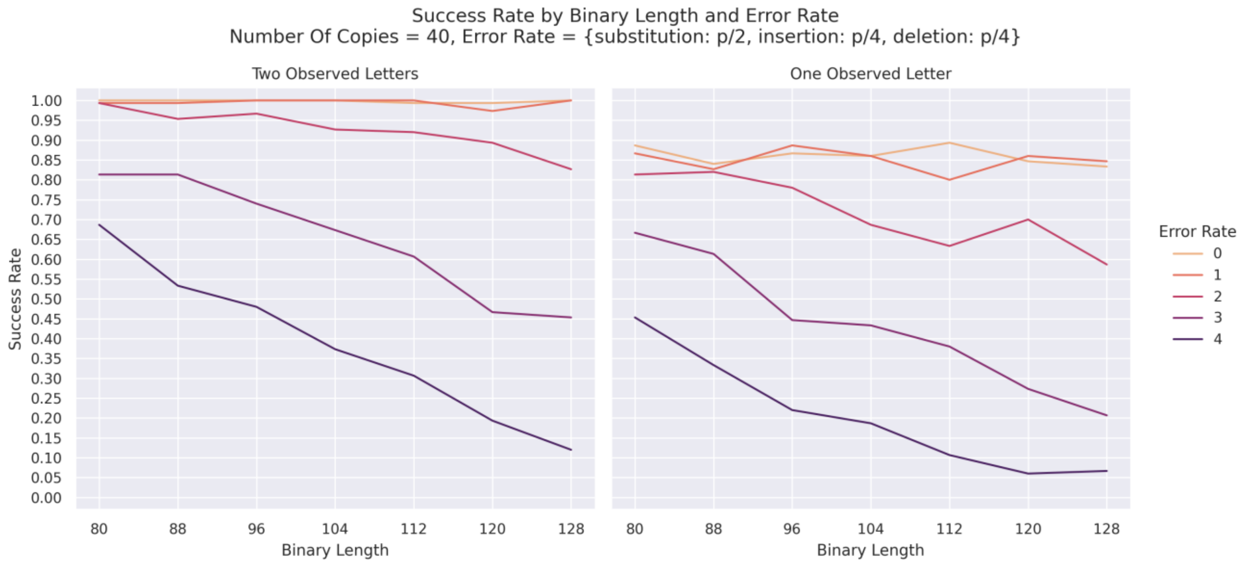
As shown in the letter selection section [5.1.2], it is notable that KL divergence does not perform as well as L1 for error rates that are not very low. As depicted in the following graphs, KL indeed presents much lower success rates for all error rates besides zero. Hence, for the rest of the tests we used the L1 norm only.



7.2.2. Comparison with Single vs. Second-best Letter Reward

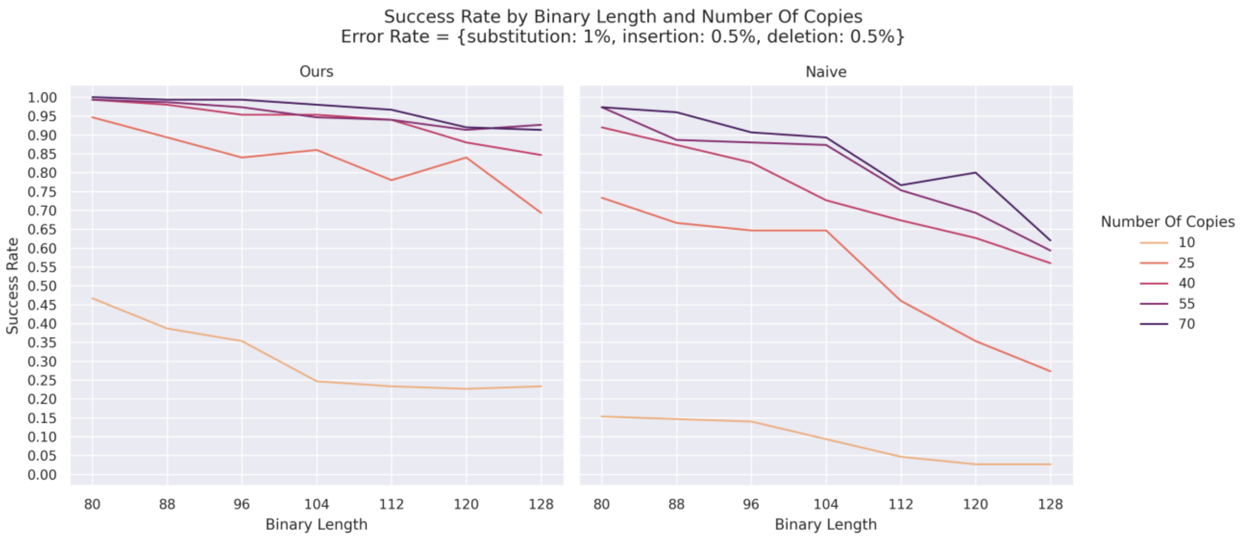
These results present the comparison between having a single reward and two rewards – as explained in the previous section [6.1]. Notably, this had a positive impact and hence for all other results and comparisons in the next sections the two-rewards approach is applied. The second reward is tuned to a size of approximately half of the first reward.

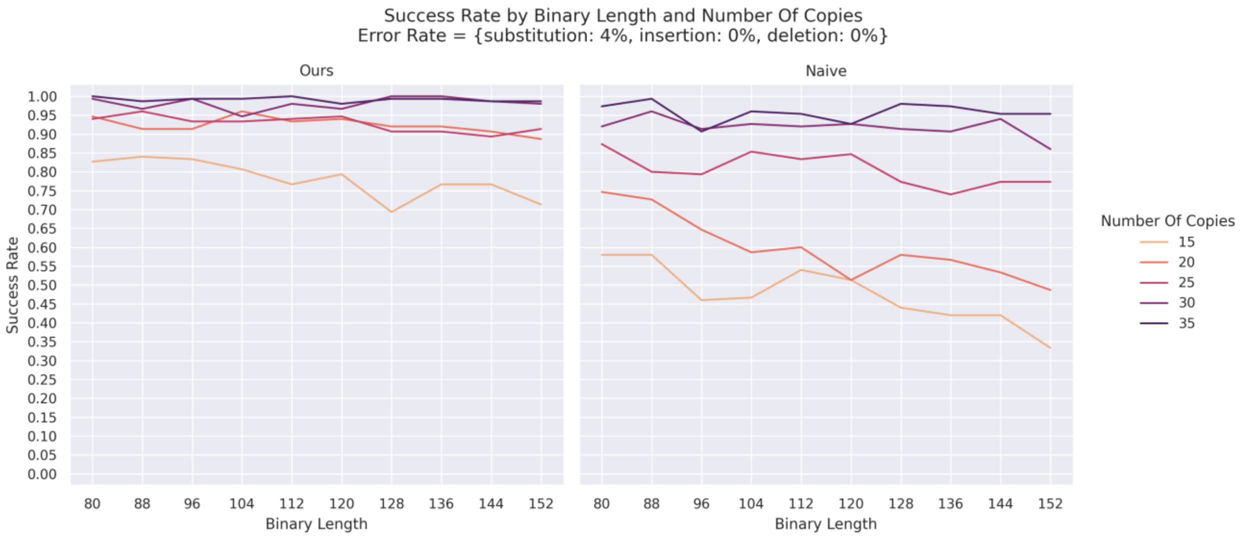


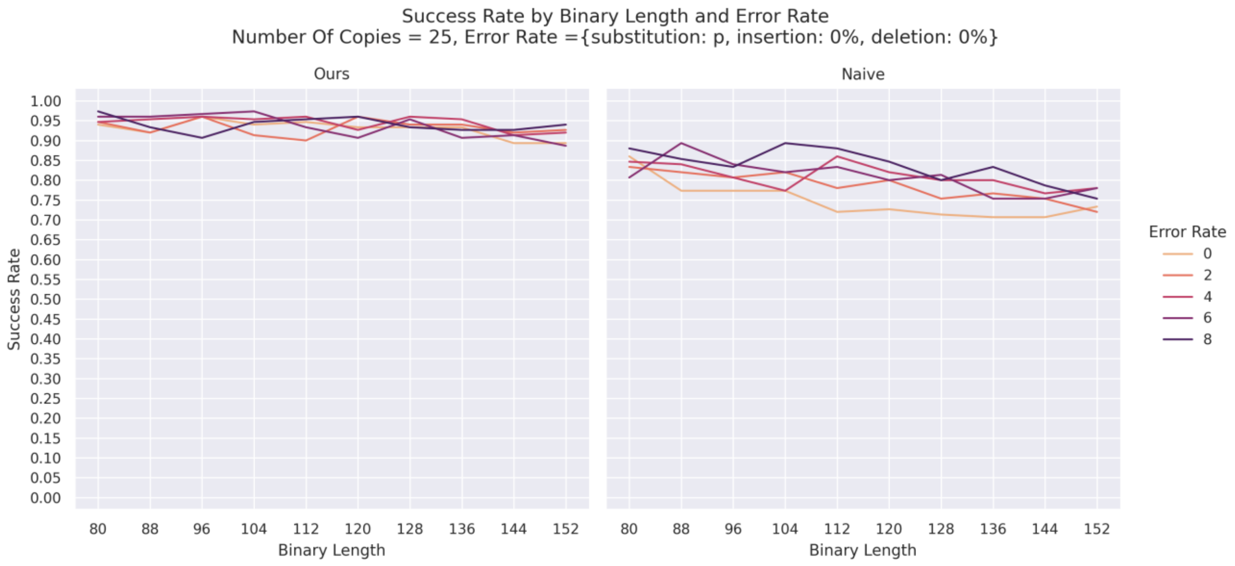


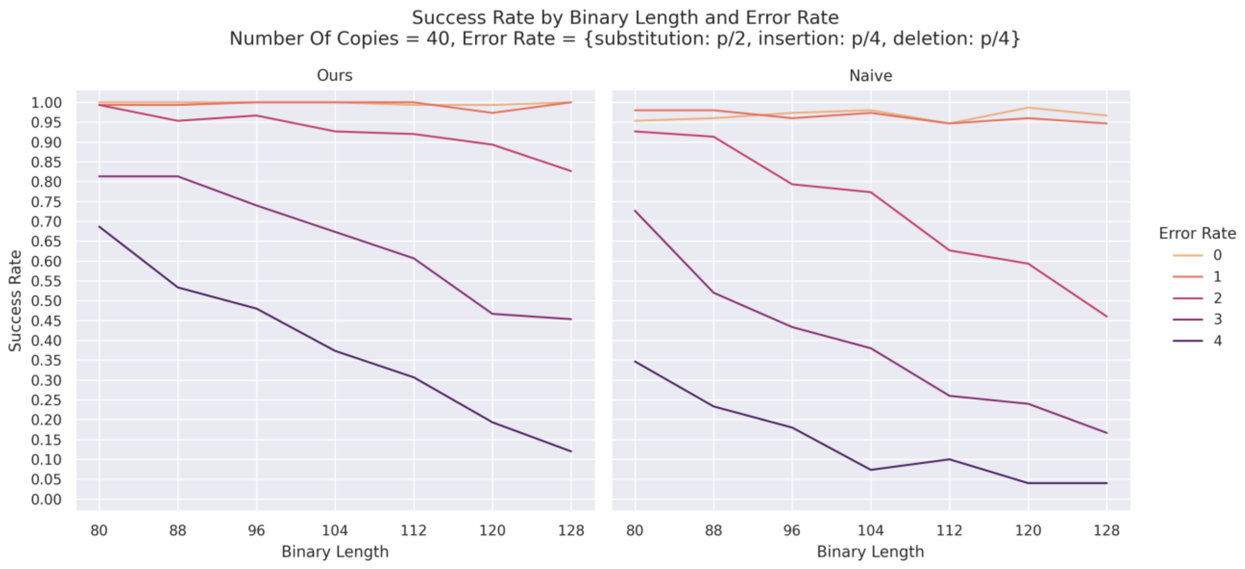
7.2.3. Comparison to the Naïve approach

To compare the results of our algorithm with a naïve approach, we have implemented a naïve model of encoding-decoding. This model receives the same binary message and encodes it to the same composite alphabet as our algorithm. It involves encoding every three bits of information to a single composite letter with a unique mapping. There are eight letters in the alphabet and hence this encoding is possible and indeed unique. The decoding phase is similar to the one presented in the composite paper. For every index, the naïve algorithm generates a distribution of DNA bases across the copies and uses KL or L1 to find the closest letter from the alphabet.


Success Rate by Binary Length and Number Of Copies
Error Rate = {substitution: 1%, insertion: 0.5%, deletion: 0.5%}
Ours
Naive
Success Rate
Binary Length
Number Of Copies
10
25
40
55
70

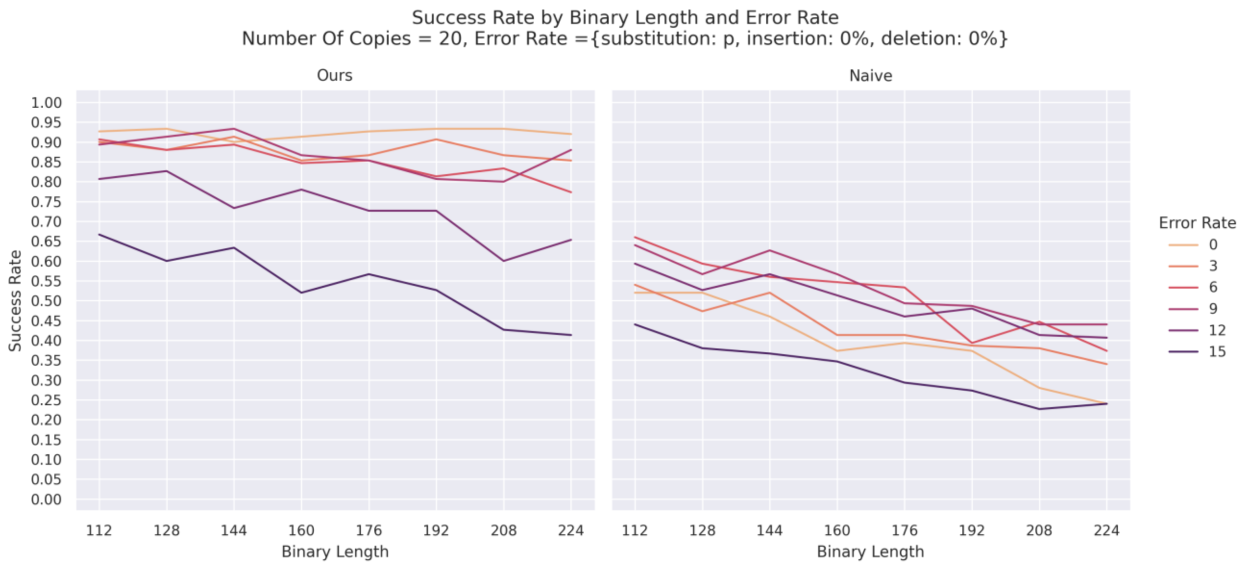

Success Rate by Binary Length and Number Of Copies
Error Rate = {substitution: 4%, insertion: 0%, deletion: 0%}
Ours
Naive
Success Rate
Binary Length
Number Of Copies
15
20
25
30
35


Success Rate by Binary Length and Error Rate
Number Of Copies = 25, Error Rate ={substitution: p, insertion: 0%, deletion: 0%}
Ours
Naive
Success Rate
Binary Length
Error Rate
0
2
4
6
8



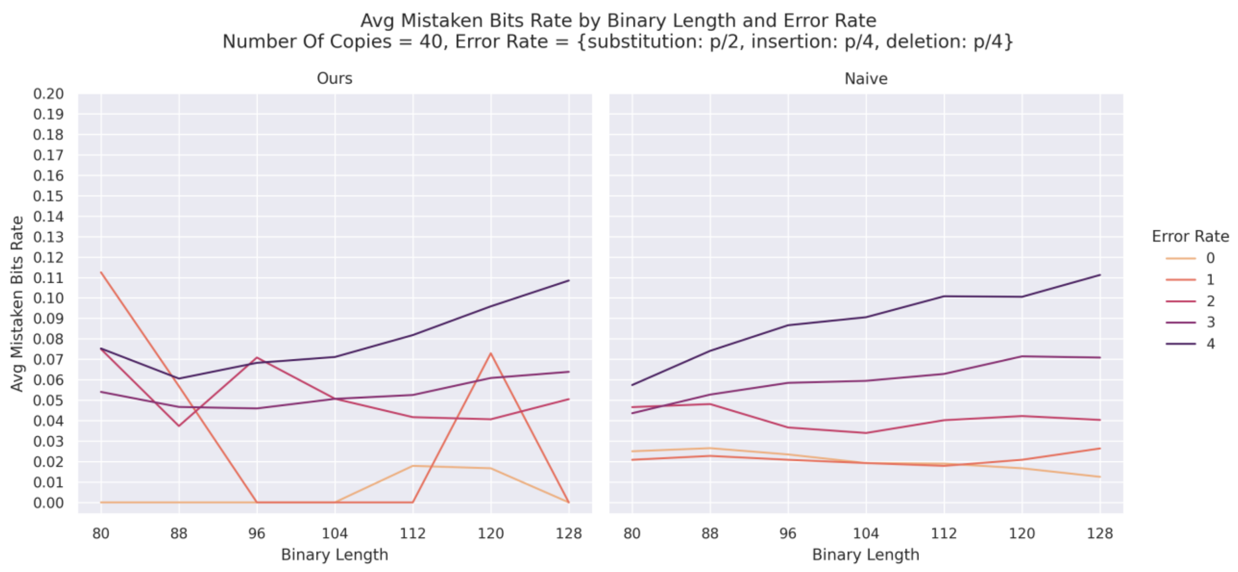
7.2.4. *Success rate by substitution error probability only*

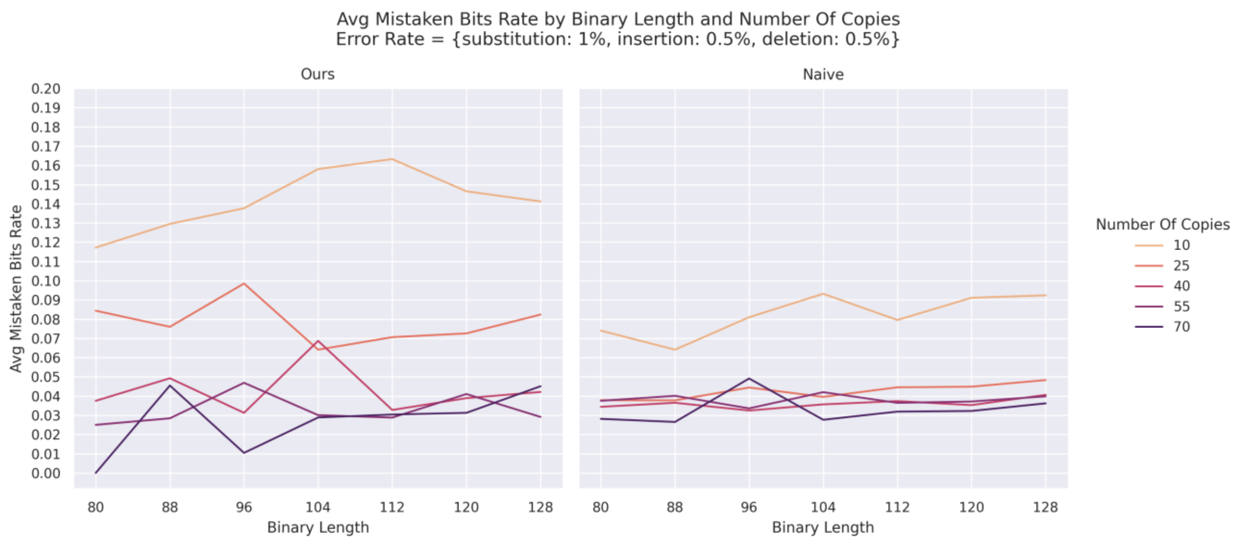
These results present the algorithm's success rate when introduced to substitutions errors only. Notably, our algorithm is much more efficient in this case.



7.2.5. *Average Number of Mistaken bits*

Each decoding mistake corresponds to a different number of erroneous bits, quantified as the Hamming distance- the number of bits that deviate from the original message. The following graphs illustrate the average percentage (rate) of erroneous bits from decoding attempts resulting in mistakes. We believe that the "jumps" in the graphs occur since the average number of erroneous bits varies tremendously between the runs. Nonetheless, we could safely infer that for the majority of the runs the number of mistaken bits does not exceed 10%.





8. Availability

The code used for the generation, encoding, decoding, and testing of this project is available on GitHub: https://github.com/AvivMaayan/DnaProject

9. References

1. *William H. Press, John A. Hawkins, Stephen K. Jones Jr, Ilya J. Finkelstein, HEDGES error-correcting code for DNA storage corrects indels and allows sequence constraints (2020).*
2. *Leon Anavy, Inbal Vaknin, Orna Atar, Roee Amit, Zohar Yakhini, Data storage in DNA with fewer synthesis cycles using composite DNA letters (2019).*
3. *P. E. Hart, N. J. Nilsson, B. Raphael, A formal basis for the heuristic determination of minimum cost paths (1968).*
4. *Additional information of composite DNA storage*