DNA storage report

Shubham Chandak

June 15, 2018

Abstract

Motivation: With the amount of data increasing rapidly, the current storage technologies are unable to keep up due to the slowing down of Moores law. In this context, DNA-based storage can offer significantly higher storage densities (petabytes/gram) and durability (thousands of years). Recent advances in sequencing technologies and DNA writing technologies have led to multiple efforts in this direction. However, the current systems suffer from high cost involved in the reading and writing processes.
Result: In this work, we study the trade-off between the coding density and reading efficiency for DNA storage. We also propose practical coding schemes to achieve close-to-optimal trade-off for a range of read/write error probabilities.
Note: This work was based on motivated by discussions with Prof. Hanlee Ji and Billy Lau. This work was done jointly with Kedar Tatwawadi and Prof. Tsachy Weissman. Part of this work was part of a course project for EE 388 (Modern Coding Theory) instructed by Prof. Andrea Montanari.
Code availability: The code is available online at `https://github.com/shubhamchandak94/dna_storage`.

1 Introduction

In recent years, the amount of data being generated and stored is increasing at exponential rates. With the slowing of Moores law, it is time to look beyond the current storage technologies like magnetic tape and solid-state disks. Interestingly, the cost of DNA sequencing has been decreasing rapidly in the past ten years. DNA is the storage medium of choice for the life on this planet which offers high storage densities (100s of Petabytes per gram [1]) and great durability (1000s of years [2]). The longevity of DNA storage makes it ideal as an archival medium to store the knowledge gained by humanity over the millennia.

There has been a series of work on DNA storage based on DNA synthesis technologies capable of generating 200nt (nucleotide) molecules. The research [1], [3], [4] has focused on various aspects, including error correction and random-access retrieval. Since both the DNA synthesis and sequencing processes are inherently noisy, error correction coding based on the characteristics of this noise is critical for reliable decoding of data. To achieve random access, part of the molecule contains a barcode, which is used to sequence only desired subset of molecules using PCR-based amplification. The coding efficiency is further reduced when storing large files, since we need some redundancy to assemble the file from short synthesized molecules.

Figure 1 shows a schematic of DNA storage system. Binary data is written to short DNA sequences (oligos) of length around 150nt (nucleotide) where each nucleotide can take value in the set {A, C, G, T}. The current writing process (synthesis) generates millions of copies of each oligo [5], possibly with errors. While the writing process can introduce some errors into the oligos, we will not consider that in this work because these errors can be combined with the reading errors leading to a simpler model. We also assume that the binary data is an independent uniformly random stream, which can be obtained by lossless compression of the actual data.

The reading process (sequencing) involves randomly sampling oligos from the pool and reading them possibly with errors (we assume that the read kength matches the oligo length). Since millions of copies of each oligo are present in the pool, we can assume that sampling is performed with replacement. We will ignore the effects of PCR bias and other factors that lead to non-uniform distribution of oligos in the pool. Note that the order in which the reads are sampled is random and is not related to the order in which the oligos were written. For this work, we will assume that the reading process only introduces substitution

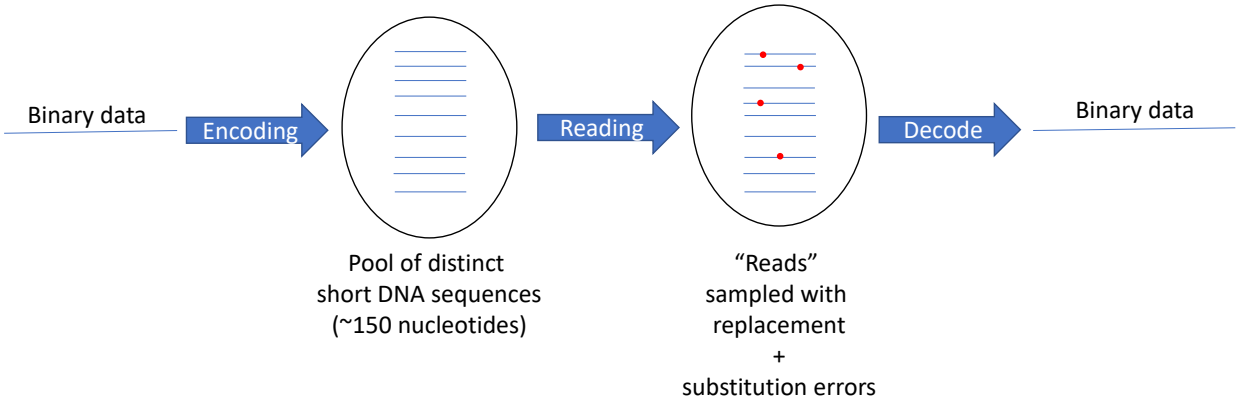


Figure 1: Schematic for DNA storage system.

errors, and in fact we will work with an independent error model which provides a worst-case model for a given average error probability. Designs for worst-case models should be able to perform better with noisier read/write processes which might be necessary to bring down the costs. Oligos with long runs of nucleotides (e.g., AAA) (homopolymers) are difficult to synthesize and sequence and hence we will not allow the encoding scheme to generate sequences with runs of three of more nucleotide.

Given these reads, the aim is to recover the binary data, which provides several challenges. Due to the random sampling process, certain oligos might not read at all (dropout). Even the oligos that are sequenced might contain errors. Furthermore the oligos are permuted during the sampling process. To handle these, we need to use coding schemes to map binary data into oligos. While previous works have proposed several coding schemes, there has been little understanding of the optimal trade-off between the coding density (information bits/base) and the reading efficiency (bases read/information bit). Most works relied on high coverage to counteract the effect of reading errors and applied coding chiefly to detect remaining errors after consensus and for recovering from dropout of oligos. However, using higher coverage for recovering from substitution errors is a highly wasteful strategy just like repetition coding for the binary symmetric channel. Recent work in [6] studied the information-theoretic capacity of DNA storage channel for a simplified model without errors, however the work has limited applicability in the realistic setting.

In Section 2, we study a simplified model where we ignore the constraints posed by the DNA alphabet and work with bits. We also assume that the positions of the reads in the original set of oligos is known to the decoder. This can be achieved by attaching an index to each oligo, which is a nearly optimal strategy for typical data sizes and oligo lengths [6], [7]. In this simplified setting we obtain bounds on the optimal performance and propose practical schemes for approaching these bounds.

In Section 3, we use the techniques developed in Section 2 to design coding schemes handling DNA constraints and indexing of oligos. We discuss the impact of various parameters on the performance of the scheme. Since previous works were optimized for different error models and constraints and we were working with simulated data, we did not perform comparisons with them. We also discuss some practical considerations in applying the proposed schemes and modifications required for real experiments.

2 Simplified Setting

Figure 2 shows the simplified storage model where nL bits are written to $n(1+\alpha)$ binary sequences of length L bits each. We read nc reads of length L which are passed through a binary symmetric channel with error probability ϵ. We assume that the decoder knows the "index" of each read. Here α denotes the number of error correction bits used per information bit and c (coverage) denotes the number of bits read per information bit. Note that the coverage is not defined in terms of the number of oligos, rather it is in terms of the number of information bits. This definition provides better understanding of the cost of reading per information bit. Throughout this section, we assume that $L = 256$.

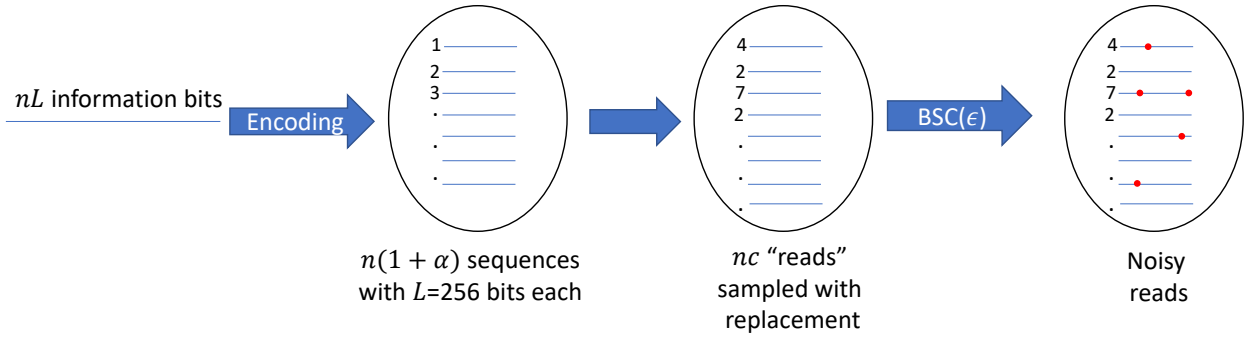


Figure 2: Schematic for simplified storage model.

2.1 Capacity Computation

We first compute the optimal trade-off between c and α when $\epsilon = 0$, i.e., the reads are error-free. In this case, for large enogh n, we can use the Poisson approximation with $\lambda = c/(1+\alpha)$. This gives us an erasure channel with capacity $1 - e^{-\lambda}$. The idea is that we expect to get around $1 - e^{-\lambda}$ fraction of oligos since $e^{-\lambda}$ approximates the probability that any given oligo is read zero times. For reliable recovery, we need that the rate $1/(1+\alpha)$ be less than the capacity. This gives us

$$c \geq (1+\alpha) \log_e \frac{1+\alpha}{\alpha}$$

We see that c decreases monotonically with α, approaching 1 as α increases to ∞.

When $\epsilon \neq 0$, we can obtain lower bounds on the capacity by considering a memoryless approximation to the channel where the input is a bit and the output is a tuple (k, k_0) where k is the number of times that bit is read and k_0 is the number of times the bit is read as 0. The transition probability for this channel is given by

$$P((k,k_0) \mid 0) = \frac{e^{-\lambda}\lambda^k}{k!} \binom{k}{k_0} (1-\epsilon)^{k_0} \epsilon^{k-k_0}$$

which indicates the probability that a Poisson random variable with parameter λ takes value k and that we have $k - k_0$ errors in the k reads of the bit. This is a binary memoryless symmetric channel and hence the capacity achieving distribution is the uniform distribution on the inputs. The capacity can be numerically computed as a function of $\lambda = c/(1+\alpha)$ and ϵ. To obtain the tradeoff between c and α, we can fix α and increase c until the capacity exceeds the rate $1/(1+\alpha)$. We note that this provides us only an achievable result since the channel is not actually memoryless due to the reads being L length strings. However, we can shuffle the bits and hence achieve this capacity even for the actual channel. Even when the errors are not independent, we can again use this shuffling argument to show that we can do at least as well as the memoryless case.

Figure 3 shows the plots of c vs. α for different values of ϵ. The plot was generated using compute_capacity.py. For $\epsilon = 0$, the coverage is around 2.5 to 3 for α around 0.05-0.1. As α gets bigger, the coverage decreases, approaching 1. For $\epsilon > 0$ the curve shifts upward, needing 3-3.5 coverage for α around 0.05-0.1. Here the coverage does not approach 1 as α tends to infinity. This is because even if we get all unique reads, we still need some additional parity bits to correct the errors.

2.2 Strategy I

The first strategy we discuss involves a two stage coding scheme where we first break the data into segments, apply an erasure code to these segments and finally apply error correction to each of these segments. This is shown in Figure 4. This strategy was the one used in [1] where Fountain code [8] was used as the outer erasure code and Reed Solomon code [9] was used as the inner code. While Raptor codes [10] can offer better erasure correction and computational complexity, [1] used Fountain codes for licensing related reasons. Both Fountain and Raptor codes are rateless codes, i.e., they can produce an infinite stream of symbols such that

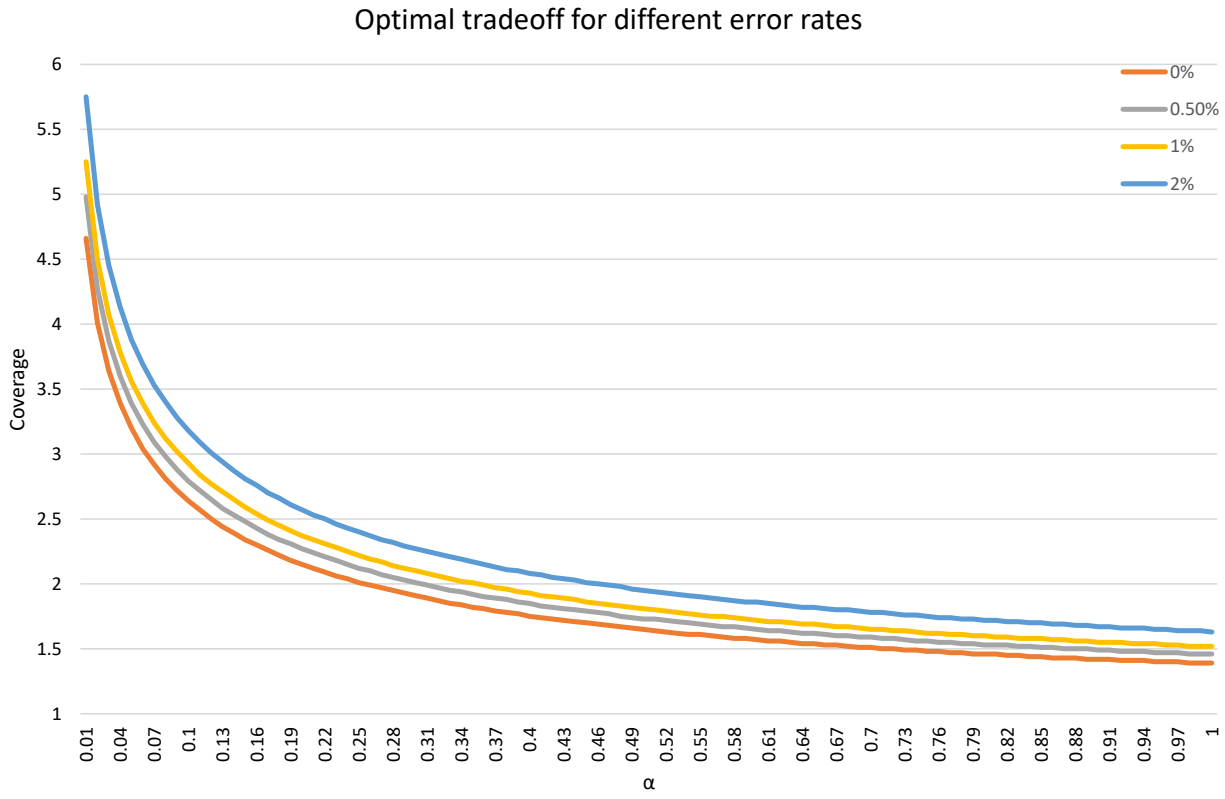


Figure 3: Trade-off between coverage c and α for four values of error probability ϵ.

Figure 4: Schematic diagram for strategy 1.

the decoder can decode the original data with a small number of symbols from the stream. The reason to use Reed Solomon codes might be motivated by the fact that errors in the sequencing can be bursty. In our case where we attempt to design systems for the worst case of independent errors, BCH codes [11] offers better minimum distance guarantees.

We use RaptorQ codes (`https://tools.ietf.org/html/rfc6330`) as the erasure codes. RaptorQ codes consist of Fountain codes along with two outer codes and have several desirable properties such as systematic encoding, low complexity and near-optimal erasure correction. For example, for K input symbols, the decoder is able to decode the input symbols with 99.9999% probability given $K + 2$ encoded symbols. In our case each segment can be thought of as a symbol. We used the Python library available at `https://github.com/mk-fg/python-libraptorq` for the implementation.

We use BCH codes for error correction within each segment. For each k, BCH codes can have block length up to $2^k - 1$ bits and out of this block length rk bits are used for the parity bits. Here r is the number of errors the BCH code can correct. We used $k = 8$ and hence we needed 8 bits for correcting 1 error. We used the Python library available at `https://github.com/jkent/python-bchlib` for the implementation.

During decoding, we first take a consensus of reads with the same index, perform BCH decoding for each segment and then perform RaptorQ decoding. We note that RaptorQ cannot handle errors so it is important that the BCH decoded segments are error free. For this, we presented the segments with fewer corrected errors first to the RaptorQ decoder, since these segments are less likely to be incorrectly decoded. We also tried using the BCH codes as error detection codes (detect up to $2r$ errors) where we threw away any segments with detected errors. However this strategy does not perform well for independent errors since the probability of getting segments will no errors is small. For realistic error models with large fraction of error-free reads, error detection rather than correction might be a better strategy, as used in [1]. For real sequencing data, a few sequences are very erroneous, and it is better to filter them out using qualities or some other indicator such as low coverage.

2.3 Strategy II

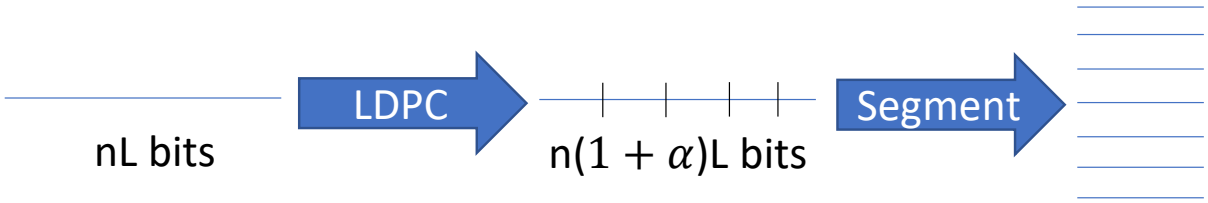


Figure 5: Schematic diagram for strategy 2.

A two-stage coding strategy is suboptimal because the error correction code is applied to a small block length (e.g., $L = 256$) and there is a fundamental limit on the achievable rate for small block lengths [12]. Some previous works such as [4] use an error correction code (Reed Solomon code) at the oligo level. However that is suboptimal because Reed Solomon code considers errors only at the oligo level, not at the base level. To solve this, we proposed another scheme that first applies error correction code on the entire binary data and then divides it into segments (shown in Figure 5). We used LDPC codes [13] as the error correction code since LDPC codes are easily applicable to any binary memoryless symmetric channel and can achieve rates close to the capacity. The channel model we used was the one discussed in Section 2.1, with the channel output being (k, k_0) where k is the number of times the bit was read and k_0 is the number of times the bit was read as a 0. Using this notation, we compute the log-likelihood ratio as follows:

$$LLR(k, k_0) = \log_e \frac{P((k, k_0) \mid 0)}{P((k, k_0) \mid 1)} = (2k - k_0) \log_e \frac{1 - \epsilon}{\epsilon}$$

This LLR can be used in the standard belief propagation decoder for the LDPC codes.

LDPC codes are represented using a bipartite graph with variable nodes (bits in the codeword) and check nodes (representing parity checks), with the check nodes connected to the variable nodes involved in the corresponding parity check equation. This graph is usually constructed randomly according to some distribution on the degrees of the variable and check nodes. Regular LDPC codes have fixed degree variable and check nodes and perform quite well in the high rate regimes for finite block lengths. Irregular LDPC codes can achieve performance closer to capacity but need some fixes to achieve good finite block length performance. Protograph LDPC codes [14] offer the best of both worlds but require some tuning to find the best protograph for the channel at hand. Based on some density evolution analysis using particle filters (particle_filter_*.py), we decided to go ahead with regular LDPC codes which were quite close to the optimal performance. A regular LDPC code is represented by two parameters (d_v, d_c) where d_v is the degree of the variable nodes and d_c is the degree of the check nodes. These quantities are related to α by the formula

$$\alpha = \frac{d_v}{d_c - d_v}$$

For best performance, d_v is set to 3 and d_c is computed according to α. We used the library available at `https://github.com/radfordneal/LDPC-codes` for creating the parity check matrices, generator matrices of the LDPC codes, and also for encoding and decoding. For decoding, we had to add a new mode for our specific channel.

2.4 Results

We conducted experiments with the two strategies to determine their performance. The relevant code is run_raptor_BCH_consensus.py and run_LDPC.py. We set $n = 1000$ which means that the LDPC code dimension was 256000. We should note that even though the LDPC dimension is quite large, the performance is dictated by the fact that the oligos are sampled rather than bits, and hence the effective dimension (for computing finite block length performance using channel dispersion ideas) is smaller.

α	Optimal coverage	Achieved coverage
0.1	2.64	2.89
0.2	2.15	2.35
0.3	1.91	2.06
0.5	1.65	1.75

Table 1: Performance of strategy I for $\epsilon = 0\%$ and $n = 1000$. 100 out of 100 random trials were successful at the achieved coverage.

Table 1 shows the results for strategy I in the noiseless case. No BCH code was applied here. We see that the achieved coverage (coverage at which 100 out of 100 random trials were successful) is quite close to the optimal coverage. Since the RaptorQ code is close to optimal, the gap is explained by the randomness in the sampling due to which we don't get 1000 unique oligos in some trials which leads to failure in decoding.

α	Optimal coverage	Achieved coverage	
		Strategy I	Strategy II
0.1	2.79	6.70	3.25
0.2	2.27	4.30	2.70
0.3	2.01	3.30	2.45
0.5	1.73	2.50	2.10

Table 2: Performance of strategies I and II for $\epsilon = 0.5\%$ and $n = 1000$. 100 out of 100 random trials were successful at the achieved coverage.

Table 2 shows the results for both strategies when error probability ϵ is 0.5%. We see that strategy I does not perform as well, especially at low values of α (which are preferable due to high synthesis costs). This can be explained by inefficiency in the BCH code at small block lengths. At low α, contributing bits

to the BCH code leads to reduction in the amount of Raptor coding. This leads to sharp increase in the coverage needed for recovery as seen in Figure 3. Strategy II does much better in this setting because it uses large block length for coding. The gap between strategy II and optimal performance is partly due to finite block length effects (as also seen in Table 1) and partly due to suboptimality of regular LDPC code for this channel (even asymptotically, the regular LDPC code can't get below roughly 0.25+Optimal coverage as seen using density evolution). We note that the optimal coverage is based on the memoryless assumption in the capacity computation and is an upper bound on the actual optimal coverage. The optimal coverage shown in Table 1 provides a lower bound for the optimal coverage with noise.

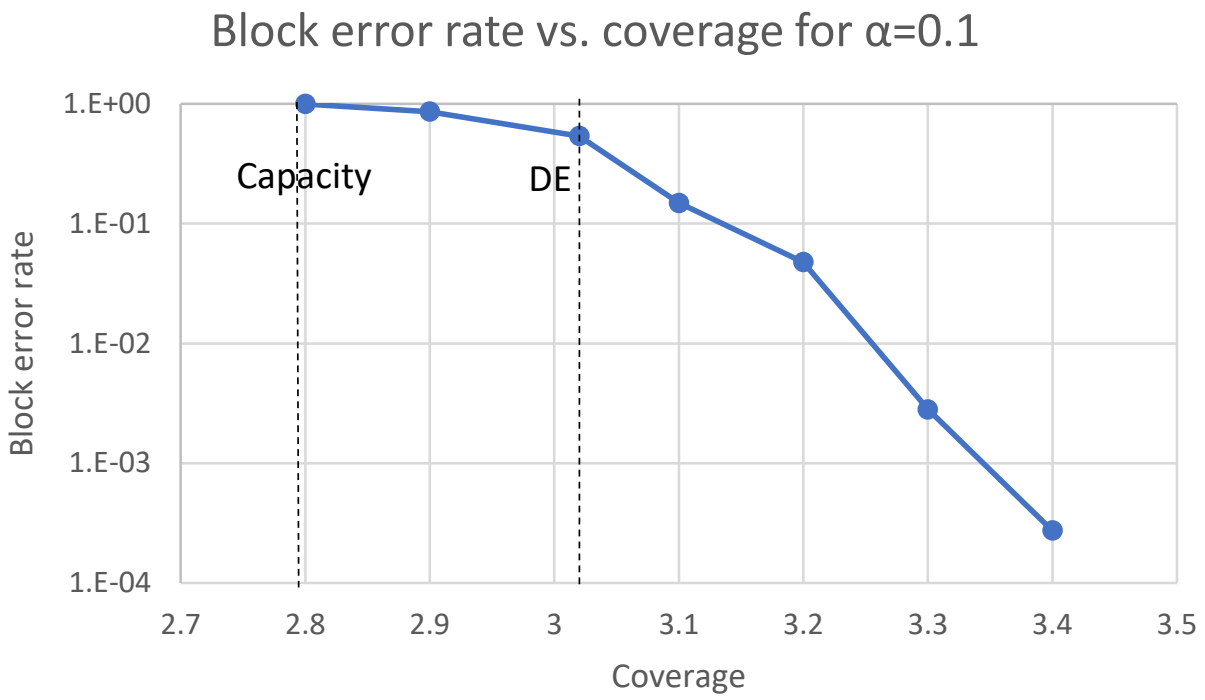


Figure 6: Block error rate for LDPC code with $\alpha = 0.1$ vs. coverage for $\epsilon = 0.5\%$ and $n = 1000$.

To further understand the block error probability of the LDPC codes, we performed larger number of trials for $\alpha = 0.1$, $\epsilon = 0.5\%$ and $n = 1000$. The results are shown in Figure 6. We see that the performance at capacity is pretty bad. At the density evolution threshold (asymptotic performance limit), roughly 50% of the trials were successful. After that the block error rate falls rapidly, getting close to 10^{-4} at a coverage of 3.4.

3 General setting

Based on the results in Section 2, we decided to use LDPC codes for our scheme. However, we cannot directly apply LDPC code to binary data and then convert the binary data to DNA using a 2 bits per base scheme because of the constrains on the DNA sequences. We also need to index the DNA oligos to allow us to find the LLR and decode the LDPC code.

3.1 Constraint coding

Certain oligos are difficult to synthesize and/or sequence using the current technology [5]. For instance, long runs of any base (homopolymers) often lead to deletion errors. Similarly oligos with too large or too small fraction of G and C tend to have more errors and dropouts. Longer repeats within the oligos can also lead to issues. For random input data and sufficiently long oligos, the probability of getting long repeats and too large/small fraction of G and C is negligible, therefore we focus mostly on the homopolymer constraint.

Previous works handled this constraint using different strategies. In [1], the rateless property of Fountain codes was used to generate large number of oligos and throw away (not synthesize) those violating the constraint. They allowed homopolymers of at most 3 bases, which led to more than 80% of oligos being discarded. However this strategy is not scalable to longer oligos because the number of oligos needed until sufficiently many valid oligos are found grows exponentially in the oligo length. Furthermore, if we impose a stricter constraint, the fraction of valid oligos drops sharply. For example, allowing homopolymers of at most 2 bases, the probability that an oligo of length 150 is valid is around 0.0007. Thus, only one in 1500 oligos will be accepted by this strategy. This strategy is more suited for rateless codes like Fountain codes or Raptor codes. While we can use this with LDPC codes by generating random masks and computing the XOR of the mask with the oligo until we obtain a valid oligo (we also need to store the seed for the mask), the overall scheme is not as elegant in the setting.

In [4], even runs of 2 bases were not allowed. To handle this, they first converted the binary data to ternary and then applied a rotating code which guarantees that the next base is different from the current base. For example, if the current base is A, then depending on the value of the ternary symbol, the next base can take any of the three values in {C, G, T}. They first use Reed Solomon codes over the oligos and then use this constraint coding for each oligo. Note that a single error in the ternary representation can lead to large number of errors in the original binary representation. They deal with this by taking consensus of sequences and by the error-resilience of Reed Solomon codes. However, this strategy is far from optimal because of the inefficiency in Reed Solomon codes over the oligos when the actual errors are point errors such as base substitutions.

In [3], a more direct strategy is used to ensure no runs of 4 or more nucleotides in the oligos. Their scheme ensures that one error in the DNA representation leads to at most two bit errors in the original binary representation and also provides some protection against deletions. However, the rate achieved is 1.6 bits per base, which is much less than the optimal rate of around 1.99 bits per base [1] when runs of at most 3 nucleotides are allowed.

In this work, we allow runs of at most 2 nucleotides. The optimal rate for this is 1.92 bits per base [1], which can be approached with the high rate schemes discussed in Section 3.1.2. We use a different strategy from [1] due to the issues mentioned before. However, directly using a high rate scheme after applying error correction does not work. This is because the high rate schemes work with large blocks and one error in DNA can lead to any number of errors in the binary block. Thus, the error correction needs to handle significantly higher error rate leading to much lower rates. Thus, we use the strategy suggested in [15], [16], and shown in Figure 7. The idea is to first apply high rate constraint encoder to the binary data and then use a systematic error correction procedure. The parity bits are encoded using a low rate constraint encoder with the property that one error in DNA leads to small number of errors in the parity bits. The high rate encoder need not satisfy this property because the error protection is applied after the constraint coding. As long as the number of parity bits are reasonably small as compared to the information bits, the overall constraint coding rate achieved is quite good. Figure 8 shows the effective constraint coding rate versus the amount of error correction used when the high rate code achieves 1.92 bits per base and the low rate code achieves 1.67 bits per base. We see that even when the number of parity bits is 50% of the information bits, the constraint coding rate is 1.8 bits per base, which is not too far from 1.92 bits per base.

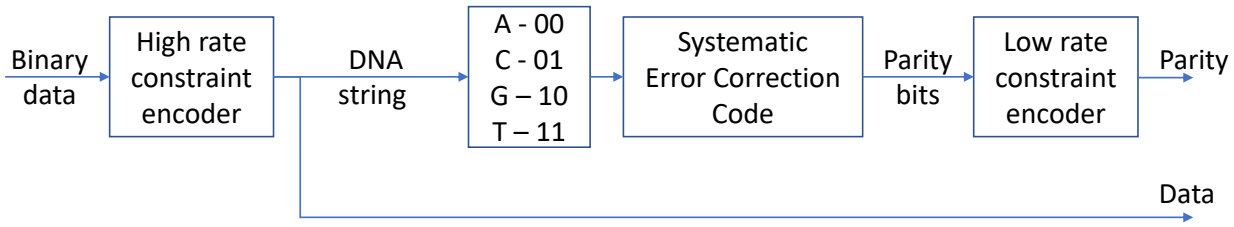


Figure 7: Schematic diagram for constraint coding scheme.

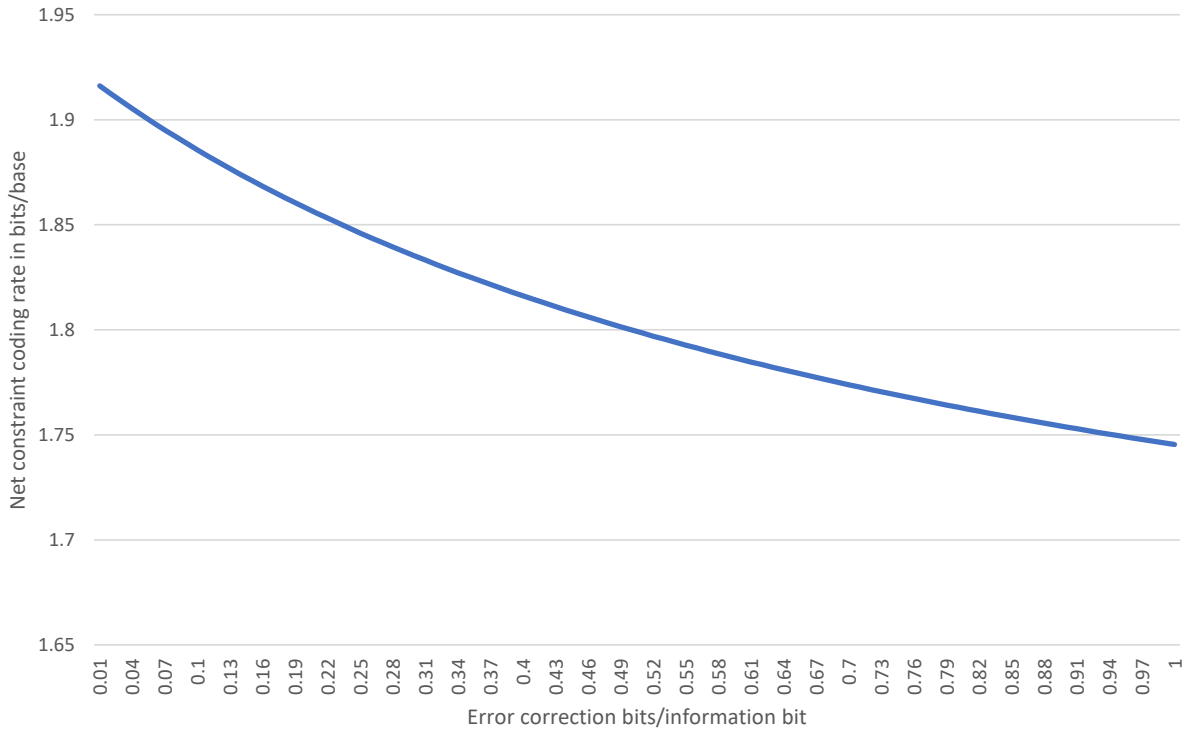


Figure 8: Effective constraint coding rate vs. error correction.

3.1.1 Low rate code

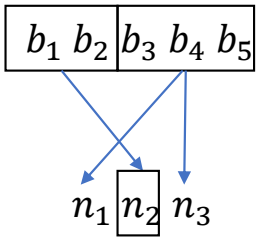
The low rate constraint code maps 5 bits to 3 bases as shown in Figure 9 (partly inspired by the ideas in [3]). The idea to ensure that the middle base is different from the other two bases, so that these blocks of three bases can be concatenated together without producing any run of 3 consecutive bases. To achieve this, the first two bits are mapped directly to the middle base. For the remaining three bits, we use the right table in the figure, which provides two options for the remaining two bases. For any row of the table and any middle base, one of the options gives a valid encoding. Further, the options are all different and hence an error in the middle base does not impact the decoding of the three bits b_3, b_4 and b_5. Finally, the right table is constructed such that a single error in the DNA leads to no more than two errors in b_3, b_4 and b_5. Thus, this scheme achieves a rate of 1.67 bits per base and a single base error affects at most two bits.

3.1.2 High rate code

For high rate constraint coding, we use a scheme suggested in [17] based on a scheme in [18]. We first define a mapping from a binary block to a quaternary block with each symbol in $\{0, 1, 2, 3\}$ such that the quarternary block has no consecutive 0's and does not start with a 0. Then the quarternary blocks will be concatenated and we take the cumulative sum modulo 4. Since each block has no consecutive 0's and does not start with a 0, the cumulative sequence will not have any three consecutive repeated value. Finally the quaternary cumulative sum sequence will be mapped to the DNA alphabet using the obvious one-to-one mapping.

Note that it is equivalent to construct a mapping from a binary block to a quaternary block with each symbol in $\{0, 1, 2, 3\}$ such that the quarternary block has no consecutive 3's and does not start with a 3 because we can always swap 0's and 3's after this step. For this mapping we use a scheme based on Fibonacci-like sequences (appropriately modified from scheme in [18]. First we fix the quaternary block length to r and compute the sequence $(n_0, \ldots, n_r)$ as follows:

- $n_0 = 1$



$b_1\ b_2$	n_2
00	A
01	C
10	G
11	T

$b_3\ b_4\ b_5$	$n_1\ n_3$	
	Option 1	Option 2
000	AA	CC
001	AC	GT
010	AG	CT
011	AT	GC
100	CG	TA
101	TC	GA
110	TG	CA
111	GG	TT

Figure 9: Low rate constraint code achieving 1.67 bits/base. One error in the DNA representation causes at most two errors in the binary representation.

- $n_1 = 3$
- For $i = 2, \ldots, r-1$:

  $\rightarrow\ n_i = 3n_{i-1} + 3n_{i-2}$
- $n_r = 3n_{r-1}$

n_r gives the maximum number of binary strings that can be encoded by this strategy and hence $\lfloor \log_2 n_r \rfloor$ gives the binary block length. To encode x such that $0 \leq x < n_r$, into $(q_{r-1}, \ldots, q_0) \in \{0,1,2,3\}^r$ we proceed as follows:

- For $i = r-1, \ldots, 0$:

  $\rightarrow\ q_i = \lfloor \frac{x}{n_i} \rfloor$

  $\rightarrow\ x = x - q_i n_i$

It can be shown that this leads to a quaternary sequence with the desired properties and such that $x = \sum_{i=0}^{r-1} q_i n_i$.

Setting $r = 173$, we get $\lfloor \log_2 n_r \rfloor = 332$ which achieves a rate of 1.919 bits per base which is very close to the optimal rate.

3.2 Index

The DNA storage problem can be thought of as a problem of coding data into a set of length L DNA oligos. One method for achieving this is to encode the data into a long vector, break the vector into short segments and attach an index to each segment. While this strategy is not optimal in general, it is nearly optimal for reasonably long oligos and for reasonable data sizes [7]. Furthermore, the indexing strategy allows us to utilize the existing error correction codes. Thus, we use an index-based scheme for our system. We use the constraint coding scheme described above for the index and use BCH code for protecting the index

separately. For n oligos, we need an index of length $\lceil \log_2 n \rceil$ bits. To simplify matters, we fix the index to 30 bits, which can handle $2^{30} \approx 10^9$ oligos, which corresponds to around 25 GB data for oligo length $L = 150$ and coding density of 1.4 bits per base. It is relatively straightforward to increase this length in the future. The 30 bits are converted to 16 bases using a high rate constraint coding scheme and then BCH coding with $k = 6$ is applied. For correcting up to r errors, the BCH code uses $6r$ parity bits, which are converted to DNA using the low rate constraint coder. During decoding, we first decode the index and remove oligos where error correction fails or the index exceeds the actual number of oligos.

3.3 Overall system

Here we summarize the overall system.

Encoding:

- Compute the effective oligo length after considering space needed for index and index BCH parity.
- Divide the input data into blocks such that each block length is less than the LDPC code dimension after applying constraint coding.
- Apply high rate constraint coding to the block.
- Apply LDPC code to obtain parity bits.
- Split the constraint coded systematic bits into oligos.
- Split the parity bits into oligos and apply low rate constraint coding.
- Add index and index parity bits to the oligos.

During decoding, we assume that the parameters and the original file size are known to the decoder so that it can compute various quantities required for decoding, for instance assigning the counts to the correct position in the LDPC code.

Decoding:

- Decode the index from oligos, removing oligos where this fails.
- For each index, determine whether that oligo contains systematic or parity bits. For systematic bits, decode according to 2 bits per base scheme. For parity bits, decode according to the low rate constraint code.
- Accumulate counts for each systematic and parity bit and compute log likelihood ratios.
- Perform LDPC decoding for each block.
- Decode the systematic bits obtained after error correction using the high rate constraint code.

3.4 Results and discussion

To test the overall scheme, we performed experiments for various parameter values and error probabilities. We encoded 30,668 bytes to DNA, where this number was chosen so that the systematic bits fit in the 256,000 dimension of the LDPC code giving a single LDPC block. The code used for encoding, decoding and various utility functions are available in util.py. The function find_min_coverage was used to find the minimum coverage at which decoding is successful for 100 out of 100 random trials. The oligo length and the read length were fixed to 150. The error was independent and identically distributed for different bases and the probability of transitioning to any of the three remaining bases was uniform.

Table 3 shows the results for three LDPC codes and four error probability values. We use more BCH protection for index for higher ϵ in order to minimize the coverage required. The density is defined as the number of bases written (number of oligos times the oligo length) per information bit/2 (30,668 times 8/2).

LDPC (d_v, d_c)	α_{LDPC}	Index BCH correction	ϵ	Density (bits/base)	Coverage (bases/(bit/2))
(3,63)	0.05	1	0.5%	1.53	5.2
(3,33)	0.1	1	0.5%	1.45	4.4
(3,9)	0.5	1	0.5%	1.01	3.0
(3,63)	0.05	2	1.0%	1.49	5.2
(3,33)	0.1	2	1.0%	1.41	4.4
(3,9)	0.5	2	1.0%	0.99	3.0
(3,63)	0.05	3	2.0%	1.46	6.0
(3,33)	0.1	3	2.0%	1.38	4.8
(3,9)	0.5	3	2.0%	0.96	3.4
(3,63)	0.05	3	3.0%	1.46	6.4
(3,33)	0.1	3	3.0%	1.38	5.2
(3,9)	0.5	3	3.0%	0.96	3.6

Table 3: Best coverage achieved for three LDPC codes and four error probabilities. α_{LDPC} denotes the number of parity bits introduced by the LDPC code per information bit. BCH index correction denotes the number of bit errors the BCH code on the index can correct. Density is the total number of bases (including effects of error correction, index and constraint coding) used per information bit while writing. Coverage is the minimum number of bases read (number of reads times read length) per 2 information bits, such that 100 out of 100 random trials were successful. Coverage was increased in steps of 0.2 until success was achieved.

The division by 2 means that the coverage of at least 1 is needed for recovery. The definition of coverage is slightly unconventional and aims to measure the cost of reading. Coverage is defined as the number of bases read (number of reads times read length) per information bit (30,668 times 8). Using the (3,63) LDPC code, we can successfully decode with coverage close to 6 with a density close to 1.5 bits per base across the error probabilities from 0.5% to 3%. We can bring the coverage down to 3-4 by using a lower rate LDPC code at density around 1 bit per base. Comparing some of these results with the binary results in Table 2, we observe a gap in the performance. This gap is due the fact that some of bases read are index bases or additional bases due to constraint coding and hence we need to read more bases to get a certain number of bits. Still, the performance is quite good and the codes are quite resilient to changes in ϵ.

LDPC code	α_{LDPC}	Index BCH correction	ϵ	Density (bits/base)	Coverage (bases/(bit/2))
(3,33)	0.1	0	1.0%	1.51	5.6
(3,33)	0.1	1	1.0%	1.45	4.6
(3,33)	0.1	2	1.0%	1.41	4.4
(3,33)	0.1	3	1.0%	1.38	4.6
(3,33)	0.1	0	2.0%	1.51	7.8
(3,33)	0.1	1	2.0%	1.45	5.4
(3,33)	0.1	2	2.0%	1.41	5.0
(3,33)	0.1	3	2.0%	1.38	4.8
(3,33)	0.1	4	2.0%	1.34	5.0

Table 4: Effect of BCH correction of index on the coverage for two different error probabilities.

Table 4 shows the effect of number of BCH protection bits on the coverage. We see that as we use more BCH protection, the required coverage first decreases then increases. The increase is because some of the coverage is wasted on reading these BCH bits without improving the index decoding. The optimum number of BCH bits depend on the error probability. We should note here that since the experiments were performed with relatively small number of oligos (around 1000-1700 depending on the parameters), even if the BCH decoding made an error, the incorrect index was typically greater than the total number of oligos. Such oligos were not used for LDPC decoding. For larger data sizes, there will be higher probability of making an error in the index and the LDPC code will need to handle higher effective error rate. For this, we allow the

user to specify a separate ϵ for calculating the LLR for LDPC decoding. We also provide a mode where we simply throw away segments where the BCH code detects an error. This mode might be useful if the errors are bursty and most index segments are read without error.

For actual experiments, the iid substitution error model is far from true. The proposed codes should do at least as well with non-independent errors, but need not be optimal in that setting. In any case, most components of this work should be applicable. It is possible that the Raptor + BCH strategy might be competitive there. Similarly, error detection rather than correction might be more appropriate if most reads are error-free and only few have lots of errors. Since the error probability is not accurately known, we also need to play around with the LDPC decoding parameters. According to [4] and [5], the sequencing errors are mostly substitutions (for Illumina), however the synthesis errors consist of deletion errors as well. Since more than 50% of reads had no deletions, we can still use the proposed code by removing the reads with smaller length than expected. A better approach could be to apply a deletion code either to each oligo or on larger blocks. This will be part of future work.

4 Conclusions

We studied the DNA storage problem to understand the trade-off between the amount of DNA written and read for reliable recovery. For a simplified binary model, we analyzed the trade-off information-theoretically and proposed schemes to approach the optimum. For the general setting, we introduced schemes for constraint coding to build an end-to-end DNA storage system. We experimented with various parameters and showed that the proposed code performs well for a range of error probabilities.

References

[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] G. R. N., H. Reinhard, P. Michela, P. Daniela, and S. W. J., "Robust chemical preservation of digital information on dna in silica with errorcorrecting codes," *Angewandte Chemie International Edition*, vol. 54, no. 8, pp. 2552–2555.

[3] M. Blawat, K. Gaedke, I. Htter, X.-M. Chen, B. Turczyk, S. Inverso, B. W. Pruitt, and G. M. Church, "Forward error correction for dna data storage," *Procedia Computer Science*, vol. 80, pp. 1011 – 1022, 2016. International Conference on Computational Science 2016, ICCS 2016, 6-8 June 2016, San Diego, California, USA.

[4] L. Organick, S. D. Ang, Y.-J. Chen, R. Lopez, S. Yekhanin, K. Makarychev, M. Z. Racz, G. Kamath, P. Gopalan, B. Nguyen, *et al.*, "Random access in large-scale dna data storage," *Nature biotechnology*, vol. 36, no. 3, p. 242, 2018.

[5] R. Heckel, G. Mikutis, and R. N. Grass, "A characterization of the DNA data storage channel," *CoRR*, vol. abs/1803.03322, 2018.

[6] R. Heckel, I. Shomorony, K. Ramchandran, and D. N. C. Tse, "Fundamental limits of DNA storage systems," *CoRR*, vol. abs/1705.04732, 2017.

[7] A. Lenz, P. H. Siegel, A. Wachter-Zeh, and E. Yaakobi, "Coding over sets for DNA storage," *CoRR*, vol. abs/1801.04882, 2018.

[8] D. J. C. MacKay, "Fountain codes," *IEE Proceedings - Communications*, vol. 152, pp. 1062–1068, Dec 2005.

[9] I. Reed and G. Solomon, "Polynomial codes over certain finite fields," *Journal of the Society for Industrial and Applied Mathematics*, vol. 8, no. 2, pp. 300–304, 1960.

[10] A. Shokrollahi, "Raptor codes," *IEEE Transactions on Information Theory*, vol. 52, pp. 2551–2567, June 2006.

[11] R. Bose and D. Ray-Chaudhuri, "On a class of error correcting binary group codes," *Information and Control*, vol. 3, no. 1, pp. 68 – 79, 1960.

[12] Y. Polyanskiy, H. V. Poor, and S. Verdú, "Channel coding rate in the finite blocklength regime," *IEEE Transactions on Information Theory*, vol. 56, no. 5, pp. 2307–2359, 2010.

[13] D. J. C. MacKay and R. M. Neal, "Near shannon limit performance of low density parity check codes," *Electronics Letters*, vol. 32, pp. 1645–, Aug 1996.

[14] Y. Fang, G. Bi, Y. L. Guan, and F. C. Lau, "A survey on protograph ldpc codes and their applications," *IEEE Communications Surveys & Tutorials*, vol. 17, no. 4, pp. 1989–2016, 2015.

[15] W. Bliss, "Circuitry for performing error correction calculations on baseband encoded data to eliminate error propagation," *IBM Tech. Discl. Bul.*, vol. 23, pp. 4633–4634, 1981.

[16] K. A. S. Immink, "A practical method for approaching the channel capacity of constrained channels," *IEEE Transactions on Information Theory*, vol. 43, no. 5, pp. 1389–1399, 1997.

[17] K. A. S. Immink and K. Cai, "Design of capacity-approaching constrained codes for dna-based storage systems," *IEEE Communications Letters*, vol. 22, pp. 224–227, Feb 2018.

[18] W. Kautz, "Fibonacci codes for synchronization control," *IEEE Transactions on Information Theory*, vol. 11, pp. 284–292, April 1965.