HEDGES error-correcting code for DNA storage corrects indels and allows sequence constraints

William H. Press1,a,1, John A. Hawkins1,b,d,e, Stephen K. Jones Jr1,d,e, Jeffrey M. Schaub1,d,e, and Ilya J. Finkelstein1,d,e

*Department of Computer Science, The University of Texas at Austin, Austin, TX 78712; 1Oden Institute of Computational Engineering and Sciences, The University of Texas at Austin, Austin, TX 78712; 1Department of Molecular Biosciences, The University of Texas at Austin, Austin, TX 78712; and 1Institute for Cellular and Molecular Biology, The University of Texas at Austin, Austin, TX 78712

Contributed by William H. Press, June 6, 2020 (sent for review March 16, 2020; reviewed by Joshua B. Plotkin and H. Vincent Poor)

Synthetic DNA is rapidly emerging as a durable, high-density information storage platform. A major challenge for DNA-based information encoding strategies is the high rate of errors that arise during DNA synthesis and sequencing. Here, we describe the HEDGES (Hash Encoded, Decoded by Greedy Exhaustive Search) error-correcting code that repairs all three basic types of DNA errors: insertions, deletions, and substitutions. HEDGES also converts unresolved or compound errors into substitutions, restoring synchronization for correction via a standard Reed–Solomon outer code that is interleaved across strands. Moreover, HEDGES can incorporate a broad class of user-defined sequence constraints, such as avoiding excess repeats, or too high or too low windowed guanine–cytosine (GC) content. We test our code both via in silico simulations and with synthesized DNA. From its measured performance, we develop a statistical model applicable to much larger datasets. Predicted performance indicates the possibility of error-free recovery of petabyte- and exabyte-scale data from DNA degraded with as much as 10% errors. As the cost of DNA synthesis and sequencing continues to drop, we anticipate that HEDGES will find applications in large-scale error-free information encoding.

DNA | information storage | error-correcting code | indel | Reed–Solomon

Data deposition: The sequenced reads used in testing are available on the Sequence Read Archive (SRA) under accession nos. SAMN14897329–SAMN14897335 (SRA Project PRJNA631963). The computer code used for the generation and testing of the inner HEDGES code and outer RS code is available on GitHub, https://github.com/wihpress/hedges.


Reviewers: J.B.P., University of Pennsylvania; and H.V.P., Princeton University.

This open access article is distributed under Creative Commons Attribution-NonCommercial-NoDerivatives License 4.0 (CC BY-NC-ND).

This paper constructs an error-correcting code for the \{A, C, G, T\} alphabet of DNA. By contrast with previous work, the code corrects insertions and deletions directly, in a single strand of DNA, without the need for multiple alignment of strands. This code, when coupled to a standard outer code, can achieve error-free storage of petabyte-scale data even when \sim 10% of all nucleotides are erroneous.
Fig. 1.  (A) Distribution of insertion and deletion errors (indels) in a typical DNA storage pipeline (Table 1); ins, insertion; del, deletion; sub, substitution.  
(B) (Left) Existing DNA-based encoding methods require sequence-level redundancy, strand alignment, and consensus calling to reduce indel errors.  
(Right) HEDGES corrects indel and substitution errors from a single read.  
(C) Overview of the interleaved encoding pipeline used throughout this paper.  
(D) HEDGES encoding algorithm in the simplest case: half-rate code, no sequence constraints. The HEDGES encoding algorithm is a variant of plaintext auto-key, but with redundancy introduced because (in the case of a half-rate code, for example) 1 bit of input generates 2 bits of output. Hashing each bit value with its strand ID, bit index, and a few previous bits “poisons” bad decoding hypotheses, allowing for correction of indels.  
(E) An example HEDGES encode, encoding bit 9 of the shown data strand (red box). As in D, half-rate code, no sequence constraints.  
(F) The HEDGES decoding algorithm is a greedy search on an expanding tree of hypotheses. Each hypothesis simultaneously guesses one or more message bits $v_i$, its bit position index $i$, and its corresponding DNA character position index $k$. A “greediness parameter” $P_{ok}$ (see SI Appendix, Supplementary Text) limits exponential tree growth: Most spawned nodes are never revisited.  
(G) Illustration of a simplified HEDGES decode. The example bit strand message is encoded and then sequenced with an insertion error. Blue squares give decoding action order: 1, Initialize Start node; 2 to 5, explore best hypothesis at each step; and 6, traceback and output the best hypothesis message. 
DNA image credit: freepik.com.
to the experimental constraints on DNA synthesis, for example, balanced GC content and the avoidance of homopolymer runs. 4) It has, effectively, zero strand ordering errors, removing a source of large bursts of errors. Although this paper’s main contribution is an efficient indel-correcting code, we also develop a specific implementation of the outer Reed–Solomon (RS) code for DNA-based storage. The RS code is applied “diagonally” across multiple DNA strands (Fig. 1C) to more evenly distribute synthesis and sequencing errors, which improves error correction performance (15). We test our strategy (both in silico and in vitro) with degraded DNA oligonucleotide pools. Based on these experiments, we use computer simulations to demonstrate that this coding strategy enables error-free exabyte \((10^{18})\)-scale DNA storage.

**Results**

**HEDGES Theoretical Design.** Fig. 1 shows the data flow for HEDGES encoding and decoding algorithms. In the terminology of coding theory, HEDGES is an infinite-constraint-length convolutional code (a “tree code”) incorporating some features specific to the DNA channel. It is decoded via a stack algorithm that assigns costs to both indels and substitutions. The decoding algorithm succeeds probabilistically, with the ability to signal success or failure. Decoding failures are then regarded as erasures (unknown bits or bytes) and can be corrected in the outer code [i.e., an RS\((255,223)\) code]. Alternatively, the error strand can be discarded and resequenced.

The simplest case is a half-rate code (1 bit encoded per nucleotide) with no constraints on the output DNA sequence, shown in Fig. 1D (see SI Appendix, Fig. S1 for full diagram). The basic plan is a variant of a centuries-old “text auto-key encoding” cryptographic technique (16) (see also Wikipedia, “Autokey cipher”). We generate a stream of pseudorandom characters \(K_i \in \{0, 1, 2, 3\} \equiv \{A, C, G, T\}\), where each \(K_i\) depends deterministically, via a hash function, on a fixed number of previous message bits \(b_{j-i}\), on the current bit position index \(i\) of the current message bit \(b_i\), and on the strand ID. We then emit a character \(C_i\) in \(\{A, C, G, T\}\) with \(C_i \leftarrow K_i + b_i\), the addition performed modulo 4. Redundancy for error correction occurs because, at each \(i\), \(C_i\) can take on only two out of four values, because \(b_i \in \{0, 1\}\). The output DNA is always completely (pseudo)random because the hash is pseudorandom. Thus, the DNA is, in this sense, independent of the encoded message. We note that nothing limits this scheme to the four “natural” nucleotides; generalization to an increased number is straightforward.

The decoding algorithm sequentially guesses message bits (Fig. 1F). Each guessed bit \(b_i\)—along with its guessed position \(i\)—allows the algorithm to predict (via the forward encoding algorithm) a nucleotide value \(C_i\). If \(C_i\) agrees with the observed value, a reward is assigned to that guess. If \(C_i\) disagrees, the guess is penalized as appropriate for a substitution. If it disagrees, but would agree at a different assumed position, plus or minus one, that distinct guess is assigned a penalty appropriate for an insertion or deletion. A heap structure keeps systematic track of all currently viable chains of guesses. To prevent the heap from growing exponentially, only chains of guesses that have close to the best total score are extended preferentially, using a variant of the A* algorithm (17).

In summary, the algorithm encodes information as a stream of nucleotides such that any single decoding error in either nucleotide identity or nucleotide position will “poison” the downstream predictions. Thus, on decoding, there will be only one good-scoring chain of guesses—the correct one. In the unlikely case that the heap grows larger than a preset size \(H_{\text{limit}}\) (e.g., \(10^6\)), we declare a decode failure and mark the remaining part of the strand as an erasure. Similarly, by including the strand ID in the hash function input, any strand with incorrect decoded strand ID will be “poisoned” for the full length of the message and fail to decode. This results in effectively zero strand ordering errors, the most expensive type of simple error in interleaved encoding schemes.

**Testing In Silico.** For in silico testing, we assumed equal rates for substitutions, insertions, and deletions with total error probability per nucleotide \(P_{\text{err}} = P_{\text{sub}} + P_{\text{del}} + P_{\text{ins}}\). Any other distribution can then be conservatively bounded by the choice \(P_{\text{err}} = 3 \times \max(P_{\text{sub}}, P_{\text{ins}}, P_{\text{del}})\). For simplicity, we assumed that errors occurred at random positions in the DNA strand. The experimental tests in vitro had no such assumption (see below).

The HEDGES algorithm was implemented in C++ for speed, with a Python-callable interface for encoding/decoding single strands. As an initial validation of programming accuracy and interface design, we used an outer-code concatenated design with packets of 255 strands of length 300 protected by RS\((255,223)\). A detailed description of the encoding design is provided in SI Appendix, Supplementary Text; all corresponding computer programs are provided in SI Appendix and available via GitHub. For every code rate \(r\) in \([0.166, 0.250, 0.333, 0.500, 0.667, 0.750, 0.833, 0.917]\) and \(r = 0.750, 0.833\), we encoded 10 packets \((\sim 10^7\) nucleotides), pooled all of the strands, and duplicated them as if sequencing to depth 5, meaning that each strand was duplicated a Poisson random number of times, with mean 5. We verified that the pooled and duplicated strands could be decoded, the packets recovered and ordered, RS applied, and the messages recovered without error—but only up to some maximum tolerable error rate \(P_{\text{err}}\) that increased with decreasing code rate. That is, we “tested to failure” on \(P_{\text{err}}\). For this test, the maximum heap size was set to an intentionally small value, \(H_{\text{limit}}\) = \(10^6\), to increase the number of decode failures and thus stress the program. The results of this test were as expected and gave us confidence to proceed with in vitro and larger-scale in silico testing (below).

We next needed to construct a statistical error model that could be extrapolated to the petabyte or exabyte scale. For this model, we needed to know the rate of bit errors and byte errors in HEDGES output (for each code rate \(r\) as a function of \(P_{\text{err}}\)). Because decode errors tended to occur in bursts, the rate of byte errors was less than the \(8x\) expected for independent bit errors. We also needed to know the probability per strand of a decode failure, and the distribution of such failures along the strands. For each pair \(\langle r, P_{\text{err}} \rangle\), we simulated 12,000 strands of length 10,000 (about \(10^8\) nucleotides), now with \(H_{\text{limit}}\) = \(10^6\). Decode failures occurred approximately uniformly along the strands, consistent with the hypothesis that the heap decode “loses its way” only on rare, local, random sequences (SI Appendix, Fig. S2). Decode failures are thus characterized by a single value, the mean run length to failure in a Poisson model. Fig. 24 shows the byte error rate as a function of code rate, while SI Appendix, Fig. S2 gives full details on observed bit and byte error rates, and mean runs to decode failures. Byte error rates were typically 3 to 5 (not 8) times the bit error rates.

Using these byte error rates, we then modeled HEDGES in an overall concatenated ECC design. Fig. 28 shows the average number of message bytes that could be decoded before encountering an uncorrectable error using the concatenated design previously described (see Methods for details). A broad set of code rates (green) are suitable for gigabyte- to exabyte-scale DNA storage. HEDGES decode failures in this region occur every \(10^4\) to \(10^5\) nucleotides but get corrected by the RS outer code or, optionally, by using another exemplar of the strand (see Discussion). Similar simulations with the imposed output constraints of no homopolymer runs (e.g., GGGG or AAAA) greater than four, and \(4 \leq \text{CG} \leq 8\) in any window of 12 nucleotides, are shown in SI Appendix, Fig. S3, and are not substantially different from Fig. 2. We also modeled the effects of sequencing.
constraints more generally (SI Appendix, Supplementary Text and Fig. S4), and the combined model and simulation results indicate that the most common sequencing constraints have minimal impact on HEDGES. In sum, in silico simulations indicate that HEDGES is capable of error-free decoding of exabyte-scale messages.

**Testing In Vitro.** We next tested real-world ECC performance on a pooled sample of 5,865 synthetic 300-base pair DNA strands that were exposed to accelerated aging or enzymatic mutagenesis. Of these, 18 packets of 255 strands were HEDGES inner-encoded (with subsets at each of six code rates) and then RS(255,223) outer-encoded across strands. Five packets, totaling 1,275 strands, were encoded with an unrelated error correction algorithm (18), but also served as a negative control on identifying and sequencing HEDGES strands into packets. Each HEDGES strand consisted of 3’ and 5’ primers of length 23 nucleotides (see Methods) flanking a 254-nucleotide DNA payload. When decoded into bytes, each payload comprised a 1-byte packet number, a 1-byte sequence number (these “salt-protected” on encryption; see Methods), a message payload whose length depended on the code rate, and a 2-byte runout. The sample was PCR amplified and prepared for Illumina-based sequencing. Additionally, we degraded the DNA separately via error-prone PCR mutagenesis or by incubation at high temperature (see Methods). Sequencing was done to a mean depth of ~50.

We performed two kinds of tests of the decoding algorithm, with and without knowledge of the encoded message. “Type A” tests assumed knowledge of the 5,865 strand sequences and could be used to characterize the nature of end-to-end DNA error rates. “Type B” tests were blind decodings of the sequenced data, with knowledge only that the pooled DNA contained HEDGES-encoded data in the specified format.

In our Type A tests, 10 to 15% of sequenced strands could not be uniquely identified with any known input strand, even using quite robust N-gram methods, and even for unmutagenized aliquots. This may be the result of the low concentration, or of contamination at some stage; but it also added to the challenge for the blind type B tests.

For strands whose progenitor sequence could be identified, Table 1 shows measured rates of substitution, insertion, and deletion errors. Notably, only the highest mutagenesis kit protocol produced a substantial increase in DNA errors. Data in ref. 3 estimate DNA degradation over a wide range of timescales and temperatures, suggesting that 50 °C incubation for 8 h should have produced significant mutagenesis. We did not find this, however. So, for further analysis here, we consider only the untreated and high-mutagenesis datasets.

Table 2 shows the results for decoding strands that were identified as belonging to packets of each code rate. Approximately 3% of the strands failed to decode even at small code rates where, in simulation, there were many fewer such failures. Identification of these strands was ambiguous and may stem from PCR mispriming, oligonucleotide misdimerization, and other next generation sequencing (NGS) library preparation artifacts that can vary from batch to batch. Indeed, for lower code rates (where the ECC was relatively unstressed), strand decode failure rates were slightly higher for the untreated case than for the high-mutagenesis case, presumably due to batch-to-batch variation in the number of such artifacts.

For this reason, the values for the mean run to an uncorrectable error in Table 2 are calculated assuming a strategy of rejecting failed decodes, rather than counting them as erasures. We adopted just this rejection strategy in our blind (type B) decoding; the input data were several $10^5$ total reads of the synthesized 5,865 strands (of which 4,590 were message bearing) plus contamination. We shuffled the reads into random order, then attempted HEDGES decoding one read at a time, populating the 18 expected packets of 255 strands with successful decodes and attempting the outer RS error correction when

![Fig. 2. In silico performance of the HEDGES algorithm. (A) The in silico byte error rate for the HEDGES algorithm as a function of code rate, r, shown for a range of simulated DNA error rates $P_{err}$. (B) The mean number of bytes to an uncorrectable error, assuming the interleaved RS(255,223) design discussed in the text.]
the number of erasures (missing strands) was small enough (see Methods for details).

As expected based on the results of Table 2, we achieved error-free decodes of all packets, except in the case of two packets with high mutagenesis at the highest code rate 0.750. With no mutagenesis, 24,000 total reads were required for all 18 packets. With high mutagenesis, 22,000 reads were required for 16 packets, while the undecodable two continued to fail indefinitely. In the successful cases, the number of reads corresponded to about depth 3 on message-bearing message strands. This depth was required merely to sufficiently populate the packets for the outer code to operate due to random strand sampling, not because of any property of HEDGES as the inner code.

**Discussion**

HEDGES is designed to be flexible with respect to DNA strand lengths, DNA sequencing and synthesis technologies, choices of outer code, and interleaving details. The most important feature of HEDGES is that it always either 1) recovers “perfect” synchronization of the individual DNA strand to which it is applied (that is, completely eliminates insertion and deletion errors) or else 2) signals that it is unable to do so by a decode failure. Here, “perfect” means that our reported bit and byte error rates, which are small enough to be completely corrected by a standard outer code such as RS, are already inclusive of any residual instances of misynchronization.

In the feasible (green) regions of Fig. 2, HEDGES decode failures occur about every $10^4$ to $10^7$ nucleotides (bottom cells). Two strategies are possible: 1) We can keep these strands and mark as erasures the bits after the failure point, or 2) we can, instead, use another strand from the pool showing the same strand ID—thus increasing the sequencing depth requirement by a tiny amount. The performance values shown in Fig. 2 use strategy 1; those in Table 2 use strategy 2. Importantly, HEDGES allows constraints on the encoded DNA strands such as reducing homopolymer runs and maintaining a balanced GC content. SI Appendix, Fig. S3, when compared to Fig. 2, shows that such constraints impose little penalty on both the code rate and error correction level. Thus, we demonstrate that both are viable strategies for error correction.

We performed both in silico and in vitro experiments to validate HEDGES across a variety of error rates. Such statistical analyses of rare events, based on both experimental data and simulations, should be a required part of all future proposals for DNA data storage. HEDGES performance on real DNA with observed total errors of ∼1% and ∼3% (Tables 1 and 2) was comparable to computer simulation at the same total DNA error rates and to the statistical model we built using simple Poisson random errors (Fig. 2). In both cases, HEDGES demonstrates the feasibility of large-scale error-free recovery at code rates up to 0.6 (1.2 bits per nucleotide) for ∼1% DNA errors; and 0.5 (1 bit per nucleotide) for ∼3% DNA errors. Error-free exabyte-scale storage is feasible at DNA error rates as large as 7 to 10% with a code rate of 0.25 (0.5 bits per nucleotide). Thus, HEDGES paves the way for robust error correction in large-scale but error-prone pooled synthesis of large DNA libraries.

**Methods**

**HEDGES Encoding in the Half-Rate Case.**

Given a message stream of bits

$$b_i, \ i = 0, 1, 2, \ldots, M, \ b_i \in \{0, 1\}$$

(“the message” or “bits”), we want to emit a stream of DNA characters

$$c_i, \ i = 0, 1, 2, \ldots, N, \ c_i \in \{A, C, G, T\} \equiv \{0, 1, 2, 3\}$$

(“the codestream” or “characters”). We first review the case of a half-rate code, where we emit exactly one $c_i$ (2 bits of output) for each $b_i$ (1 bit of input). Then we will generalize to codes at other rates $r$ (message bits per codestream bit), $0 < r < 1$, so that the streams $b_i$ and $c_i$ are not then in lockstep, and $M \neq N$. One should think of $N$ as being on the order of $10^2$ to $10^4$, the maximum length of a single DNA strand that can be cheaply synthesized today or in the foreseeable future. We want to be able to decode, without residual errors, a received codestream $C'$ that differs from $C$ by substitutions (errors), insertions, and deletions (collectively “indels”). Indels are silent: Their positions in the codestream $C'$ are not known to the receiver.

We generate a keystream of characters $k_i \in \{0, 1, 2, 3\}$, where each $k_i$ depends pseudorandomly (but deterministically by a hash function) on some number of previous message bits $b_j$ (with $j < i$), and also directly on the bit position index $i$ and the strand ID,

$$k_i = f(S_i, l_i, B_i),$$

where $S_i$ is a bits of salt (strand ID), $l_i$ is the lower $q$ bits of the bit index, $B_i$ is the previous $p$ bits, and $f$ is a chosen hash function. (See SI Appendix, Supplementary Text for initialization.) We then emit a codestream character

$$c_i = k_i + b_i,$$

the addition performed modulo 4.

The redundancy necessary for error correction comes from the fact that $b_i$ takes on only two values, while $k_i$ and $c_i$ can have four values. This generates (only) 1 bit of redundancy per character, that is, can be acausally valid by chance half of the time. However, the dependence of $k_i$ on many previous message bits ties any given message bit to many future bits of redundancy. Similarly, the dependence of $k_i$ on $i$ ties every bit to its position index, so that insertions can be identified and removed, and deleted values can be restored.

**Table 2. Measured in vitro performance and inferred extrapolation to large datasets**

<table>
<thead>
<tr>
<th>Code rate</th>
<th>0.166</th>
<th>0.250</th>
<th>0.333</th>
<th>0.500</th>
<th>0.600</th>
<th>0.750</th>
</tr>
</thead>
<tbody>
<tr>
<td>Strand decode failure rate</td>
<td>0.033</td>
<td>0.033</td>
<td>0.040</td>
<td>0.045</td>
<td>0.055</td>
<td>0.069</td>
</tr>
<tr>
<td>Observed byte error rate</td>
<td>0.00061</td>
<td>0.00110</td>
<td>0.00182</td>
<td>0.00240</td>
<td>0.00248</td>
<td>0.00547</td>
</tr>
<tr>
<td>Mean byte errors per RS decode</td>
<td>0.16</td>
<td>0.28</td>
<td>0.46</td>
<td>0.61</td>
<td>0.63</td>
<td>1.39</td>
</tr>
<tr>
<td>Mean bytes to uncorrectable</td>
<td>1.8E+28</td>
<td>9.2E+23</td>
<td>2.1E+20</td>
<td>2.2E+18</td>
<td>1.3E+18</td>
<td>3.8E+12</td>
</tr>
</tbody>
</table>

The upper two values in each box are as experimentally measured in vitro. The bottom values are inferred from the measured quantities for error-free decoding of large datasets under the same experimental conditions. Colors indicate feasibility for large data storage, by the same criteria as Fig. 2.
Some further details about the encoding algorithm are given in SI Appendix, Supplementary Text.

**HEDGES Decoding Algorithm.** For simplicity, assume that error rates are "small," so that "most" DNA bases are received as they were intended. (We saw, in Results, that DNA character error rates up to ~5 to 10% are tolerable.) Suppose we have correctly decoded and synchronized the message through bit \( b_i \) and now want to know bit \( b_{i+1} \). Guessing the two possibilities, \( \{0, 1\} \), we use Eq. 3 to predict two possibilities for the character \( C_i \). In the absence of an error, only one of these is guaranteed to agree with the observed character \( C_i' \). Assign, to a guess that generates disagreement with \( C_i' \), a penalty score equal (conceptually) to the negative log probability of observing a substitution error. In other words, a wrong guess might actually be right, but only if a substitution has occurred. If neither guess produces the correct \( C_i \), then both are assigned the substitution penalty.

We have not yet accounted for the possibility of insertions and deletions. In fact, there are more than the above two possible guesses. We must guess not just \( b_i \in \{0, 1\} \), but also a "skew" \( \Delta \in \{-\ldots, -1, 0, 1, \ldots\} \) that tells us whether, in comparing \( C_i \) to \( C_i' \), we should skip characters (\( \Delta > 0 \)) because of insertions, or post missing characters (\( \Delta < 0 \)) because of deletions (in which case, there is no comparison to be done). As a practical simplification, we consider only \( \Delta \in \{-1, 0, 1\} \), requiring multiple indels to resolve as concatenated single indels. Then there are six guesses for \( (b_i, \Delta) \in \{0, 1\} \times \{-1, 0, 1\} \). Each can be scored by an appropriate log probability penalty for any implied substitution, insertion, or deletion.

Log probability penalties accumulate additively along any chain of guesses. In the causal case of a chain of all-correct guesses, we accumulate penalties only in the (relatively rare) case of actual errors. However, because of the way that the key \( K_i \) (Eq. 3) is constructed, a single wrong guess for either \( K_i \) or \( \Delta \) throws us into the acasual case where \( 3/4 \) of subsequent comparisons of computed \( C_i \) (at some bit position \( i \)) to observed \( C_i' \) (at some index \( k \)) will not agree—thus penalties will accumulate rapidly. The decoding problem, conceptually a maximum likelihood search, thus reduces to a shortest-path search in a tree with branching factor 6, but with the saving grace that the correct path will be much shorter than any deviation from it.

The rate of decode errors rises in the last several bytes of message, because some incorrect chains don't have time to accumulate bad scores. To counter this, we pad each strand with (typically) two "runout bytes" of message zeroes at encode, and ignore them at decode. The need for runout bytes makes the HEDGES algorithm inefficient (and thus unsuitable) for an application needing very short DNA strands (e.g., tens rather than hundreds of nucleotides).

Further details about the decoding algorithm are given in SI Appendix, Supplementary Text.

**Use of Salt to Protect Critical Message.** In Eq. 3 and Fig. 1D, we allowed for some number of bits of known salt \( S_i \) when message bit \( b_i \) is encoded. The use of salt is optional, but desirable in an overall interleaved design where several arranged strands from a pool need to be correctly ordered at decode time (SI Appendix, Supplementary Text). This is generally the case when the outer code is interleaved across strands. To the outer decoder, each incorrectly ordered strand is equivalent to a full strand length of random errors, so it is very important to protect strand ID message bits that determine the strand ordering for outer decoding. Here is how salt is enabling of extra protection: Suppose we want to protect an initial \( s \) message bits. Then define, recursively, the salt by

\[
S_0 = b_0;
S_i = S_{i-1} \oplus b_i, \quad i = 1, \ldots, s - 1 \quad (\text{denoting concatenation}). \quad [4]
\]

Most errors in the first \( s \) bits will be corrected as usual by the shortest-path heap search. But any residual error that gets through will "poison" the salt for the entire rest of the strand, rendering it unreadable. In effect, we convert an error in the protected bits into an erasure of the whole strand. This may seem drastic, but it is just what we want: A strand with incorrect serial number (and hence incorrect ordering among other strands) would look like a strand of errors to the outer code; an erased strand is equivalent to only half as many errors.

**Code Rates Other than One-Half.** A simple modification of the encode and decode algorithms described above allows for code rates other than one-half. Take the input bitstream of expression 1 and partition it into a stream of values \( v_i \) with variable numbers of bits in the range 0 to 2, according to a repetitive pattern like the ones shown in Table 3. See also SI Appendix, Fig. S1.

Here are two examples showing how to interpret the entries in Table 3 (with adjacency denoting 2-bit values in \( \mathbb{Z}_4 \)):

- Rate 0.750: \( v_0 = b_0 b_1, v_1 = b_2, v_2 = b_1 b_2, v_3 = b_3, \ldots \)
- Rate 0.250: \( v_0 = b_0, v_1 = 0, v_2 = b_1, v_3 = 0, \ldots \)

Eq. 3 for encoding now becomes

\[
C_i = K_i + v_i = \mathbb{F}(S_i, i_v, V_i) + v_i \pmod{4}, \quad [5]
\]

where \( v_i \) is composed of concatenated previous variable bits. Pattern values of 0 provide 1 bit of additional redundancy check relative to the base case of code rate one-half, while pattern values of 2, encoding 2 bits per DNA character, provide 1 bit less (i.e., zero). By construction, the code rate is one-half the average of the integers in the pattern. The column in the table labeled \( P_{uk} \) is a "greediness parameter" that mitigates the tendency of the heap to expand exponentially; see SI Appendix, Supplementary Text for details of this.

Decoding follows exactly the same pattern. Guessing a 2-bit \( v_i \) spawns 12 child hypotheses, while guessing a zero-bit \( v_i \) spawns only 3.

**HEDGES Parameters.** For encoding, the parameter choices are 1) the choice of code rate and variable bit pattern (as in Table 3), the default case being code rate 0.5; 2) the number \( q > 0 \) of low-order bits of position index in the hash; 3) the number \( p > 0 \) of previous message bits in the hash; 4) the number \( s \geq 0 \) of salt bits; and 5) the number \( n \geq 0 \) of initial message bits to be protected by salt. Aspects of the choices of, and trade-offs among, these parameters are further discussed in SI Appendix, Supplementary Text.

It is an important point that choosing the decode runtime parameters, for example, \( H_{min} \) or \( P_{uk} \), is not an irrevocable choice. Given a DNA message, one can make multiple tries, varying the decode parameters adaptively until acceptable performance is achieved. Simply increasing \( H_{min} \) and retrying will often rescue failed decodes (SI Appendix, Supplementary Text and Fig. S6). One can evaluate success by running time and by the count of errors needing correction by the outer RS code. The parameter values that we suggest may be viewed as starting points.

**Imposing DNA Output Sequence Constraints.** DNA synthesis and sequencing platforms have sequence-dependent error profiles. Imbalanced GC content and homopolymer runs are well known to be problematic, for example, leading to indel and substitution errors or even whole strand dropout errors in popular sequencers such as those from Illumina and Oxford Nanopore (19, 20). Thus, many proposed ECCs impose constraints on GC content, homopolymer runs, or both. These typically involve one-off coding designs for each constraint, often reducing significantly the effective code rate \( (7) \).

An important property of the HEDGES algorithm is that it can accommodate a large class of sequence constraints without additional, one-off, code design. Moreover, constraints may be imposed without decreasing the code rate (a seeming paradox that we explain below).

Consider the class of constraints that can be applied to a nucleotide sequence with only a "past" information. That is, from the emitted sequence \( \ldots, C_{i-3}, C_{i-2}, C_{i-1}, C_i \) we can determine whether the choice of \( C_i \) is constrained and, if so, what is the set of its allowed values, here denoted \( \{C_i^*\} \), whose size we denote \( \#C_i^* \). We assume \( 0 < \#C_i^* \leq 4 \), because a zero value would imply a constraint so severe that the strand cannot be continued at

<table>
<thead>
<tr>
<th>Code Rate</th>
<th>Pattern</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.750</td>
<td>2, 1, 2, 1, \ldots</td>
</tr>
<tr>
<td>0.600</td>
<td>2, 1, 1, 1, 1, 1, \ldots</td>
</tr>
<tr>
<td>0.500</td>
<td>1, 1, \ldots</td>
</tr>
<tr>
<td>0.333</td>
<td>1, 1, 0, 1, 1, 0, \ldots</td>
</tr>
<tr>
<td>0.250</td>
<td>1, 0, 1, 0, \ldots</td>
</tr>
<tr>
<td>0.166</td>
<td>1, 0, 0, 1, 0, \ldots</td>
</tr>
</tbody>
</table>

*See Code Rates Other than One-Half.
The number of erasures is thus a random Poisson variable and the fraction (in either DNA or bytes) of such erasures is thus sufficiently rare. (One could adopt the convention of relaxing such constraints if they are sufficiently rare. )

While the sum of two Poisson variables is Poisson, the factor one-half in practically (in their power-of-ten exponents, which is all we care about). As

functions (21). In our regime of interest, the upper and lower bounds (22),

one-half and pretend that it is still Poisson.

While the sum of two Poisson variables is Poisson, the factor one-half in practically (in their power-of-ten exponents, which is all we care about). As

functions (21). In our regime of interest, the upper and lower bounds (22),

one-half and pretend that it is still Poisson.

While the sum of two Poisson variables is Poisson, the factor one-half in practically (in their power-of-ten exponents, which is all we care about). As

functions (21). In our regime of interest, the upper and lower bounds (22),

one-half and pretend that it is still Poisson.

While the sum of two Poisson variables is Poisson, the factor one-half in practically (in their power-of-ten exponents, which is all we care about). As

functions (21). In our regime of interest, the upper and lower bounds (22),

one-half and pretend that it is still Poisson.

While the sum of two Poisson variables is Poisson, the factor one-half in practically (in their power-of-ten exponents, which is all we care about). As

functions (21). In our regime of interest, the upper and lower bounds (22),

one-half and pretend that it is still Poisson.

While the sum of two Poisson variables is Poisson, the factor one-half in practically (in their power-of-ten exponents, which is all we care about). As

functions (21). In our regime of interest, the upper and lower bounds (22),

one-half and pretend that it is still Poisson.

While the sum of two Poisson variables is Poisson, the factor one-half in practically (in their power-of-ten exponents, which is all we care about). As

functions (21). In our regime of interest, the upper and lower bounds (22),

one-half and pretend that it is still Poisson.

While the sum of two Poisson variables is Poisson, the factor one-half in practically (in their power-of-ten exponents, which is all we care about). As

functions (21). In our regime of interest, the upper and lower bounds (22),

one-half and pretend that it is still Poisson.

While the sum of two Poisson variables is Poisson, the factor one-half in practically (in their power-of-ten exponents, which is all we care about). As

functions (21). In our regime of interest, the upper and lower bounds (22),

one-half and pretend that it is still Poisson.

While the sum of two Poisson variables is Poisson, the factor one-half in practically (in their power-of-ten exponents, which is all we care about). As

functions (21). In our regime of interest, the upper and lower bounds (22),

one-half and pretend that it is still Poisson.

While the sum of two Poisson variables is Poisson, the factor one-half in practically (in their power-of-ten exponents, which is all we care about). As

functions (21). In our regime of interest, the upper and lower bounds (22),

one-half and pretend that it is still Poisson.

While the sum of two Poisson variables is Poisson, the factor one-half in practically (in their power-of-ten exponents, which is all we care about). As

functions (21). In our regime of interest, the upper and lower bounds (22),

one-half and pretend that it is still Poisson.

While the sum of two Poisson variables is Poisson, the factor one-half in practically (in their power-of-ten exponents, which is all we care about). As

functions (21). In our regime of interest, the upper and lower bounds (22),

one-half and pretend that it is still Poisson.