Modular non-repeating codes for DNA storage

Ian Holmes[1,*]
1 Department of Bioengineering, University of California, Berkeley, CA, USA
* E-mail: ihh@berkeley.edu

1 Abstract

We describe a strategy for constructing codes for DNA-based information storage by serial composition of weighted finite-state transducers. The resulting state machines can integrate correction of substitution errors; synchronization by interleaving watermark and periodic marker signals; conversion from binary to ternary, quaternary or mixed-radix sequences via an efficient block code; encoding into a DNA sequence that avoids homopolymer, dinucleotide, or trinucleotide runs and other short local repeats; and detection/correction of errors (including local duplications, burst deletions, and substitutions) that are characteristic of DNA sequencing technologies. We present software implementing these codes, available at `github.com/ihh/dnastore`, with simulation results demonstrating that the generated DNA is free of short repeats and can be accurately decoded even in the presence of substitutions, short duplications and deletions.

Keywords: DNA storage, finite-state transducers, mixed-radix trees, deletion-insertion correcting codes

Contents

2 Introduction

DNA can store petabytes of information per gram [1] and can last intact for tens of thousands of years [2]. This makes it an appealing prospect for long-term archival storage. However, DNA synthesis, sequencing, and replication are prone to errors, which may limit its potential as a storage medium [3]. These errors can be controlled by applying the tools of information theory, treating DNA storage as a noisy channel coding problem.

Recently, several coding schemes for DNA storage have been proposed that address the interrelated issues of error avoidance, error correction and redundancy [1, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14]. Goldman *et al* [1] made DNA using a ternary (radix-3) code that avoids repeated nucleotides, synthesizing to fourfold redundancy for additional error correction. Gupta *et al* introduced a ternary Golay code for error correction [7]. Strauss *et al* used a Huffman code to map binary to ternary, and further improved on Goldman *et al*'s redundancy coding using an exclusive-or method [14]. Milenkovic *et al* have considered the avoidance of DNA structure in codes [4], proposed several codes with analysis of their combinatoric properties [8, 9, 10, 13] and presented a design for a random-access rewritable filesystem [6]. Avoidance of secondary structure was also discussed by Hunt *et al* [11]. Studies by Yachie *et al* [5] and Jain *et al* [12] have specifically addressed the storage of information in the DNA of living organisms, the latter work (which appeared while this manuscript was in preparation) describing a duplication-correcting code. In earlier work not specifically oriented to DNA storage, MacKay *et al* developed synchronizing error-correcting codes that are resistant to substitutions, insertions and deletions ("indels"), including watermark codes [15, 16] and marker codes [17].

Here we describe a modular strategy for constructing error-tolerant DNA codes. As noted above, much recent work has used Goldman *et al*'s ternary code as a starting point. The ternary code arises from avoidance of dinucleotide repeats, which leaves 3 available possibilities at each position; hence the radix of 3. The rationale advanced by Goldman *et al* for this system is that it guards against the most common class of errors in DNA resequencing, which occur at homopolymer runs. However, there are hidden assumptions in this reasoning; specifically, that it is worth spending a particular amount on synthesis in order to mitigate a certain class of technology-dependent sequencing error. Assuming that this is the case, it may also be desirable to avoid other error-prone motifs, such as repeated dinucleotides, trinucleotides or longer repeats. Avoiding these sequences does not reduce the information content of DNA much, but they are not too far off homopolymer runs in their error rate [18]. We may also wish to introduce error-correcting codes such as Hamming codes [19], Golay codes [7], turbo codes [20, 21], or Gallager codes [22]. For entirely independent reasons, we might also want to constrain the coding scheme to avoid certain reserved codewords, whether for biological function (e.g. binding sites for restriction enzymes or other factors) or to organize our nucleic acid filesystem (e.g. barcodes, synchronization signals, or metadata block boundaries). And yet none of this strategy can be taken for granted: much of the above reasoning is predicated on the idea that it's worth mitigating error by synthesizing more DNA, but (for the moment at least) synthesis is vastly more expensive than sequencing. An alternate strategy would be to pack in as much information as possible and assume that sequencing will be cheap enough to correct errors.

In short, different application requirements and changing economics may require different codes. As noted above, several researchers have developed solutions to some of these problems. Here, we combine some of these ideas, and introduce some new ones, using a modular strategy for code design. With this method, codes can be assembled to meet requirements including error-avoidance, error-correction, and demarcation of metadata.

The core idea of our approach is to convert raw binary data into a mixed-radix sequence wherein successive digits may be binary (radix-2), ternary (radix-3) or quaternary (radix-4). The radix at any given position is effectively specified by the number of nucleotides available for coding at that position (which may vary due to avoidance of repeats or reserved motifs). The mixed-radix sequence can then be efficiently converted to a DNA sequence. Error-correcting codes (for example, Hamming codes that introduce parity bits, or codes that maintain synchronization in the presence of indels) can be applied upstream of the conversion from binary to mixed-radix, while the readout process (errors from which may reintroduce repeats and prohibited motifs) can be integrated into the decoding model as a step that follows the conversion from mixed-radix to DNA.

Our strategy uses several techniques borrowed from other areas of bioinformatics and information theory: finite-state transducers, De Bruijn graphs, arithmetic coding, and synchronization codes. Foremost among these are transducers [23, 24], automata-theoretic models of sequence transformation which we use to represent in a uniform way the various steps of error-correction (e.g. via introduction of parity or watermark bits), radix conversion, repeat-avoidant nucleotide encoding, and random sequencing error. Standard algorithms for combining and decoding transducers can then be applied. Transducers have been previously used in bioinformatics for protein classification [25], phylogenetic indel models [26, 27, 28], and cancer informatics [29] To build the transducer that converts a mixed-radix sequence into a non-repeating DNA sequence, we make use of De Bruijn graphs and their relationship to finite-context automata (the probabilistic versions of which are known as order-N Markov models). De Bruijn graphs are widely used for genome assembly [30, 31, 32, 33], while order-N Markov models are often employed for gene-finding [34, 35]. To build the transducer that converts a binary sequence into a mixed-radix sequence, we borrow concepts from the arithmetic coding algorithm [36, 19].

3 Methods

Section 3.1 reviews preliminary concepts such as operations on weighted finite-state transducers, De Bruijn graphs, patterns in DNA sequences, and the arithmetic coding algorithm. Section 3.2 describes the repeat-avoiding transducer that accepts a mixed-radix sequence on its input and generates a DNA sequence without homopolymers, short microsatellites or other local repeats. Section 3.2.1 reformulates this transducer so each state has "knowledge" of its past and future context, facilitating error models that use this context to identify local duplications or context-sensitive substitutions. Section 3.3 develops a transducer motivated by arithmetic coding that converts a binary input sequence into a mixed-radix sequence suitable for subsequent encoding as DNA. Section 3.4 describes an error-model transducer that mutates and deletes bases and reintroduces repeats. Section 3.5 describes a transducer that introduces markers and a watermark pilot sequence for synchronization. Section 3.6 shows how all these transducers can be combined.

3.1 Preliminary concepts

3.1.1 Weighted finite-state transducers

Following [23]: Assume a general semiring $K = (\mathbb{K}, \oplus, \otimes, \underline{0}, \underline{1})$ which for our purposes is typically the probability semiring $(\Re, +, \times, 0, 1)$ or the tropical semiring $(\Re_+ \cup \infty, \min, +, \infty, 0)$.

A weighted finite-state transducer is defined as a tuple $T = (\Sigma, \Omega, Q, E, i, f)$ consisting of an input alphabet Σ, an output alphabet Ω (both alphabets being finite sets), a finite set of states Q, a finite set of transitions $E \subseteq Q \times (\Sigma \cup \{\epsilon\}) \times (\Omega \cup \{\epsilon\}) \times \mathbb{K} \times Q$, an initial state $i \in Q$ and a final state $f \in Q$.

The transducer T can be thought of as an edge-labelled directed graph where each state is a vertex and each transition $t = (p[t], \ell_i[t], \ell_o[t], \mathbb{W}[t], q[t]) \in E$ is an edge from state $p[t]$ to state $q[t]$ with input label $\ell_i[t]$, output label $\ell_o[t]$ and weight $\mathbb{W}[t]$.

A path in T is a series of transitions that form a path in this graph. The input sequence and output sequence for a path are the concatenation of (respectively) the input and output labels of the transitions in the path. The path weight is the $\otimes$-product of the transition weights. A successful path is one that starts in i and ends in f. The transduction weight for a given input sequence $\sigma \in \Sigma^*$ and output sequence $\omega \in \Omega^*$ is the $\oplus$-sum of all successful paths having σ as the input sequence and ω as the output sequence. Thus T provides a mapping $\mathbb{T} : (\Sigma^* \times \Omega^*) \to \mathbb{K}$ from sequence-pairs to weights. We call this mapping $\mathbb{T}$ the transducer function. For a given pair of sequences (σ, ω) and a semiring wherein $\oplus$ and $\otimes$ are amortized-constant resource operations, it can be evaluated in time $\mathcal{O}(|\sigma|{\cdot}|\omega|{\cdot}|E|)$ and memory $\mathcal{O}(|\sigma|{\cdot}|\omega|{\cdot}|Q|)$ by dynamic programming, analogously to the Forward algorithm in the probabilistic semiring or the Viterbi algorithm in the tropical semiring [37].

A state $q \in Q$ has past input context $\bar{v}$ if every path from i to q has an input sequence with suffix $\bar{v}$, and future input context $\bar{w}$ if every path from q to f has an input sequence with prefix $\bar{w}$. The past output context and future output context of a state are defined similarly.

A "waiting state" is a state that has no outgoing transitions with empty input labels. A "waiting machine" is a transducer where all the transitions with nonempty input labels originate from waiting states. Any transducer T with a transducer function $\mathbb{T}$ can be transformed into an equivalent waiting machine that has the same transducer function $\mathbb{T}$ and at most $2|Q|$ states and $|Q| + |E|$ transitions [27].

3.1.2 Transducer composition

Given transducers $R = (\Sigma, \Gamma, Q_R, E_R, i_R, f_R)$ and $S = (\Gamma, \Omega, Q_S, E_S, i_S, f_S)$ where R's output alphabet is the same as S's input alphabet, we can readily find a composite transducer $T = R + S = (\Sigma, \Omega, Q_T, E_T, i_T, f_T)$ such that, if $\mathbb{R}$, $\mathbb{S}$ and $\mathbb{T}$ are the corresponding transducer functions, then

$$\forall \sigma \in \Sigma^*, \omega \in \Omega^* : \quad \mathbb{T}(\sigma, \omega) = \bigoplus_{\gamma \in \Gamma^*} \mathbb{R}(\sigma, \gamma) \mathbb{S}(\gamma, \omega)$$

that is, T models the feeding of R's output into S's input (and this intermediate sequence is then summed out—i.e. marginalized, if we are in the probabilistic semiring).

Loosely speaking, we can construct T using the following recipe:

- Each T-state corresponds to a pair of R- and S-states, so $q_T = (q_R, q_S)$ and $Q_T \subseteq Q_R \times Q_S$.
- The initial T-state $i_T = (i_R, i_S)$ pairs the initial states of R and S.
- The final T-state $f_T = (f_R, f_S)$ pairs the final states of R and S.
- T-transitions $t_T = ((p_R, p_S), \ell_i, \ell_o, \mathbb{W}_T, (q_R, q_S))$ can represent synchronized pairs of R-transitions $t_R = (p_R, \ell_i, \ell_m, \mathbb{W}_R, q_R)$ and S-transitions $t_S = (p_S, \ell_m, \ell_o, \mathbb{W}_S, q_S)$ which share the same intermediate label ℓ_m. The composite transition weight $\mathbb{W}_T$ is the product $\mathbb{W}_R \otimes \mathbb{W}_S$.

- T-transitions can also represent transitions wherein only one of R or S changes state. In these transitions, the intermediate label (ℓ_m) and either the input or output label (ℓ_i or ℓ_o) must be empty. Some further synchronization may be required; for example, we can require that S is a waiting machine, and that R can only change state when S is in a waiting state [27].

This algorithm can be made precise enough to automate; more detailed workings are given elsewhere [38, 23, 39, 40, 27, 28]. The transducers yielded by a fully automated implementation can typically be aggressively optimized by hand to minimize the number of transitions and/or states, and hence, the time and/or memory complexity of dynamic programming algorithms.

3.1.3 Transducer concatenation

Another operation on transducers that is useful in constructing DNA codes is concatenation. Suppose $A = (\Sigma, \Omega, Q_A, E_A, i_A, f_A)$ and $B = (\Sigma, \Omega, Q_B, E_B, i_B, f_B)$ are transducers with the same input and output alphabets and disjoint state spaces. The concatenated transducer $T = A \circ B$ can be constructed by taking the union of the state spaces and adding an ϵ-labeled unit weight transition from f_A to i_B. This models feeding the part of a sequence through A and then feeding the second part through B.

3.1.4 Transducer union

The union of two transducers (whose state spaces are assumed disjoint) is found by taking the union of their state spaces and transition graphs and adding a new initial state with two unit-weight transitions, one to each of the initial states of the two transducers being combined.

3.1.5 Kleene closure

The operation analogous to Kleene closure (generating all strings over a given alphabet) is simply to add a unit-weight transition from the final state of the transducer back to the initial state.

3.1.6 De Bruijn graphs

Denote by $\Gamma^{\mathcal{K}}$ the set of all possible $\mathcal{K}$-symbol strings $x_1 x_2 \ldots x_{\mathcal{K}}$ over some alphabet Γ.

The $\mathcal{K}$-dimensional De Bruijn graph over Γ has vertex set $\Gamma^{\mathcal{K}}$ and a directed edge $u \to v$ between any two vertices $u = x_1 x_2 \ldots x_{\mathcal{K}}$ and $v = x_2 \ldots x_{\mathcal{K}} x_{\mathcal{K}+1}$ that overlap by $\mathcal{K} - 1$ symbols; this edge is labeled with symbol $x_{\mathcal{K}+1}$ (the last symbol of v). Thus, each vertex has $|\Gamma|$ incoming and $|\Gamma|$ outgoing edges [30, 31]. Denote this graph by $\mathcal{G}_b = (\Gamma^{\mathcal{K}}, \mathcal{E}_b)$.

3.1.7 DNA alphabets and repeats

Let $\Gamma_{\text{DNA}} = \{\text{A}, \text{C}, \text{G}, \text{T}\}$ be the nucleotide alphabet. Let $\tilde{x}$ denote the complement of a nucleotide symbol x, and $\tilde{\gamma}$ the reverse complement of a nucleotide sequence γ. Let $\hat{x}$ denote the nucleotide related to x by a transition substitution (so $\hat{\text{A}} = \text{G}$, $\hat{\text{C}} = \text{T}$, etc.).

A direct tandem repeat of length k is a nucleotide sequence followed by an exact copy of itself, $\gamma\gamma$, where the length of each copy is $|\gamma| = k$ (note this includes repeated single nucleotides when $k = 1$). Similarly, a direct inverted repeat of length k is a k-nucleotide sequence followed by its reverse complement, $\gamma\tilde{\gamma}$. A local inverted repeat of length k and separation l is a k-nucleotide sequence, followed an l-nucleotide sequence, followed by the reverse complement of the first sequence; that is, $\gamma\lambda\tilde{\gamma}$, where $|\gamma| = k$ and $|\lambda| = l$.

These repeats arise very commonly in naturally occurring DNA (e.g. see [41]), and are also hotspots for sequencing error [18].

3.1.8 Arithmetic coding

Arithmetic coding is a method for compressing a message $x_1 x_2 \ldots x_N$ given a probability model over messages whose predictive marginals for the next character in the sequence, $Q_n(k) \equiv P(x_{n+1}|x_1 \ldots x_n)$, can be efficiently evaluated. Suppose without loss of generality that the symbol alphabet (including the end-of-block character) is the set of integers $1 \ldots K$ and let $C_n(k) \equiv P(x_{n+1} \leq k|x_1 \ldots x_n)$ be the cumulative distributions for the predictive marginals. Arithmetic coding uses the following ideas:

- The probability distribution over the first character divides the interval $[0, 1)$ into K subintervals. The k'th subinterval is $[C_1(k - 1), C_1(k))$.

- Each subinterval is then further subdivided by the second character, then the third, and so on until the end of message. So, for example, the subinterval for the first character is $[C_1(x_1-1), C_1(x_1))$, for the second character $[C_1(x_1 - 1) + Q_1(x_1)C_2(x_2 - 1), C_1(x_1) + Q_1(x_1)C_2(x_2))$, and so on.

- If the interval before encoding the n'th character is $[A_n, B_n)$ then $[A_0, B_0) = [0, 1)$, $A_{n+1} = (B_n-A_n)C_{n+1}(x_{n+1}-1)$, and $B_{n+1} = (B_n-A_n)C_{n+1}(x_{n+1})$.

- The upshot of all of this is that each message is associated with a finite subinterval of $[0, 1)$ and these subintervals are lexicographically ordered (i.e. sorted by the first symbol, then the second, then the third, and so on).

- Any finite-precision floating-point number also specifies a subinterval of $[0, 1)$ containing all floating-point numbers of greater precision that would be rounded to that number. For example, the decimal floating-point number 0.413 specifies the subinterval $[0.4125, 0.4135)$.

- To encode the message $x_1 x_2 \ldots x_N$, we simply need to transmit enough digits of a (binary) floating-point number such that its rounding interval is fully contained within the interval $[A_{N+1}, B_{N+1})$. The number of floating-point binary digits this requires should not exceed the Shannon entropy of the message, $-\log_2 P(x_1 x_2 \dots x_N)$, by more than one bit.

3.2 A transducer for encoding signals as DNA without short repeats or reserved words

In this section we contruct, in several steps, a transducer T_{code} that accepts a mixed-radix sequence on the input (that is, every input state either accepts binary symbols $\{0_2, 1_2\}$, ternary symbols $\{0_3, 1_3, 2_3\}$ or quaternary symbols $\{0_4, 1_4, 2_4, 3_4\}$) and outputs a (uniquely decodable) nucleotide sequence that is free of repeated nucleotides, short tandem repeats, or inverted repeats. The input sequence may optionally be interleaved with special control digits $\{M[i] : 1 \leq i \leq N_c\}$ which force the machine to output predictable, recognizable nucleotide sequences that can be used to flag metadata, demarcate boundaries, or to embed biologically functional motifs.

Start with $\mathcal{G}_b$, the $\mathcal{K}$-dimensional De Bruijn graph over Γ_{DNA}. Delete all nodes corresponding to sequences that are (or contain substrings which are) direct tandem repeats of any length, direct inverted repeats of length ≥ 2, or local inverted repeats of length $\geq L_{\text{invrep}}$ and separation ≥ 2. Denote the resulting graph by $\mathcal{G}_r = (\mathcal{V}_r, \mathcal{E}_r)$.

Let $A \subset \mathcal{V}_r$ be a set of vertices to avoid, let $D \in A$ be a target vertex, and denote by $\mathcal{S}(D, A, N) \subset \mathcal{V}_r$ the set of all vertices from which there exists a length-N path to D that does not pass through any of the vertices in A (although the path is allowed to originate from one of those vertices). Define $\mathcal{L}(D, A)$ to be the smallest value of N for which $\mathcal{S}(D, A, N) = \mathcal{V}_r$, if such a value of N exists; otherwise, let $\mathcal{L}(D, A)$ be ∞. We can find $\mathcal{S}(D, A, N)$ from $\mathcal{S}(D, A, N-1)$ by recursive backtracking, and so determine in k steps whether $\mathcal{L}(D, A) \leq k$.

We now allocate N_c "control words": $\mathcal{K}$-mers that will be reserved for marking metadata boundaries, such as the start and end of messages. In the transducer, these will be unreachable except by specially constructed paths of uniform length. Specifically, we seek an indexed set of vertices $\mathcal{M} = \{M_1, M_2 \dots M_{N_c}\}$ such that $\mathcal{L}(M_n, \mathcal{M}) \leq k$ for all n and for some pre-specified value of k. Our implementation finds this list $\mathcal{M}$ by brute-force recursive search (using $k = 2\mathcal{K}$) and further attempts to maximize the shortest Hamming distance between any two control words in $\mathcal{M}$. Note that there is a ceiling to the number of control words that may be found for any given value of $\mathcal{K}$ (and L_{invrep}), though this ceiling grows rather rapidly with $\mathcal{K}$. In practice we only need a few control words for most purposes.

Having designated some $\mathcal{K}$-mers as control words via analysis of $\mathcal{G}_r$, we now construct a new graph $\mathcal{G}_c$ in which the control words are unreachable from the other words except by paths that we construct. Starting with graph $\mathcal{G}_r$, delete all incoming transitions to control words $M_n \in \mathcal{M}$, rendering them unreachable, and prune the graph of any other nodes that become unreachable as a result. Next, for every control word $M_n \in \mathcal{M}$ and every path length

$1 \leq k < \mathcal{L}(M_n, \mathcal{M})$, we create a new vertex set $\mathcal{V}_c(n,k)$ with a one-to-one correspondence to $\mathcal{S}(M_n, \mathcal{M}, k)$. We connect the newly-added vertices such that there is an edge from $u \in \mathcal{V}_c(n,k)$ to $v \in \mathcal{V}_c(n,k-1)$ for every corresponding edge $(u', v') \in \mathcal{E}_r$ between $u' \in \mathcal{S}(M_n, \mathcal{M}, k)$ and $v' \in \mathcal{S}(M_n, \mathcal{M}, k-1)$. We also add edges from $u \in \mathcal{V}_r$ to $v \in \mathcal{V}_c(n, \mathcal{L}(M_n, \mathcal{M}))$ for the first steps in the paths to control words, as well as edges from $u \in \mathcal{V}_c(n,1)$ to M for the final steps. In each case the newly-added edge is copied from a corresponding edge in $\mathcal{E}_r$ and inherits the same edge label.

It can be useful to force the transducer to start with a particular control word M_i and finish with another (or the same) control word M_f. To guarantee this we can add a chain of initial vertices and edges leading from source vertex to the initial control word, $u_i^{(0)} \to u_i^{(1)} \to u_i^{(2)} \to \ldots \to u_i^{(\mathcal{K}-1)} \to M_i$, with each transition labeled with consecutive symbols from the initial control word. We also need to add a transition from the final control word to the final state. (If the start and end control words are the same, then this vertex should be duplicated to prevent cycles.)

We have now constructed the graph $\mathcal{G}_c$ from which our transducer T_{code} is derived via the following recipe:

- Every vertex in $\mathcal{G}_c$ is a state in T_{code}. The initial and final vertices of $\mathcal{G}_c$ are the initial and final states of T_{code}.
- Every edge in $\mathcal{G}_c$ is a unit-weight transition in T_{code}. The output label of the transition is the label of the edge.
- The input label of each transition is determined as follows:
  - For states in $\mathcal{V}_r$ with two outgoing transitions to other states in $\mathcal{V}_r$, those transitions are input-labeled with the binary digits (bits) 0_2 and 1_2.
  - For states in $\mathcal{V}_r$ with three outgoing transitions to other states in $\mathcal{V}_r$, those transitions are input-labeled with the ternary digits (trits) 0_3, 1_3 and 2_3.
  - For states in $\mathcal{V}_r$ with four outgoing transitions to other states in $\mathcal{V}_r$, those transitions are input-labeled with the quaternary digits (quats) 0_4, 1_4, 2_4 and 3_4.
  - Transitions from states in $\mathcal{V}_r$ to states in $\mathcal{V}_c(n, \mathcal{L}(M_n, \mathcal{M}))$, which begin a path to the n'th control word M_n, are input-labeled with the special control digit $M[n]$.
  - All other transitions have input label ϵ.

Thus, the input alphabet of T_{code} is $\{0_2, 1_2, 0_3, 1_3, 2_3, 0_4, 1_4, 2_4, 3_4, M[1] \ldots M[N_c]\}$.

Figure 4 illustrates some of the code transducers that are generated by this procedure for the simplest case $\mathcal{K} = 2$.

3.2.1 Trading past context for future context

By virtue of derivation from the De Bruijn graph, most of the states in the transducer T_{code} of Section 3.2 have $\mathcal{K}$ nucleotides of past output context. We can transform this transducer into an equivalent one T_{delay} wherein most of the states have $\mathcal{K}/2$ nucleotides of past output context and $\mathcal{K}/2$ nucleotides of future output context.

This can be a convenient way to think about context, for the purpose of modeling errors in DNA replication and sequencing. Common decoding and replication errors include local tandem and inverted duplications, as well as context-dependent substitutions. These can occur on either strand of the DNA double helix and therefore (depending on the representation) may be best represented as depending on future context as well as past context.

The transformation requires that the output sequence of all successful paths through the transducer begin with a particular $\mathcal{K}$-mer and end with a particular $\mathcal{K}$-mer, which can be ensured using the method described in Section 3.2.

Let $v_1 \dots v_{\mathcal{K}} = \bar{v}$ be the past output context for a given state q. All of the transitions t into q have the same input label $\ell_i[t] = v_{\mathcal{K}}$, which corresponds to the most recent nucleotide of output context for that state. If, instead, we set $\ell_i[t]$ to be $v_{\mathcal{K}/2}$, the $(\mathcal{K}/2)$'th symbol of γ, for all transitions into q, then we have effectively delayed all output by $\mathcal{K}/2$ symbols. We can now predict the next $\mathcal{K}/2$ symbols in the output sequence for the path from a state, so we have traded $\mathcal{K}/2$ of past output context for $\mathcal{K}/2$ of future output context.

We need to add $\mathcal{K}/2$ extra padding states after (what was originally) the final state, with a chain of transitions that flushes out the final $\mathcal{K}/2$ delayed output symbols. Conversely, the first $\mathcal{K}/2$ transitions from the initial state (into states with $\mathcal{K}/2$ or fewer nucleotides of output context) will, after the transformation, be null transitions (with both input and output labels equal to ϵ), so these transitions and the corresponding states can be deleted.

An analogous procedure can be used to transform a machine with $\mathcal{K}$ symbols of past input context into a machine with $\mathcal{K}/2$ past input context and $\mathcal{K}/2$ future input context.

3.3 A transducer that converts a binary sequence into a mixed-radix sequence of binary, ternary and quaternary digits

We here describe a transducer T_{radix} for converting the binary sequence to the mixed-radix sequence that we use to encode it in DNA that is free of short repeats. We can always convert a binary sequence to a mixed-radix sequence trivially by encoding 0_2 and 1_2 as 0_R and 1_R in whatever radix R is available; that is, we simply never use the extra symbols in our alphabet $\{2_3, 2_4, 3_4\}$. However, we can do better than this and pack some extra information into the additional symbols when they are available. There are at least two issues we need to consider here: (i) efficient conversion between binary and other radices (especially ternary, which is the trickiest) via block codes, while allowing for prematurely truncated blocks; (ii) the fact that our repeat-avoiding DNA code is mixed-radix, that is, it has a radix that varies from position to position.

Considering first the issue of conversion from binary to ternary, it is tempting to try something like the machine of Figure 2, which maps binary $0_2 0_2$ to ternary 0_3, $0_2 1_2$ to 1_3, and 1_2 to 2_3; that is, it sometimes encodes two bits as one trit. In fact, it does this exactly half the time (assuming a uniform distribution over input bits) so there are, on average, 1.5 input bits per output trit, which is quite close to the theoretical maximum of $\log_2(3) \simeq 1.585$ bits per trit. However, we have to be a little careful with this approach: the machine of Figure 2 can get stuck in a state where it has queued up a zero input bit and is waiting for the next input bit before it outputs a trit. If the input ends at this point, or if the encoded signal includes a non-bit symbol such as one of the reserved control symbols we allowed for in Section 3.2, then we have to flush out that queued-up zero bit somehow.

A fragile workaround to this problem is to send an extra padding bit in such cases; more robustly, we can introduce an end-of-block symbol '$', perhaps as part of a block code. Of course, once we start adding additional symbols like end-of-block, the number of input bits we can encode per output trit (or output bit, or quat) has to fall, since we have to reserve some codewords for transmitting these symbols.

The Huffman block code for binary encoding is well known, and generalizes readily to ternary [14]. This brings us, however, to the second issue we face in radix conversion. A ternary Huffman code is imperfectly suited to the coding of information of DNA wherein we are trying to avoid dinucleotide, trinucleotide or higher-order repeats; nor is it well-suited to the situation where we are allowing repeats but avoiding certain reserved words. The reason for the mismatch is that, in such situations, the number of available nucleotides at any given position may vary between 1, 2, 3 or 4. Positions where only one nucleotide is legal cannot carry information; but if there are 2, 3 or 4 options, then we need to think about encoding a binary, ternary, or quaternary digit, as appropriate.

For mixed-radix output, in the special case where the radices at each position are fully specified, the relevant cousin of the Huffman code is the mixed-radix Huffman code, for the discovery of which optimal dynamic programming strategies have been presented [42, 43]. However, taking this approach in our situation would require a different mixed-radix Huffman code for every possible combination of radices, and so is not particularly convenient for the situation when the number of available coding nucleotides varies from site to site (as in the repeat-avoiding code).

In this section, we present a strategy for developing compact automata (transducers) that convert short binary codewords into any mixed-radix encoding. The code approaches optimality at long block lengths. Unfortunately, the number of states used by the machine grows exponentially with the sequence length, so this asymptotic limit is hard to realize. Nevertheless, the codes generated are not totally awful, and approach the asymptotic limit reasonably fast, as shown in Section 4.

We apply the concepts of arithmetic coding (Section 3.1.8) to encode input words as mixed-radix floating point numbers. The set of input words to be encoded is the set of all length-N binary strings, plus the set of all length-k binary strings ($0 \leq k < N$) that are followed by the '\$' character. Thus, if $N = 2$ then the input word set is $\{\$, 0\$, 1\$, 00, 01, 10, 11\}$. There are $2^{N+1} - 1$ words in the input word set.

We assume that a probability model as described above is defined over this input set; typically we will use a simple model that assigns a small probability ν to '\$' at any position, independently of previous context, and splits the remaining probability evenly between 0 and 1. Each input word w is then associated with an interval $[A_w, B_w)$ of size $P(w) = B_w - A_w$.

Since we are working with small fixed-length input codewords and finite-state machines, we will drop the lexicographical ordering of codeword intervals which arithmetic coding uses to implement online compression and decompression for arbitrary-length messages. Instead, we sort the input words w by probability $P(w)$, which will tend to lead to rarer words sharing similar encoded prefixes (c.f. Huffmann coding). We also apply several optimizations (dynamically adjusting the probability distribution to improve performance on rare input words, merging identical states in the transducer, and pruning unnecessary states) that help offset the inefficiency of using a short block code (as compared to arithmetic coding where the block code length is effectively the length of the entire message).

Pseudocode to generate the transducer is shown in Algorithm 1. The essence of the approach is to build a tree of the digits of all possible radices that the arithmetic coding approach would use to encode each position, with some optimizations (as noted) to merge states and adjust output intervals. Figure 3 shows a transducer that was built by this algorithm for 2-digit codewords with $\nu = 1/100$.

3.4 A model of substitutions, local tandem and inverted duplications, and other indels

In this section we describe a statistical error model, implemented as a transducer T_{error}, that includes tandem duplications ($\text{ACG}\rightarrow\text{ACG}\underline{\text{ACG}}$), forward inverse duplications ($\text{ACG}\rightarrow\text{ACG}\underline{\text{CGT}}$), reverse inverse duplications ($\text{ACG}\rightarrow\underline{\text{CGT}}\text{ACG}$), and point substitutions ($\text{ACG}\rightarrow\text{A}\underline{\text{T}}\text{G}$). The duplications can be imperfect: they can include substitutions (e.g. $\text{ACG}\rightarrow\text{ACG}\underline{\text{A}\underline{\text{T}}\text{G}}$). We also describe how to account for partial observation of the sequence (at least at the level of reconstructing individual reads; the broader problem of reassembling a data file from fragments is not addressed here, beyond the general recipe for demarcating metadata with control words that was given in Section 3.2, which can be used to mark up shorter blocks with their locations in the file).

The full working is rather detailed, but the basic idea is very simple. We use the transducer state space to build up a context of past nucleotides (i.e. the last few nucleotides it has most recently seen on the input) and future nucleotides (i.e. the next few nucleotides it is prepared to receive on the input, in a given state). The higher-order structure of the transducer is determined by how much

```
Data: $W$, the list of input words sorted by decreasing probability
Data: $A_w, B_w, P(w)$, the bounds and size of the probability interval for
      each input word
Result: A finite-state transducer that converts an input word into a
        mixed-radix sequence with all possible combinations of binary,
        ternary and quaternary radix at every available position
First construct the prefix tree of the input words, assign a state to every
node and an input-labeled transition to every correspondingly labeled
edge. The root of the prefix tree serves as the initial and final state ;
for $w \in W$ do
    Let $l$ be the leaf $l$ of the prefix tree associated with $w$ ;
    The input interval is $[A_w, B_w)$ ;
    Let $m = A_w + \frac{1}{2}(B_w - A_w)$ be its midpoint ;
    Let $S$ be the list of states still to be processed ;
    Let $F$ be the set of final states ;
    Let $O_s$ denote the output prefix of state $s$ ;
    Let $[D_s, E_s)$ denote the output interval ;
    (The output prefix is the sequence of symbols that must be emitted to
    reach $s$. The output interval is the subinterval of $[0, 1)$ that has been
    encoded by the output prefix) ;
    Initialize $S \leftarrow \{l\}$, $O_l \leftarrow \epsilon$, $D_l \leftarrow 0$, $E_l \leftarrow 1$ ;
    while $S \neq \emptyset$ do
        Let $s$ be the first state in $S$ ;
        Delete $s$ from $S$ ;
        if $D_s \geq A_w$ and $E_s \leq B_w$ then
            Add $s$ to $F$.
        else
            for $R \leftarrow 2$ to $4$ do
                Find the integer $r$ such that if $d(r) = D_s + r(E_s - D_s)/R$
                then $d(r) \leq m < d(r + 1)$ ;
                Create a new state $t$ ;
                Add a transition from $s$ to $t$ with output label $r_R$ ;
                Set $D_t \leftarrow d(r)$, $E_t \leftarrow d(r + 1)$ and $O_t \leftarrow O_s \cdot r_R$ ;
                Add $t$ to $S$.
    (Having generated all mixed-radix encodings for $w$, we can now shrink
    its input interval to just enclose the output intervals used by the
    encodings. Its upper bound drops from $B_w$ to $E_{\max}$, increasing the
    space available for the remaining input words by a factor of $\alpha$) ;
    Let $E_{\max} = \max_{s \in F} E_s$ and $\alpha = (1 - E_{\max})/(1 - B_w)$ ;
    for $x \in W$, $A_x > A_w$ do
        $A_x \leftarrow \alpha(A_x - B_w) + E_{\max}$ ;
        $B_x \leftarrow \alpha(B_x - B_w) + E_{\max}$
If any states have a unique output prefix $O_s$, then remove all their
descendants ;
Merge all states whose sets of possible output sequences are identical ;
Form the Kleene closure of the constructed transducer.
```

Algorithm 1: Algorithm to generate a transducer that converts from binary to a mixed-radix sequence (Section 3.3).

past and future context it has built up (in the early stages), and how much future context it has yet to work through (in the later stages). The local structure of the transducer involves states for substitutions, deletions and insertions (which are actually duplications that copy the local context).

The transducer as constructed can only handle sequences whose length is at least $\mathcal{K}$. Shorter sequences will have no successful path through the machine. This is not envisaged to be a problem for decoding DNA-stored data since reads of length $< \mathcal{K}$ will contain very little data, probably insufficient for assembly and lacking in metadata since $\mathcal{K}$ is the number of nucleotides required to encode a control signal. Nevertheless, the transducer can readily be adapted for such edge cases by adding extra blocks to the higher-order structure (see Figure 5(a)).

The error-model transducer is defined over the probabilistic semiring, has input alphabet Ω, output alphabet Ω, past & future context $L = \mathcal{K}/2$, and the following state space:

- There is a state $S_a(\epsilon, \epsilon)$ which is the initial state
- For every $k : 1 \leq k < L$ and every k-mer $\bar{w} \in \Omega^k$ there is a state $S_b(\epsilon, \bar{w})$
- For every $k : 1 \leq k < L$, every k-mer $\bar{v} \in \Omega^k$ and every L-mer $\bar{w} \in \Omega^L$ there are
  - two states $\{S_c(\bar{v}, \bar{w}),\ D_c(\bar{v}, \bar{w})\}$
  - L states $\{R_i^{(c)}(\bar{v}, \bar{w}, i) :\ 1 \leq i \leq L\}$
  - $2k$ states $\{T_c^{(i)}(\bar{v}, \bar{w}),\ F_c^{(i)}(\bar{v}, \bar{w}) : 1 \leq i \leq k\}$
- For every pair of L-mers $\bar{v}, \bar{w} \in \Omega^L$ there are
  - two states $\{S_d(\bar{v}, \bar{w}),\ D_d(\bar{v}, \bar{w})\}$
  - $3L$ states $\{T_d^{(i)}(\bar{v}, \bar{w}),\ F_d^{(i)}(\bar{v}, \bar{w}),\ R_d^{(i)}(\bar{v}, \bar{w}) :\ 1 \leq i \leq L\}$
- For every $k : 1 \leq k < L$, every L-mer $\bar{v} \in \Omega^L$ and every k-mer $\bar{w} \in \Omega^k$ there are
  - two states $\{S_e(\bar{v}, \bar{w}),\ D_e(\bar{v}, \bar{w})\}$
  - $2L$ states $\{T_e^{(i)}(\bar{v}, \bar{w}),\ F_e^{(i)}(\bar{v}, \bar{w}) :\ 1 \leq i \leq L\}$
  - k states $\{R_e^{(i)}(\bar{v}, \bar{w}) :\ 1 \leq i \leq k\}$
- There is a state $S_f(\epsilon, \epsilon)$ which is the final state

These states have the following significance (see Figure 5):

- S_b states load input symbols into the future context queue
- S_c states have fully loaded future context queues. They continue to load input symbols, but also start emitting output symbols and shifting input symbols to the past context queue

- S_d states have fully loaded past and future context queues. They load input symbols, shift input symbols from future to past context queues, drop input symbols off the back of the past context queue, and emit output symbols

- S_e states have fully loaded past context queues, but emptying future context queues. No future input symbols are loaded at this point. They shift input symbols from future to past context queues, drop input symbols off the back of the past context queue, and emit output symbols

- Figure 5(a) shows an additional block of S_g-states that would be required for the transducer to handle sequences of length $< 2L$, which never make it to block #d since they are too short to fully load the past and future context queues. Since these sequences probably contain too little information to be useful (especially if we are using $2L$-nucleotide words to mark up metadata), we have omitted them from the formal description.

- D_α states are used for deletions ($\alpha \in \{c, d, e\}$)

- $T_\alpha^{(i)}$ states are used for tandem duplications ($1 \leq i \leq L$)

- $F_\alpha^{(i)}$ states are used for forward inverse duplications

- $R_\alpha^{(i)}$ states are used for reverse inverse duplications

- Each state is indexed with its past context $\bar{v}$, its future context $\bar{w}$ and (for duplication states) the remaining duplication length i

The transitions $(p, \ell_i, \ell_o, \mathbb{W}, q)$, that involve S-states, so $p = S_\alpha(\dots)$ and $q = S_\beta(\dots)$, are shown in the table. In all cases $1 < k \leq L$ measures a partial context length, $v_1 \dots v_L \in \Omega$ represent past input context symbols, $w_1 \dots w_L \in \Omega$ represent future input context symbols and $y \in \Omega$ represents an output symbol.

p	ℓ_i	ℓ_o	$\mathbb{W}$	q
$S_a(\epsilon,\ \epsilon)$	w_1	ϵ	1	$S_b(\epsilon,\ w_1)$
$S_b(\epsilon,\ w_1 \dots w_{k-1})$	w_k	ϵ	1	$S_b(\epsilon,\ w_1 \dots w_k)$
$S_b(\epsilon,\ w_1 w_2 \dots w_L)$	w_{L+1}	y	$\mathbb{P}^*_n(0, L) \cdot \mathbb{P}_s(w_1, y)$	$S_c(w_1,\ w_2 \dots w_L w_{L+1})$
$S_c(v_1 \dots v_{k-1},\ w_1 w_2 \dots w_L)$	w_{L+1}	y	$\mathbb{P}^*_n(k-1, L) \cdot \mathbb{P}_s(w_1, y)$	$S_c(v_1 \dots v_{k-1} w_1,\ w_2 \dots w_L w_{L+1})$
$S_c(v_1 \dots v_{L-1},\ w_1 w_2 \dots w_L)$	w_{L+1}	y	$\mathbb{P}^*_n(L-1, L) \cdot \mathbb{P}_s(w_1, y)$	$S_d(v_1 \dots v_{L-1} w_1,\ w_2 \dots w_L w_{L+1})$
$S_d(v_1 v_2 \dots v_L,\ w_1 w_2 \dots w_L)$	w_{L+1}	y	$\mathbb{P}^*_n(L, L) \cdot \mathbb{P}_s(w_1, y)$	$S_d(v_2 \dots v_L w_1,\ w_2 \dots w_L w_{L+1})$
$S_d(v_1 v_2 \dots v_L,\ w_1 w_2 \dots w_L)$	ϵ	y	$\mathbb{P}^*_n(L, L) \cdot \mathbb{P}_s(w_1, y)$	$S_e(v_2 \dots v_L w_1,\ w_2 \dots w_L w_{L+1})$
$S_e(v_1 v_2 \dots v_L,\ w_1 w_2 \dots w_k)$	ϵ	y	$\mathbb{P}^*_n(L, k) \cdot \mathbb{P}_s(w_1, y)$	$S_e(v_2 \dots v_L w_1,\ w_2 \dots w_{k-1})$
$S_e(v_1 \dots v_L,\ w_1)$	ϵ	y	$\mathbb{P}^*_n(L, 1) \cdot \mathbb{P}_s(w_1, y)$	$S_f(\epsilon,\ \epsilon)$

The other transitions can be deduced using the following rules:

For every state of the form...

$S_\alpha(\bar{v},\bar{w})$	with $\bar{v} = v_1 \dots v_j,\ 0 \le j \le L$ and $\bar{w} = w_1 \dots w_k,\ 0 \le k \le L$

...there are transitions of the form...

p	ℓ_i	ℓ_o	$\mathbb{W}$	q	
$S_\alpha(\bar{v},\bar{w})$	ϵ	ϵ	$\mathbb{P}_t\mathbb{P}_l(i)$	$T_\alpha^{(i)}(\bar{v},\bar{w})$	$\forall 1 \le i \le j$
$S_\alpha(\bar{v},\bar{w})$	ϵ	ϵ	$\mathbb{P}_f$	$F_\alpha^{(1)}(\bar{v},\bar{w})$	
$S_\alpha(\bar{v},\bar{w})$	ϵ	ϵ	$\mathbb{P}_r\mathbb{P}_l(i)$	$R_\alpha^{(i)}(\bar{v},\bar{w})$	$\forall 1 \le i \le k$
$T_\alpha^{(i)}(\bar{v},\bar{w})$	ϵ	y	$\mathbb{P}_s(v_{j+1-i}, y)$	$T_\alpha^{(i-1)}(\bar{v},\bar{w})$	$\forall 1 < i \le j$
$T_\alpha^{(1)}(\bar{v},\bar{w})$	ϵ	y	$\mathbb{P}_s(v_j, y)$	$S_\alpha(\bar{v},\bar{w})$	
$F_\alpha^{(i)}(\bar{v},\bar{w})$	ϵ	y	$\mathbb{P}_s(v_{j+1-i}, \tilde{y})$	$F_\alpha^{(i+1)}(\bar{v},\bar{w})$	$\forall 1 \le i < j$
$F_\alpha^{(1)}(\bar{v},\bar{w})$	ϵ	y	$\mathbb{P}_l(i)\mathbb{P}_s(v_{j+1-i}, \tilde{y})$	$S_\alpha(\bar{v},\bar{w})$	$\forall 1 \le i \le j$
$R_\alpha^{(i)}(\bar{v},\bar{w})$	ϵ	y	$\mathbb{P}_s(w_i, \tilde{y})$	$R_\alpha^{(i-1)}(\bar{v},\bar{w})$	$\forall 1 < i \le k$
$R_\alpha^{(1)}(\bar{v},\bar{w})$	ϵ	y	$\mathbb{P}_s(w_1, \tilde{y})$	$S_\alpha(\bar{v},\bar{w})$	

For every transition of the form...

p	ℓ_i	ℓ_o	$\mathbb{W}$	q	
$S_\alpha(\bar{v},\bar{w})$	x	y	$\mathbb{P}_n^*(\dots)\mathbb{P}_s(x,y)$	$S_\beta(\bar{v}',\bar{w}')$	with $\bar{v},\bar{v}',\bar{w},\bar{w}' \in \Omega^*$ and $x, y \in \Omega$

...there are also transitions of the form...

p	ℓ_i	ℓ_o	$\mathbb{W}$	q	
$S_\alpha(\bar{v},\bar{w})$	x	ϵ	$\mathbb{P}_d$	$D_\beta(\bar{v}',\bar{w}')$	if $\beta \neq f$
$S_\alpha(\bar{v},\bar{w})$	x	ϵ	$\mathbb{P}_d$	$S_f(\epsilon,\epsilon)$	if $\beta = f$
$D_\alpha(\bar{v},\bar{w})$	x	ϵ	$\mathbb{P}_x$	$D_\beta(\bar{v}',\bar{w}')$	if $\alpha \neq b$
$D_\alpha(\bar{v},\bar{w})$	ϵ	ϵ	$\mathbb{P}_e$	$S_\alpha(\bar{v}',\bar{w}')$	if $\alpha \neq b$

The probability parameters are $\mathbb{P}_d$ to open a deletion, $\mathbb{P}_x$ to extend a deletion, $\mathbb{P}_e$ to end a deletion, $\mathbb{P}_t$ for a tandem duplication, $\mathbb{P}_f$ for a forward inverse duplication, $\mathbb{P}_r$ for a reverse inverse duplication, $\mathbb{P}_n$ for no gap, $\mathbb{P}_i$ for a transition substitution (A↔G or C↔T), $\mathbb{P}_v$ for a transversion substitution (A↔C, A↔T, C↔G, G↔T), $\mathbb{P}_m$ for no substitution, and $\mathbb{P}_l(k)$ for the probability that a duplication has length k. The constraints on these parameters are $\mathbb{P}_d + \mathbb{P}_t + \mathbb{P}_f + \mathbb{P}_r + \mathbb{P}_n = 1$, $\mathbb{P}_x + \mathbb{P}_e = 1$, $\mathbb{P}_i + \mathbb{P}_v + \mathbb{P}_m = 1$, and $\sum_{k=1}^{\mathcal{K}} \mathbb{P}_l(k) = 1$.

The substitution matrix is $\mathbb{P}_s(x, y)$, defined to be $\mathbb{P}_m$ if $x = y$ (no substitution), $\mathbb{P}_i$ if $x = \hat{y}$ (transition) and $\mathbb{P}_v/2$ otherwise (transversion).

The context-adjusted probability of not opening a gap at a site with j nucleotides of past context and k nucleotides of future context is

$$\mathbb{P}_n^*(j,k) = \mathbb{P}_n + (\mathbb{P}_t + \mathbb{P}_f) \sum_{i=j+1}^{\mathcal{K}} \mathbb{P}_l(i) + \mathbb{P}_r \sum_{i=k+1}^{\mathcal{K}} \mathbb{P}_l(i)$$

The probability parameters can be estimated directly from trusted alignments using the Baum-Welch algorithm [37].

To keep our implementation simple we used a basic error model. It would be straightforward to give the error model a richer parameterization; for example, allowing more free parameters in the substitution matrix, or allowing the mutation parameters to depend on the adjacent context.

The transducer T_{error} described in this section models errors but still assumes observation of the full-length sequence. By composing it with a 3-state transducer $L \to M \to R$ whose initial and final states (L, R) erase their input and whose middle state (M) echoes its input unmodified, we can model both errors and partial observation.

3.5 Codes to correct insertion, deletion and substitution errors

In Section 3.3 we described a code that attempted to pack as much encoded information as possible into the mixed-radix sequence. However, we can also use the extra information in the ternary and quaternary bits for error-correction purposes: either to store parity bits, or to store a watermark.

The watermark was introduced by Davey and MacKay as an "inner code" to correct for insertion and deletion errors in combination with an "outer code", such as a low-density parity check (LDPC) code, to correct substitution errors [15, 16]. The watermark is a pilot sequence, known to encoder and decoder, which is XOR'd with a sparsified transform of the input signal (so that some or most of the watermark is transmitted unmodified). Deletions may corrupt a few bits of the input signal, but the extent of the deletion can be identified with reference to the watermark. Watermark codes stand in contrast to "marker" codes, another strategy for insertion-deletion correction which explicitly flags the boundaries of blocks [17].

Using transducers we can readily introduce both watermark- and marker-style synchronization in several ways: (i) as a marker approach that uses the "reserved words" developed in Section 3.2, we can construct an M-state cyclic transducer that introduces a reserved word into the output DNA after every M input bits; (ii) as a watermark approach that uses the spare information in the higher-radix digits, we can give each of the M states a unique pseudorandom signature for radix conversion, so that every (binary) input digits is mapped uniquely to a single (binary, ternary or quaternary) output digit, with this mapping performed in a pseudorandom way by each of the M states; (iii) for additional synchronization, we can use an approach closer to Davey and MacKay's original watermark technique and have the transducer interleave input bits with watermark bits at some ratio.

When using synchronization codes, we first encode the message using an LDPC code, which we then apply to the decoded sequence at the other end. The block length of the LDPC code can be matched to the block length of the synchronization code (i.e. the length of the watermark, or the spacing between markers) to facilitate decoding of message fragments obtained via DNA sequencing.

3.6 Combining the component transducers

We have described several component transducers so far, including machines T_{radix}, for converting a sequence of binary symbols into a mixed-radix sequence (Section 3.3); T_{code}, for converting the mixed-radix sequence into nonrepeating DNA (Section 3.2); and T_{error} for modeling sequencing errors during decoding (Section 3.4). Other relevant machines are depicted in the figures, such as T_{1b} for implementing the Hamming(7,4) error-correction code (Figure 1(b)).

These machines can be combined by the process of transducer composition (Section 3.1.2) into a single integrated automaton suitable for coding. For example, $T_{1b} + T_{\text{radix}} + T_{\text{code}} + T_{\text{error}}$ implements all four steps described in the previous paragraph.

The transducer composition procedure implicitly resolves the mixed-radix conversion. Specifically, T_{radix} converts a binary input into all possible mixed-radix sequences, but T_{code} will only accept one series of radices (the radix at each point in the sequence depending on the most recent $\mathcal{K}$ nucleotides that have been emitted previously, and thus, the number of nucleotides available at the next variable position without generating a tandem or inverted repeat). Thus, only one of the possible output sequences of T_{radix} is valid. The selection of this valid output happens in the transducer composition $T_{\text{radix}} + T_{\text{code}}$ when we match transitions from T_{radix} with those of T_{code}; only transitions with compatible radices are used by this procedure.

The matching of transitions and states that occurs during transducer composition implcitly discards large parts of the composite state space that are unreachable. We can use this to particular advantage when combining T_{error} with T_{delay} (the version of T_{code} described in Section 3.2.1 that has both past and future context). States in T_{error} will only mesh with states in T_{delay} that have compatible contexts, so we can efficiently throw out many of the possible combinations of states.

While some error-correcting codes, such as short Hamming codes, can be conveniently represented as state machines with exhaustively enumerable state spaces, more sophisticated codes—such as LDPC codes—generally have too many possibilities to enumerate efficiently. Instead, LDPC codes are generally decoded using sum-product belief propagation on cyclic graphical models [22, 20]. Following Davey and MacKay [15], we have here implemented decoding as a two-stage process wherein the HMM corrects indel errors and the LDPC code corrects any remaining substitution errors. Unlike Davey and MacKay, we do not currently make use of “soft” information from the first decoding stage. Since LDPC codes and HMMs can both be represented as factor graphs of (respectively) trellis and cyclic topology [44], these could in principle offer a common likelihood framework in which uncertainty from the HMM decoding could inform the second stage of LDPC bit-correction.

4 Results

We have developed open-source computer programs implementing the algorithms in this paper. MIXRADAR implements the MIXed-RADix ARithmetic code of Section 3.3. DNASTORE implements the non-repeating DNA code of Section 3.2, a subset of the error-correcting code of Section 3.6, and the transducer composition algorithm from Section 3.1.2. WATMARK generates transducers that implement the watermark synchronization of Section 3.5.

All these programs are available at `https://github.com/ihh/dnastore` along with auxiliary codes (for example the Hamming(7,4) code of Figure 1(b)) and scripts to generate those codes.

4.1 MIXRADAR

Table 1 shows statistics for codes generated with MIXRADAR. It is straightforward to achieve (near-)ideal performance in converting bits to quats, but to approach ideal performance in bit-to-trit conversion requires a block code. Finding the optimal block length is equivalent to finding positive integers m, n for which 3^m is only a little more than 2^n; the number of bits/trit is then n/m. For example, good values of (m, n) are $(7, 11)$ (since $3^7 \simeq 2^{11} \times 1.07$) and $(12, 19)$ (since $3^{12} \simeq 2^{19} \times 1.01$). These offer bit/trit ratios of $11/7 \simeq 1.571$ and $19/12 \simeq 1.583$, respectively (compared to the theoretical maximum $\log_2(3) \simeq 1.585$ bits/trit). However, implementing codewords of 11 or 19 bits in length using a state machine would require millions or billions of states to encode all the possible mixed-radix outputs. The best we can achieve at lower codeword lengths is $(m, n) = (4, 6)$ (where $3^4 \simeq 2^6 \times 1.27$). This codeword length (6 bits) gives a performance that is near-ideal for radix-2 output (1 bit/base) and radix-4 output (2 bits/base) and decent for radix-3 output (1.5 bits/base), at 2125 states and 6375 transitions (see Table 1).

4.2 DNASTORE

In the absence of any other specified coding scheme, DNASTORE uses the naive binary-to-ternary conversion of Figure 2 and adds a zero padding bit when the machine needs to be flushed, in contrast to the MIXRADAR block codes that do not require padding bits. This can cause spurious single-bit insertion errors at the end of the message, and more serious problems when the machine is composed with other coding schemes. To circumvent such problems, and for general flexibility, DNASTORE can be supplied with alternative coding machines described in a compact JSON format and can compose multiple machines using the algorithm outlined in Section 3.1.2. This allows the straightforward incorporation of composite state machines incorporating other stages (e.g. more sophisticated radix conversion, or error correcting codes).

Table 2 shows statistics for codes generated with DNASTORE using the algorithm of Section 3.2. For most of the simulations and composite codes in this paper, we have used a short code of $\mathcal{K} = 4$ nucleotides, which avoids single-base and dinucleotide repeats, but not trinucleotide (or longer) repeats.

4.3 Error correction

DNASTORE implements error correction using a simplified version of the transducer T_{error} of Section 3.4, modeling only tandem duplications and not inverted duplications. It can also estimate the parameters of the error model from training data (in the form of alignments of sequenced DNA to known reference copies of that DNA).

To test the performance of these codes, we tried encoding and decoding a uniformly random sequence of 8,192 bits while simulating the controlled corruption of the sequenced DNA by various types of random mutation: point substitutions, non-overlapping deletions of up to 4 bases, and tandem duplications of up to 4 bases. We explored several different composite codes, with components including the length-4 and length-8 codes from Table 2; the mixed-radix block code of 2-bit codewords from Table 1; the Hamming(7,4) error-correction code of Figure 1(b); the synchronization codes of Section 3.5; and an outer LDPC(16384,8192) code implemented using Neal's LDPC-codes software available from `https://github.com/radfordneal/LDPC-codes`. We varied substitution, deletion and insertion probabilities, using the Baum-Welch algorithm to estimate the probability parameters from data, repeating the experiment 12 times at each setting, and measuring the Levenshtein edit distance (normalized by the number of input bits) between the original and recovered bit-sequences.

The results of these computational experiments are summarized in Figures 6 to 9. Figure 6 shows the decoder accuracy in the presence of simulated tandem duplication events. Since the code (by construction) contains no tandem duplications of $\mathcal{K}$ or fewer bases, the decoder (which models duplications as special types of error) should be able to recover from these perfectly, and this is indeed the case—as long as the maximum duplicated sequence length is less than the context length of the error decoder (lower curve of Figure 6). When the duplication size exceeds the error model's context length, the error model is unable to recognize them and duplications then significantly affect decoder accuracy (upper curve of Figure 6).

Figure 7 shows the performance of various codes in the presence of single-nucleotide substitution errors scattered randomly throughout the sequence. At very low substitution rates, the basic DNASTORE code (without any additional error-correction) is still able to correct some errors, probably because those substitution errors introduce illegal sequences. Composing a MIXRADAR code in front of the DNASTORE further improves the mitigation of substitution errors, but both DNASTORE and MIXRADAR+DNASTORE codes quickly start to exhibit decoding errors as the substitution rate rises. This can be contrasted with two explicitly error-correcting codes, based respectively on the Hamming(7,4) code (which due to the low number of states can be represented as a state machine and incorporated directly into the Viterbi decoder) and a watermark code combined with an LDPC code (the LDPC encoding and decoding take place independently of the main state machine). These codes are signficantly more robust to substitution errors, maintaining a zero decoded error rate up to around 0.2 substitutions per nucleotide. The Hamming code performs slightly better than the LDPC code at the highest substitution rates, which may be due to a combination of reasons: (i) the Hamming code is implemented as a transducer (so error correction can be performed as part of Viterbi decoding, and is therefore exact), (ii) this simulation did not include any "burst" style errors, which would impact Hamming code performance more than LDPC, (iii) the LDPC code starts to deteriorate significantly when it misses block boundaries (and, again, the two-stage decoding protocol we used does not allow the LDPC decoder to influence the Viterbi decoding, so any block boundary errors that the Viterbi decoder makes are final).

In Figure 8, we investigated the same codes under simulated deletions. Several points are notable from this plot. First, deletions have a much bigger impact than substitutions on all codes (each simulated deletion event removes from one to four bases, so the average number of deleted bases is 2.5 times the x-axis labels on this plot; still, even accounting for this, the impact of deletions is worse than the substitutions of Figure 7 and more comparable to the duplications of Figure 6). Second, unlike with substitutions, the Hamming code accuracy deteriorates even at low error rates—which is to be expected, since deletions introduce burst errors that Hamming cannot correct. Third, the watermark-based code does work as advertised to correct errors with 100% accuracy—but only at low deletion rates. When the deletion rate exceeds 0.02 (meaning, with our average deletion size of 2.5 bases, that roughly 5% of the bases are deleted) then the error rate starts creeping up. And fourth, for all the codes tried, when the deletion rate exceeds around 0.13 (so roughly 1/3 of bases are deleted) no signal is recoverable at all.

To further explore the optimal design of watermark codes, we repeated the deletion experiment while varying the number of watermark bits and the length of the watermark signal (Figure 9). Empirically, it appears from this preliminary experiment that the single most influential factor on code accuracy is the correct recovery of LDPC block boundaries, since missed boundaries cause large chunks of the signal to be dropped, and if not dropped then they can still be impossible to error-correct if the block alignment is wrong. The single factor that seems most important for correct block boundary identification and alignment is the ratio of signal bits to watermark bits: the sparser the signal relative to the watermark, the better chance the decoder has of recovering from large and/or frequent deletions.

5 Discussion

We have outlined general design principles for efficiently encoding signals in DNA by converting a binary sequence to a mixed-radix sequence and then a nonrepeating nucleotide sequence. A distinct feature of our approach is its modularity, which allows integrated simultaneous decoding of multi-layer codes that are representable as finite-state transducers. This allows for straightforward combination and refinement of component automata for interleaving signals with watermarks, inserting markers or metadata, adding parity bits, converting binary to mixed-radix, generating repeat-free DNA, modeling errors, or other tasks.

An important aspect of DNA storage which we have not explicitly addressed is the higher-level organization of the DNA "filesystem", including aspects such as block or packet structure, redundancy, addressing, versioning, and metadata. This has been explored in work by others [6, 14]. The code described here operates at a lower level. However, our allowance for reserved "control words" can be used to embed in-band metadata that may be useful in organizing files into manageably-sized but readily-reassembled blocks or packets. Given that assembly of shorter DNA synthesis into longer contiguous sequences is currently one of the more expensive and tricky parts of DNA synthesis [45], and indeed one of the more algorithmically challenging steps of DNA sequencing [31], and given the difficulties we observe in maintaining code synchronization over longer sequences, the optimal structure for a DNA filesystem may be one in which a large file is broken into many short fragments, each individually addressed, which may be reassembled by the decoder, somewhat analogously to IP packets. In this case nucleic acid control words flanking encoded packet addresses could theoretically be recognized and selected by molecular hybridization.

The decoding approach that we have used (Viterbi decoding, followed by decoding of any outer codes such as LDPC) is rather simple. It does not use the posterior probability information available from the HMM, or allow any other cross-talk between the (cyclic) LDPC factor graph and the (trellis) HMM graph. It also does not incorporate multiple reads from the same DNA sequence, and it is subject to computational resource constraints due to the number of states and transitions in the composite transducer, which grow quickly when multiple coding layers are composed on top of one another. These are all, in principle, features that could be addressed within the probabilistic framework by approximate or stochastic inference algorithms, such as loopy belief propagation or Markov Chain Monte Carlo.

6 References

References

[1] N. Goldman, P. Bertone, S. Chen, C. Dessimoz, E. M. LeProust, B. Sipos, and E. Birney. Towards Practical, High-Capacity, Low-Maintenance Information Storage in Synthesized DNA. *Nature*, 494(7435):77–80, Feb 2013.

[2] R. E. Green, J. Krause, A. W. Briggs, T. Maricic, U. Stenzel, M. Kircher, N. Patterson, H. Li, W. Zhai, M. H. Fritz, N. F. Hansen, E. Y. Durand, A. S. Malaspinas, J. D. Jensen, T. Marques-Bonet, C. Alkan, K. Prufer, M. Meyer, H. A. Burbano, J. M. Good, R. Schultz, A. Aximu-Petri, A. Butthof, B. Hober, B. Hoffner, M. Siegemund, A. Weihmann, C. Nusbaum, E. S. Lander, C. Russ, N. Novod, J. Affourtit, M. Egholm, C. Verna, P. Rudan, D. Brajkovic, Z. Kucan, I. Gusic, V. B. Doronichev, L. V. Golovanova, C. Lalueza-Fox, M. de la Rasilla, J. Fortea, A. Rosas, R. W. Schmitz, P. L. Johnson, E. E. Eichler, D. Falush, E. Birney, J. C. Mullikin, M. Slatkin, R. Nielsen, J. Kelso, M. Lachmann, D. Reich, and S. Paabo. A Draft Sequence of the Neandertal Genome. *Science*, 328(5979):710–722, May 2010.

[3] G. M. Church, Y. Gao, and S. Kosuri. Next-Generation Digital Information Storage in DNA. *Science*, 337(6102):1628, Sep 2012.

[4] O. Milenkovic and N. Kashyap. DNA codes that avoid secondary structures. August 2005. `http://arxiv.org/abs/cs.DM/0508055`.

[5] N. Yachie, Y. Ohashi, and M. Tomita. Stabilizing synthetic data in the DNA of living organisms. *Syst Synth Biol*, 2(1-2):19–25, Jun 2008.

[6] S. M. Yazdi, Y. Yuan, J. Ma, H. Zhao, and O. Milenkovic. A Rewritable, Random-Access DNA-Based Storage System. *Scientific Reports*, 5:14138, 2015.

[7] V. Dhameliya, D. Limbachiya, M. Khakhar, and M. K. Gupta. On optimal family of codes for DNA storage. *Computing Research Repository*, abs/1501.07133, 2015. `http://arxiv.org/abs/1501.07133`.

[8] H. M. Kiah, G. J. Puleo, and O. Milenkovic. Codes for DNA storage channels. *Computing Research Repository*, abs/1410.8837, 2014. `http://arxiv.org/abs/1410.8837`.

[9] S. M. Yazdi, H. M. Kiah, E. R. Garcia, J. Ma, H. Zhao, and O. Milenkovic. DNA-based storage: Trends and methods. *Computing Research Repository*, abs/1507.01611, 2015. `http://arxiv.org/abs/1507.01611`.

[10] R. Gabrys, H. M. Kiah, and O. Milenkovic. Asymmetric Lee distance codes for DNA-based storage. *Computing Research Repository*, abs/1506.00740, 2015. `http://arxiv.org/abs/1506.00740`.

[11] F. H. Hunt, S. Perkins, and D. H. Smith. Channel models and error correction codes for DNA information storage. *International Journal of Information and Coding Theory*, 3(2):120–136, October 2015.

[12] S. Jain, F. Farnoud, M. Schwartz, and J. Bruck. Duplication-correcting codes for data storage in the DNA of living organisms. arXiv, 2016. 1606.00397.

[13] R. Gabrys, E. Yaakobi, and O. Milenkovic. Codes in the damerau distance for DNA storage. *Computing Research Repository*, abs/1601.06885, 2016. `http://arxiv.org/abs/1601.06885`.

[14] J. Bornholt, R. Lopez, D. M. Carmean, L. Ceze, G. Seelig, and K. Strauss. A DNA-based archival storage system. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '16, pages 637–649, New York, NY, USA, 2016. ACM. `http://doi.acm.org/10.1145/2872362.2872397`.

[15] M. C. Davey and D. J. C. MacKay. Watermark codes: Reliable communication over insertion/deletion channels. In *In ISIT 2000*, pages 47–7, 2000.

[16] M. C. Davey and D. J. C. MacKay. Reliable communication over channels with insertions, ddeletions, and substitutions. *IEEE Transactions on Information Theory*, 47(2):687–698, 2001.

[17] Edward A. Ratzer and David J.C. MacKay. Codes for channels with insertions, deletions and substitutions. In *In 2nd International Symposium on Turbo Codes and Related Topics*, pages 149–156, 2000.

[18] D. Laehnemann, A. Borkhardt, and A. C. McHardy. Denoising DNA deep sequencing data-high-throughput sequencing errors and their correction. *Briefings in Bioinformatics*, 17(1):154–179, Jan 2016.

[19] D. J. C. Mackay. *Information Theory, Inference and Learning Algorithms*. Cambridge University Press, 1st edition, 2003.

[20] B. Frey and D. MacKay. A revolution: Belief propagation in graphs with cycles. In *In Neural Information Processing Systems*, pages 479–485. MIT Press, 1998.

[21] K. P. Murphy, Y. Weiss, and M. I. Jordan. Loopy belief propagation for approximate inference: An empirical study. In *Proceedings of the Fifteenth Conference on Uncertainty in Artificial Intelligence*, UAI'99, pages 467–475, San Francisco, CA, USA, 1999. Morgan Kaufmann Publishers Inc. `http://dl.acm.org/citation.cfm?id=2073796.2073849`.

[22] D. J. C. Mackay and R. M. Neal. Near Shannon limit performance of low density parity check codes. *Electronics Letters*, 33(6):457–458, 1997. `http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=585036`.

[23] M. Mohri, F. Pereira, and M. Riley. Weighted finite-state transducers in speech recognition. *Computer Speech and Language*, 16(1):69–88, 2002.

[24] *Finite-State Transducer (Wikipedia article)*, Accessed May 19, 2016. `https://en.wikipedia.org/w/index.php?title=Finite_state_transducer&oldid=718569101`.

[25] E. Eskin, W. N. Grundy, and Y. Singer. Protein family classification using sparse Markov transducers. In P. Bourne, M. Gribskov, R. Altman, N. Jensen, D. Hope, T. Lengauer, J. Mitchell, E. Scheeff, C. Smith, S. Strande, and H. Weissig, editors, *Proceedings of the Eighth International Conference on Intelligent Systems for Molecular Biology*, pages 134–135, Menlo Park, CA, 2000. AAAI Press.

[26] B. Paten, J. Herrero, S. Fitzgerald, K. Beal, P. Flicek, I. Holmes, and E. Birney. Genome-Wide Nucleotide-Level Mammalian Ancestor Reconstruction. *Genome Research*, 18:1829–1843, Nov 2008.

[27] O. Westesson, G. Lunter, B. Paten, and I. Holmes. Phylogenetic automata, pruning, and multiple alignment. arXiv, 2012. 1202.4026.

[28] O. Westesson, G. Lunter, B. Paten, and I. Holmes. Accurate Reconstruction of Insertion-Deletion Histories by Statistical Phylogenetics. *PLoS ONE*, 7(4):e34572, 2012.

[29] R. F. Schwarz, A. Trinh, B. Sipos, J. D. Brenton, N. Goldman, and F. Markowetz. Phylogenetic Quantification of Intra-Tumour Heterogeneity. *PLoS Computational Biology*, 10(4):e1003535, Apr 2014.

[30] N. G. de Bruijn. A combinatorial problem. *Koninklijke Nederlandsche Akademie Van Wetenschappen*, 49(6):758–764, June 1946.

[31] P. A. Pevzner, H. Tang, and M. S. Waterman. An Eulerian Path Approach to DNA Fragment Assembly. *Proceedings of the National Academy of Sciences of the United States of America*, 98(17):9748–9753, Aug 2001.

[32] D. R. Zerbino and E. Birney. Velvet: Algorithms for De Novo Short Read Assembly Using de Bruijn Graphs. *Genome Research*, 18(5):821–829, May 2008.

[33] Z. Iqbal, M. Caccamo, I. Turner, P. Flicek, and G. McVean. De Novo Assembly and Genotyping of Variants Using Colored De Bruijn Graphs. *Nature Genetics*, 44(2):226–232, Feb 2012.

[34] C. Burge and S. Karlin. Prediction of Complete Gene Structures in Human Genomic DNA. *Journal of Molecular Biology*, 268(1):78–94, Apr 1997.

[35] S. L. Salzberg, M. Pertea, A. L. Delcher, M. J. Gardner, and H. Tettelin. Interpolated Markov Models for Eukaryotic Gene Finding. *Genomics*, 59(1):24–31, Jul 1999.

[36] J. J. Rissanen. Generalized Kraft inequality and arithmetic coding. *IBM Journal of Research and Development*, 20(3):198–203, May 1976. http://dx.doi.org/10.1147/rd.203.0198.

[37] R. Durbin, S. Eddy, A. Krogh, and G. Mitchison. *Biological Sequence Analysis: Probabilistic Models of Proteins and Nucleic Acids*. Cambridge University Press, Cambridge, UK, 1998.

[38] F. C. N. Pereira and M. Riley. Speech recognition by composition of weighted finite automata. *Computing Research Repository*, cmp-lg/9603001, 1996. http://arxiv.org/abs/cmp-lg/9603001.

[39] I. Holmes. Using Guide Trees to Construct Multiple-Sequence Evolutionary HMMs. *Bioinformatics*, 19 Suppl 1:i147–157, 2003.

[40] I. Holmes. Phylocomposer and Phylodirector: Analysis and Visualization of Transducer Indel Models. *Bioinformatics*, 23(23):3263–3264, Dec 2007.

[41] H. Ellegren. Microsatellites: simple sequences with complex evolution. *Nature Reviews Genetics*, 5(6):435–445, Jun 2004.

[42] K.-C. Chu and J. Gill. Computer science. chapter Mixed-Radix Huffman Codes, pages 209–218. Plenum Press, New York, NY, USA, 1992. http://dl.acm.org/citation.cfm?id=166961.167006.

[43] M. Golin, X. Xu, and J. Yu. A generic top-down dynamic-programming approach to prefix-free coding. In *Proceedings of the Twentieth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '09, pages 758–767, Philadelphia, PA, USA, 2009. Society for Industrial and Applied Mathematics. http://dl.acm.org/citation.cfm?id=1496770.1496853.

[44] F. R. Kschischang, B. J. Frey, and H. A. Loeliger. Factor graphs and the sum-product algorithm. *IEEE Transactions on Information Theory*, 47(2):498–519, September 2006. http://dx.doi.org/10.1109/18.910572.

[45] D. G. Gibson, L. Young, R. Y. Chuang, J. C. Venter, C. A. Hutchison, and H. O. Smith. Enzymatic assembly of DNA molecules up to several hundred kilobases. *Nature Methods*, 6(5):343–345, May 2009.

7 Tables

				Mean output symbols/input bit		
N	ν	States	Transitions	Radix 2	Radix 3	Radix 4
2	0.01	32	96	1.125	0.875	0.625
3	0.01	102	306	1.042	0.6667	0.6667
4	0.001	287	861	1.016	0.7344	0.5156
5	0.001	865	2595	1.006	0.6687	0.6
6	0.001	2125	6375	1.003	0.6667	0.5026
		Theoretical minima:		$\log_2(2) = 1$	$\log_3(2) \simeq 0.631$	$\log_4(2) = 0.5$

Table 1: Statistics for several code transducers generated using MIXRADAR (Section 4.1). N is the block length in bits; ν is the probability of the end-of-block symbol (signifying premature termination of the codeword) appearing at any position.

$\mathcal{K}$	L_{invrep}	N_c	Repeats?	States	Transitions	Bases/bit	Control words	Notes
2		0	yes	50	96	0.5		Figure 4(a)
2		1	yes	41	77	0.5607	`TT` (start)	Figure 4(b)
2		1	yes	47	83	0.5607	`TT` (start, end)	Figure 4(c)
2		0	no	26	48	0.6667		Figure 4(d)
2		1	no	19	33	0.8298	`TG` (start)	Figure 4(e)
2		1	no	26	40	0.8298	`TG` (start, end)	Figure 4(f)
4		0	no	130	256	0.8055		
4		2	no	208	322	0.8377	`TGTC` (start), `CTGT` (end)	
6		0	no	698	1384	0.8374		
6		4	no	2033	2692	0.8597	`TGTCTG` (start), `ACAGAC` (end), `GCGTAG`, `GTAGCA`	
8		0	no	3802	7512	0.8537		
8		4	no	10772	14456	0.8592	`TGTCTGTA` (start), `GTATCTGT` (end), `CGCTACTC`, `ACGAGCGT`	
10	4	0	no	20346	39976	0.8696		
10	4	4	no	56884	76494	0.8706	`TGTCTGTATG` (start), `ACAGACATAC` (end), `CACGAGTCGT`, `GTGCTCAGCA`	
12	4	0	no	101274	196472	0.9022		
12	4	4	no	298175	393352	0.9028	`TGTCTGTATGTC` (start), `ACAGATGTCTAT` (end), `CAGTGCTGACAG`, `TAGCGTGCGAGC`	

Table 2: Statistics for several code transducers generated using DNASTORE (Section 4.2). $\mathcal{K}$ is the codeword length, L_{invrep} is the minimum length for excluding nonlocal exact inverted repeats (which must be separated by at least 2 nucleotides), and N_c is the number of control words. The first six rows correspond to the transducers shown in Figure 4. In general the expected bases/bit rise with the codeword length $\mathcal{K}$, since more repeats are excluded from the De Bruijn graph.

8 Figures

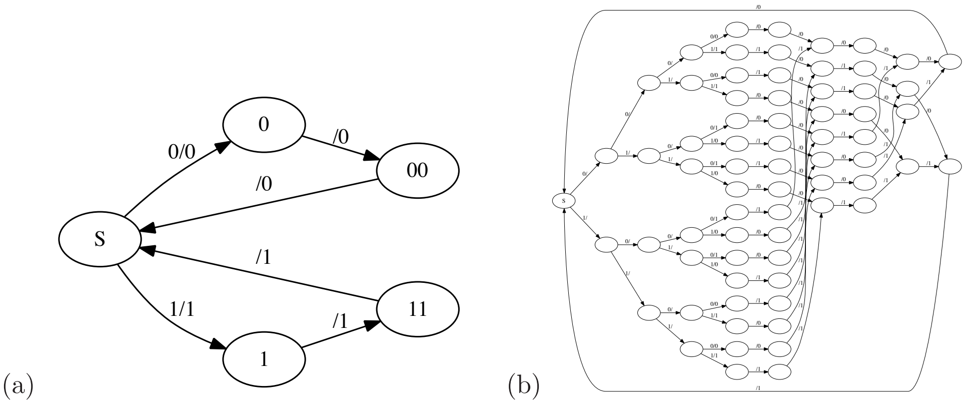

Figure 1: State machines implementing Hamming codes for error correction. Left: Transducer T_{1a} implements the Hamming(3,1) error-correcting code (which simply repeats every bit three times). S is both the initial state and the final state. Right: Transducer T_{1b} implements the Hamming(7,4) error-correcting code, with four data bits and three parity bits. Due to the large number of states, the state names have been omitted from this diagram, as have the ϵ labels for empty inputs or outputs.

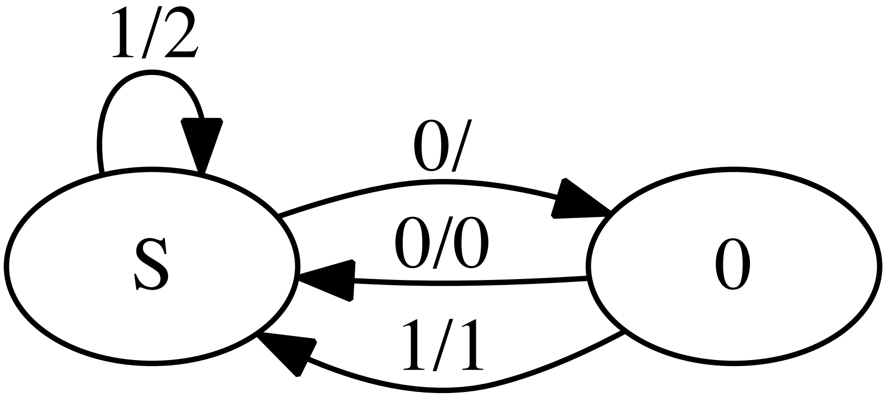

Figure 2: A transducer that converts binary to ternary. This machine maps $0_2 0_2 \to 0_3$, $0_2 1_2 \to 1_3$ and $1_2 \to 2_3$. The problem with this machine is that a dangling zero bit on the input can leave it in a state (0) that needs to be "flushed" with an extra input bit before the machine can finish.

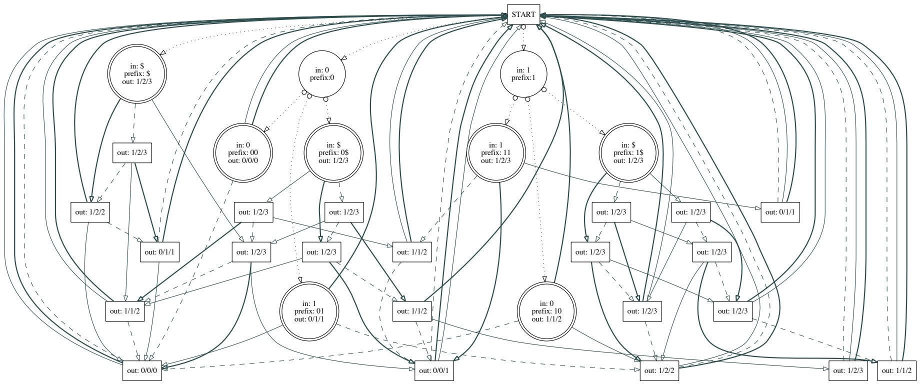

Figure 3: A transducer generated by MIXRADAR (Section 4.1) using Algorithm 1 of Section 3.3 with block length 2 and parameter ν (the probability of a premature end-of-block) equal to 1/100. This transducer converts a binary codeword of length 0, 1 or 2 bits into a sequence of mixed-radix selected by the receiver. The length of the output sequence in general depends on the radices accepted at each position. For example, the input sequence $1_2 1_2$ is converted into the output sequences $1_2 1_2 0_2$, $1_2 1_2 0_3$, $1_2 1_2 1_4$, $1_2 1_3$, $1_2 1_4$, $2_3 0_2$, $2_3 1_3$, $2_3 1_4$, $3_4 0_2$, $3_4 0_3$, or $3_4 1_4$ (requiring three output symbols if the first two symbols are binary, and two otherwise), while the input sequence $0_2 0_2$ is converted into the output sequences $0_2 0_2$, $0_2 0_3$, $0_2 0_4$, 0_3, or 0_4 (requiring two output symbols if the first symbol is binary, and one otherwise). Dotted lines with circles at the source end show transitions that input a bit or an end-of-block symbol ('\$'). Dashed lines show transitions that output a radix-2 digit (i.e. a bit). Solid regular-weight lines show transitions that output a radix-3 digit (i.e. a trit). Solid bold-weight lines show transitions that output a radix-4 digit (i.e. a quat). States are labeled with (as applicable) their input symbol, the input prefix so far, and the choice of output symbols (depending on whether the radix at the next position is 2, 3, or 4). Thus, transitions leaving a state labeled "out: $i/j/k$" output symbols i_2, j_3 and k_4.

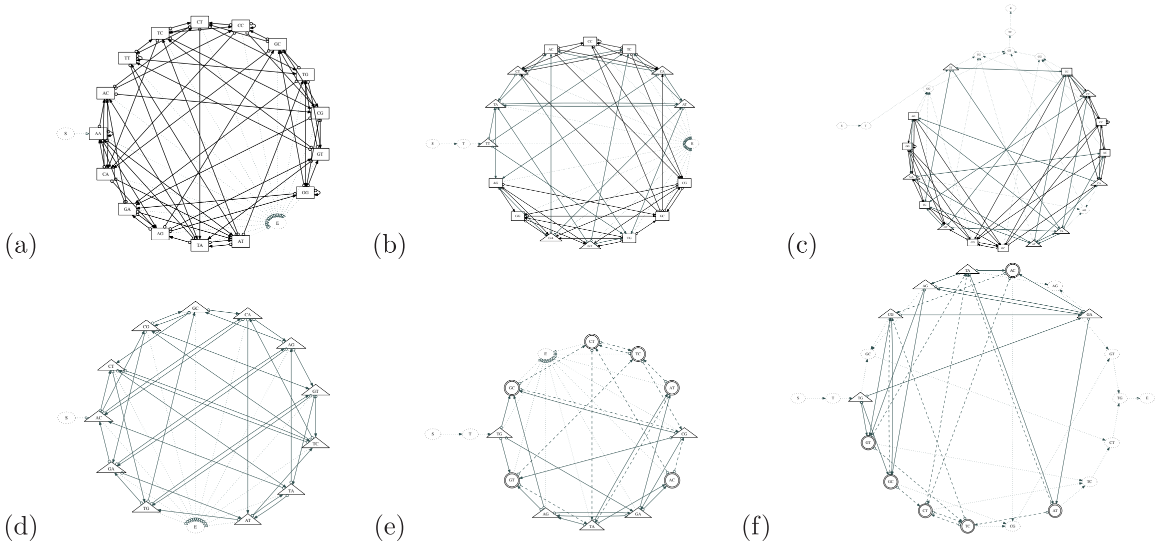

Figure 4: Transducers generated by DNASTORE (Section 4.2) using the method of Section 3.2 with $\mathcal{K} = 2$. These codes are all fundamentally based on the 2-dimensional De Bruijn graph from which vertices are duplicated, deleted and added to arrive at a transition graph with the required properties. Top row (a,b,c): codes in which dinucleotide repeats are allowed. Bottom row (d,e,f): codes in which dinucleotide repeats are prohibited. Left column (a,d): codes in which there are no reserved control words, so the machine start and end in arbitrary states. Central column (b,e): codes in which there is one reserved control word, which only ever appears once, at the start of the encoded DNA sequence. Right-hand column (c,f): codes in which there is one reserved control word, which only ever appears twice: once at the start of the encoded DNA sequence and once at the end. The leftmost machine in the bottom row (d) is similar to the ternary code of [1]. **Key:** Transition label annotations have been omitted from this diagram. Instead, the labels may be deduced from the node and edge shapes, as follows: Solid bold-weight transitions from rectangular states encode quaternary input digits. Solid regular-weight transitions from triangular states encode ternary input digits. Dashed-line transitions from double-circle states encode binary input digits. Dotted-line transitions do not encode input digits; states that can only be exited via these transitions are shown as rectangles. Transitions that encode input digits have empty circles at the source end; transitions that encode output digits have filled arrowheads at the destination end. States are labeled with their past context: the output label of a transition into a state XY is always either Y or ϵ.

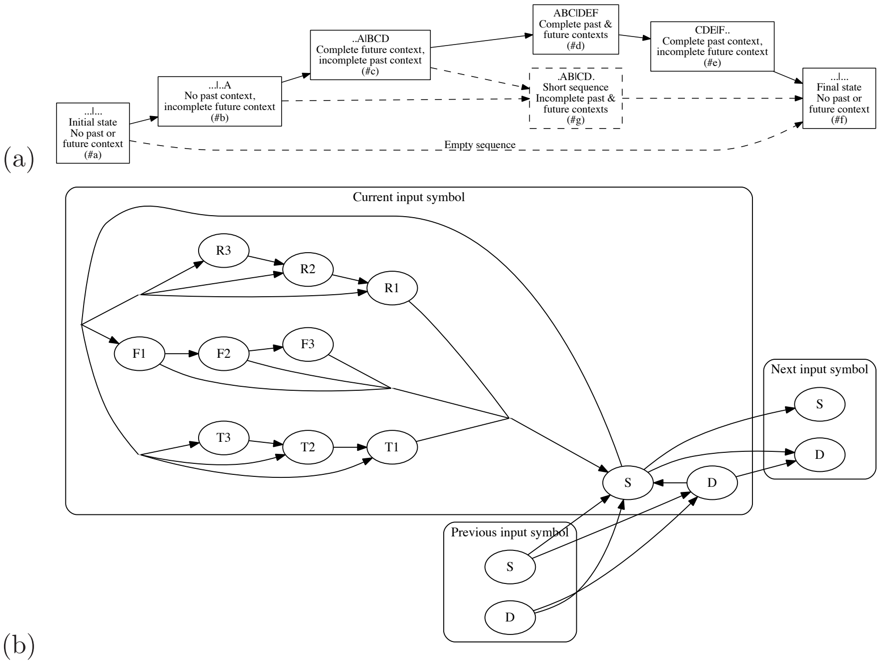

Figure 5: Models of error and partial observation (Section 3.4). Top: Higher-order structure of the error model, showing blocks #a through #f. The dashed lines show the extra block (#g) and transitions that would need to be added to handle sequences shorter than the total (past+future) context length, which require paths that bypass the full loading of both context queues (block #d). Middle: Local neighborhood of a state in the error model for context length $L = 3$. For simplicity, the diagram includes only transitions within block #d, and only transitions to or from states with a given context (representing the "current" input symbol). The D states model deletions; the S states model substitutions; $T1, T2, T3$ model tandem duplications (given past context ACG, insert ACG, yielding the observed mutation $ACG \to ACGACG$); $F1, F2, F3$ model forward inverted duplications (given past context ACG, insert CGT, yielding the observed mutation $ACG \to ACGCGT$); and $R1, R2, R3$ model reverse inverted duplications (given future context ACG, insert CGT, yielding the observed mutation $ACG \to CGTACG$). The probabilities and length distributions of these various events can be modeled.

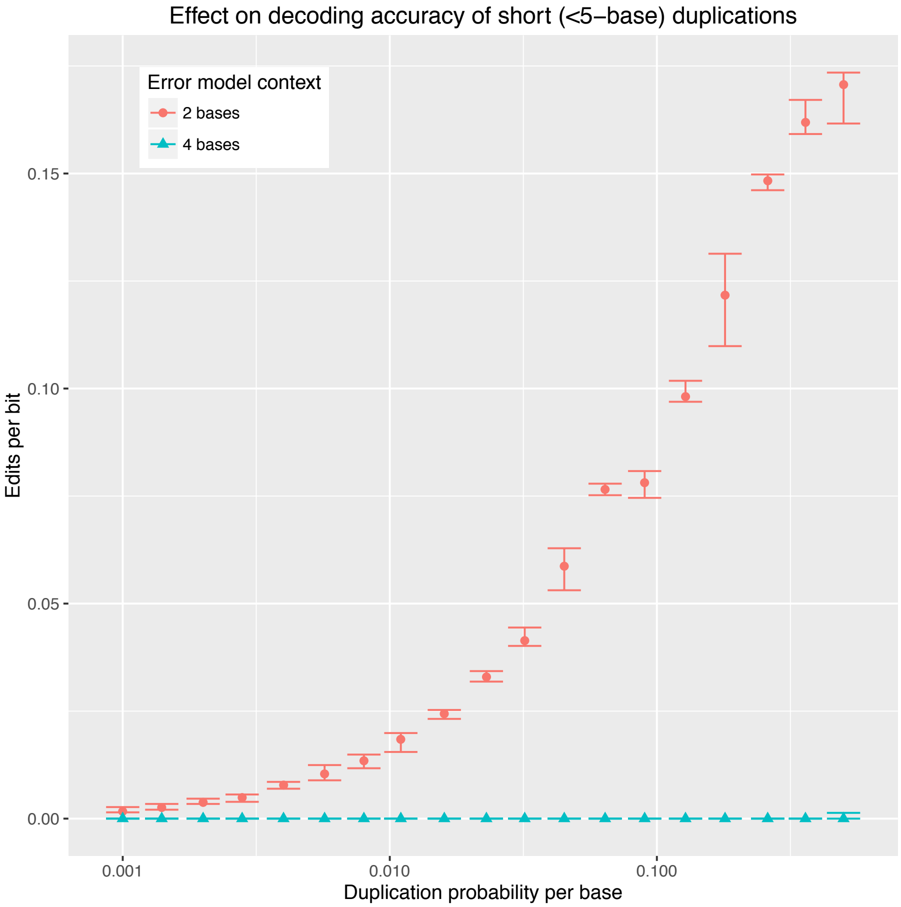

Figure 6: Effect of simulated duplications on decoder accuracy for two different codes with error models of different context length. The codes combine the 2-nucleotide mixed-radix arithmetic code of Figure 3 (MIXRADAR(2)) with the 4- and 8-nucleotide non-repeating codes of Table 2 (DNASTORE(4) and DNASTORE(8)). MIXRADAR(2)+DNASTORE(4) has 2 bases of error-model context; MIXRADAR(2)+DNASTORE(8) has 4 bases of error-model context. The x-axis is the probability of initiating a duplication event at each base; from one to four nucleotides were copied in each random duplication event. Overlapping (nested) duplications were excluded. The y-axis is the Levenshtein edit distance scaled by the length of the input sequence (8192 bits). Median and interquartile range are shown. The decoder with 2 bases of error context is unable to recognize duplications of 3 or 4 nucleotides, and so introduces errors during decoding.

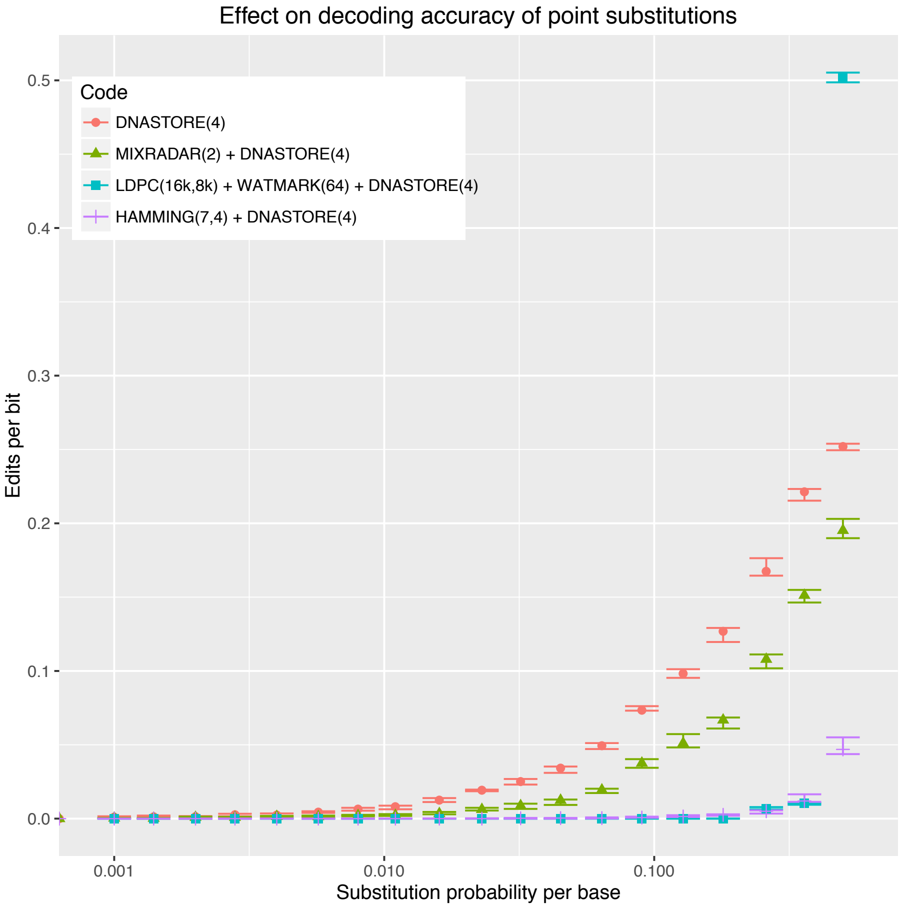

Figure 7: Effect of simulated substitution errors on decoder accuracy for four different codes. The codes all start with the 4-nucleotide non-repeating code of Table 2 (DNASTORE(4)). On top of this are layered the 2-nucleotide mixed-radix arithmetic code of Figure 3 (MIXRADAR(2)), a 2048-bit, 1024-parity bit low-density parity check code combined with a 64-bit watermark code (LDPC(2k,1k)+WATMARK(64)), and the Hamming(7,4) code as implemented in Figure 1(b). The x-axis is the probability of substituting each base, with a transition/transversion ratio of 10. The y-axis is the Levenshtein edit distance scaled by the length of the input sequence (8192 bits). Median and interquartile range are shown.

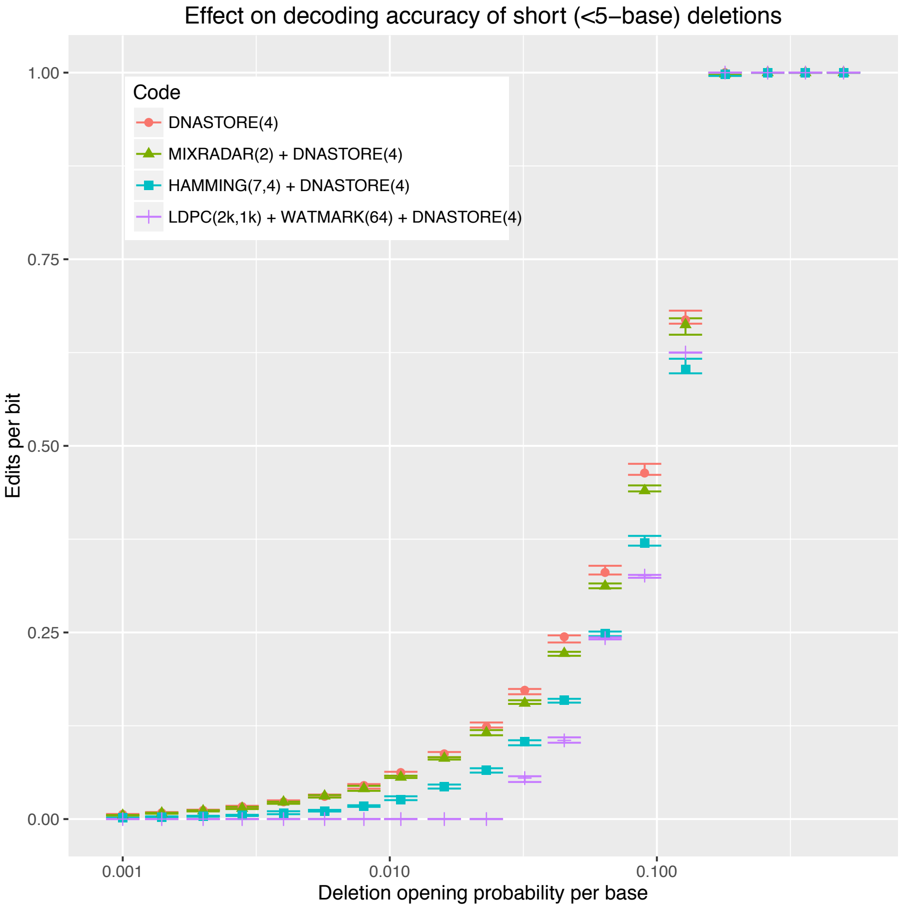

Figure 8: Effect of simulated deletions on decoder accuracy for four different codes. The codes are as described in Figure 7. The x-axis is the probability of initiating a deletion event at each base; a random number of nucleotides (uniformly sampled from one to four) were erased in each deletion event, so that the probability of a nucleotide being deleted is $\sim 2.5\times$ the deletion initiation probability. The y-axis is the Levenshtein edit distance scaled by the length of the input sequence (8192 bits). Median and interquartile range are shown.

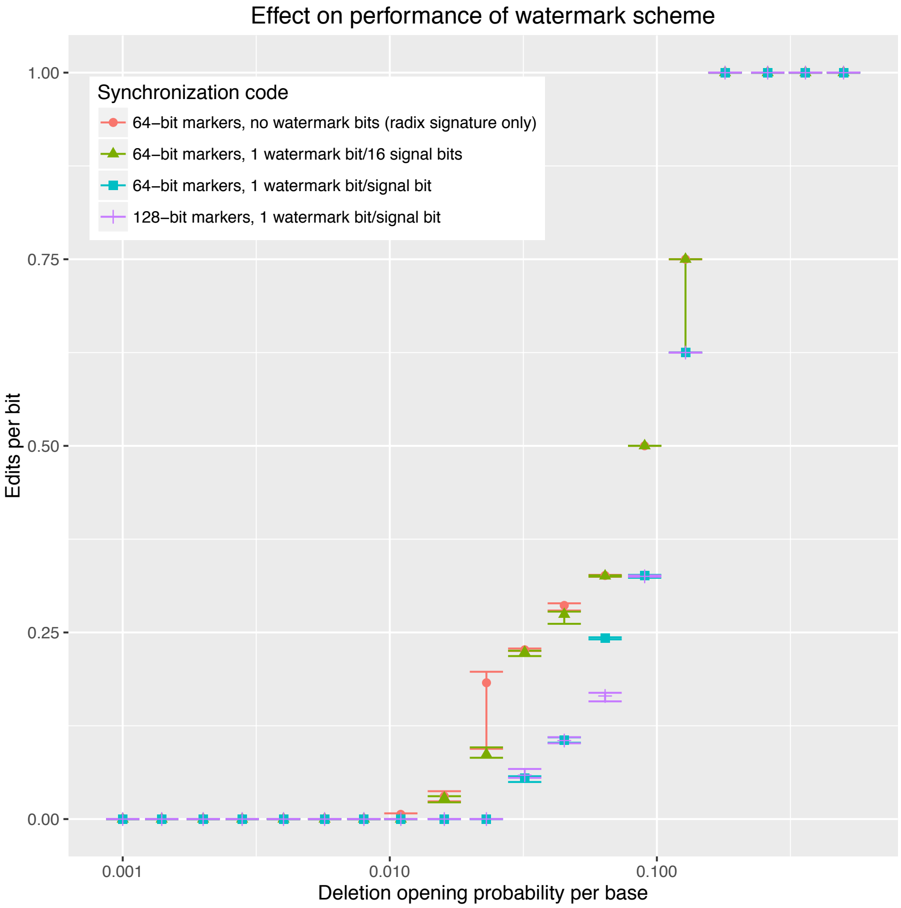

Figure 9: Effect of watermark scheme on decoder accuracy in presence of simulated deletions. The inner code was the 4-nucleotide nonrepeating DNASTORE code (Table 2). Next were various watermarking options: (i) 64 bits of signal flanked by a 4-nucleotide control word, with a small amount of watermarking information encoded in the choice of ternary and quaternary digits used to encode each bit; (ii) as (i) but with one pure watermark bit for every 16 signal bits; (iii) as (i) but with one pure watermark bit for every signal bit; (iv) as (iii) but with a longer period of 128 signal bits (and 128 watermark bits) between each control word. The outermost code was 2048-bit, 1024-parity bit LDPC. The x-axis is the probability of initiating a deletion event at each base; the probability of deleting a nucleotide is $\sim 2.5\times$ this, as outlined in the caption to Figure 8. The y-axis is the Levenshtein edit distance scaled by the input sequence length (8192 bits). Median and interquartile range are shown. The y-axis values are apparently discretized because the 8192-bit message is split into eight 1024-bit blocks for LDPC. The LDPC decoder discards incomplete blocks, so the decoded sequence is close to a multiple of 1024 in length. The length difference between the sequences dominates the edit distance: once the decoder starts missing whole blocks, the per-bit edit distance tends to be rounded up to the next multiple of 1/8.