The 42nd Symposium on Information Theory
and its Applications (SITA2019)
Kirishima, Kagoshima, Japan, Nov. 26–29, 2019

DNA Codes with Run-Length Limitation and Knuth-Like Balancing of the GC Contents

Danny Dubé [*] Wentu Song [†] Kui Cai [†]

Abstract— Digital information can be stored within DNA base sequences. Each base can be seen a symbol from the alphabet `A`, `C`, `G`, `T`. In order to reduce the probability of sequencing error, the sequences should obey certain constraints. Here, we choose to limit the length of the runs to three symbols, the RLL-3 constraint, and we aim at an exact balance between `A`s and `T`s versus `C`s and `G`s, the GC-contents constraint. We propose a fast construction of codewords that proceeds in two steps. First, we efficiently embed source information into a string made of `A`s, `C`s, `G`s, and `T`s that obeys RLL-3. Second, we balance the string using a variant of the Knuth balancing scheme. We analyze the time complexity and the embedding rate of our construction.

Keywords— DNA code, balanced code, run-length-limited code

1 Introduction

1.1 Coding Information into DNA

In a DNA-based storage system, the payload binary data is mapped to a large number of DNA sequences. A DNA sequence can be viewed a string on the alphabet $\Sigma = \{\mathtt{A}, \mathtt{C}, \mathtt{G}, \mathtt{T}\}$. These DNA sequences are synthesized and stored in a DNA pool. To retrieve the original data, the stored DNA sequences are sequenced and mapped inversely to the binary user data.

1.2 Constraints on the Codes

In order to reduce the probability of errors in the processes of DNA synthesizing and DNA sequencing, various coding techniques, such as constrained coding and error-correction coding, are introduced to DNA-based storage systems [2, 3, 4, 5, 6, 7, 8, 9, 10, 12, 14]. Since the DNA are intended to carry the payload information and provide some protection against errors, we also refer to them as *codewords*. Together, these codewords form a *codebook*.

Among the various coding techniques, two seem to constitute the most important protecting factors: limiting the length of the homopolymer runs (i.e. repetitions of the same nucleotide) and balancing the GC content of DNA sequences (i.e. the fraction of the symbols in a codeword that are `G`s and `C`s).

Many other coding techniques are considered by authors or, to the very least, mentioned as potentially beneficial. Keeping a minimum Hamming distance between the codewords of a codebook provides error-correcting capabilities. It is also beneficial to keep a minimum distance between any codeword and the reverse-complement of any codeword; the same with the reverse of any codeword. Finally, not only repetitions of the same base ought to be avoided but also successive repetitions of the same substring.

1.3 Our Goals in Code Design

In this work, we aim at efficiently building codewords that obey the two major constraints; that is, we limit the length of runs and we balance the GC contents. Let us describe our constraints more precisely.

- Run-length constraint: we choose to limit the length of each run to three occurrences of the same symbol. For brevity, we call this constraint *RLL-3*.
- GC-content constraint: since we find this constraint quite easy to satisfy, we choose to require an exact ratio of one half; i.e. exactly half of the symbols of a codeword are in $\{\mathtt{C}, \mathtt{G}\}$. By saying that this constraint is *easy* to satisfy, we mean that having it obeyed exactly does not incur much redundancy.

Of course, an ever present goal consists in maximizing the quantity of information that we embed into the codewords.

2 Previous Work

Variants of this problem have been addressed several times, in the literature. Grass et al. [8] presented a method to encode binary sequences satisfying the run-length constraint with a coding rate of 1.78 bits/nt. Other methods that construct codes while obeying the run-length constraint are presented in [2, 3, 5, 7], all of which achieve code rates no greater than 1.6 bits/nt.

Based on the sequence replacement technique, Immink and Cai [9] presented a method for constructing k-constrained q-ary codes, where the length of the runs of zeros is at most k and the code rate is $\frac{n-1}{n}$. A method transforming k-constrained binary sequence into DNA nucleotide strands with homopolymer runs of length at most $\lceil \frac{k}{2} \rceil$ is also proposed in [9].

A DNA fountain method dealing with DNA sequences that obey both the run-length and the GC-content constraints was proposed in [6]. Although the rate of the resulted codes is close to the theoretical channel capacity, its iterative decoding process can lead to severe error propagation.

DNA codes with constant GC-content are intensively studied in [10], where theoretical upper and lower bounds on the maximum size of DNA codes with constant GC-content w and minimum Hamming distance d as well as some explicit construction of codes are presented.

[*] Université Laval, Canada
[†] Singapore University of Technology and Design, Singapore

Recently, Song et al. [13] have presented a construction of DNA codes that achieve an embedding rate of $\frac{2n-1}{n}$ bits/nt. Their codes obey RLL-3 strictly but they only approximately balance the GC constents. The parameter n is the size of the substrings that are used as tiles to quickly build any valid codeword. Obviously, a larger value for n leads to a better rate. However, the collections of tiles enlarges exponentially with n and, consequently, it is costly to establish and store. It is their result that we intend to improve, in this paper.

3 Proposed Scheme

3.1 Complete Procedure

Our procedure for constructing DNA codewords performs two steps.

The first step consists in building a string drawn from Σ that obeys RLL-3. Note that this string might suffer from imbalance of its GC contents. This step is the one that embeds payload data into DNA symbols.

The second step consists in performing a relatively simple transformation on the string obtained in the first step in order to make it balanced *while* taking care not to create a violation of RLL-3 by doing so. This second step is a straightforward adaptation of the well known construction of balanced binary strings by Knuth. The second step does not embed additional payload information but rather adds a prefix to the string obtained in the first step. The added prefix only has logarithmic size, compared to the string of the first step.

We choose to work on RLL-3 first and on the GC balance only after because we deem that RLL-3 is the stronger of the two constraints: it applies locally everywhere in a string and the symbol selection may be affected at any single position; e.g., as soon as a run of length-3 appears just before.

3.2 Construction of RLL-3 Obeying Strings

We start by characterizing the strings drawn from Σ that obey RLL-3. Let us consider the following definitions.

$$\mathcal{R}_l = \left\{ w \in \Sigma^l \,\middle|\, \begin{array}{l} \text{none of } \{\texttt{AAAA}, \texttt{CCCC}, \texttt{GGGG}, \\ \texttt{TTTT}\} \text{ is a substring of } w \end{array} \right\} \quad (1)$$

$$\mathcal{R}_{l,3} = \mathcal{R}_l \cap \mathcal{R}_{\max(l-3,0)} \cdot \{c^3 \mid c \in \Sigma\} \quad (2)$$

$$\mathcal{R}_{l,2} = \mathcal{R}_l \cap \mathcal{R}_{\max(l-2,0)} \cdot \{c^2 \mid c \in \Sigma\} - \mathcal{R}_{l,3} \quad (3)$$

$$\mathcal{R}_{l,1} = \mathcal{R}_l - \mathcal{R}_{l,2} - \mathcal{R}_{l,3} \quad (4)$$

The set $\mathcal{R}_l$ contains the strings of length L drawn from Σ that obey RLL-3. The sets $\mathcal{R}_{l,r}$, for $1 \leq r \leq 3$, partition $\mathcal{R}_l$ according to the length of the rightmost run in the strings. Note that we ignore the corner case for $l = 0$.

These sets grow exponentially fast in l. Let us describe the sizes of these sets using the following recurrences.

$$N_{1,r} = \begin{cases} 4, & \text{if } r = 1 \\ 0, & \text{if } 2 \leq r \leq 3 \end{cases}$$
$$N_{l,r} = \begin{cases} 3 \cdot \sum_{r'=1}^{3} N_{l-1,r'}, & \text{if } r = 1 \\ N_{l-1,r-1}, & \text{if } 2 \leq r \leq 3 \end{cases} \quad (5)$$
$$\text{if } l \geq 2$$

There are four cases. The first case indicates that there are $N_{1,1} = 4$ strings in $\mathcal{R}_{1,1}$. These four strings are plainly those in Σ. Indeed, each of them ends with a run of length 1. The second case indicates that there is $N_{1,r} = 0$ (i.e. no) string in $\mathcal{R}_{1,r}$, for $r \geq 2$. Indeed, no string of length 1 can end with a run of length greater than 1. The third case indicates that any non-trivial string in $\mathcal{R}_{l,1}$ can be obtained by picking any string x in $\mathcal{R}_{l-1}$ and adding a new symbol to the right of x, provided this symbol differs from the last one of x. In other words, there are always three choices. Changing the symbol ensures that we break the last run in x and start a new one. Finally, the fourth case indicates that any non-trivial string in $\mathcal{R}_{l,r}$, for $2 \leq r \leq 3$, can be obtained by picking a string x in $\mathcal{R}_{l-1,r-1}$ and repeating the last symbol of x. There is no freedom in the choice of the repeated symbol, here.

By doing some algebra, we can determine that the sequences $\{N_{l,r}\}_{l=1}^{\infty}$, for $1 \leq r \leq 3$, asymptotically grow with a factor of approximately 3.951. This means that strings that obey RLL-3 may embed $\log_2 3.951 \approx 1.982$ bits/nt, asymptotically.

In order to build RLL-3-obeying strings of some desired length l, we propose to perform *source decoding*. The compression model that is used here is that of enumerative coding applied on the strings from $\mathcal{R}_l$. A random string taken from $\mathcal{R}_l$ is any one among the $\sum_{r=1}^{3} N_{l,r}$ possible ones. By giving equal probability to each string in $\mathcal{R}_l$, it is possible to perform ideal source coding on such strings. Each possible string would be mapped to a compressed codeword of the same length. This ideal source coding would be done using an arithmetic coder (AC). The strategy here consists in doing the reverse: we start from payload data and pretend that it is the result of compression and we decode the string in $\mathcal{R}_l$ that "produced" this compressed data.

Of course, a perfect AC is needed to perform ideal compression and to convert between strings in $\mathcal{R}_l$ and binary payload data of exactly the same size. In practice, an AC of finite precision must be used. It is quite standard to compute the loss in performance that may be incurred by such a finite-precision AC. So, by doing quite standard calculations, we can determine how many payload bits may get embedded into a string of $\mathcal{R}_l$, in the worst case. We omit these calculations here. We simply point out that an AC does not require many bits of precision in the representation of its state in order to achieve very good performance. As a consequence the precision that is provided by performing integer calculations on a common CPU is more than sufficient, for our purpose.

3.3 Knuth-Like Balancing of the GC Contents

Knuth's technique is a well known method to build balanced blocks of bits; i.e. strings that contain exactly as many zeros as ones [11]. It has a very good data-embedding rate as the technique needs only prepend a short prefix. The prefix only has logarithmic size, compared to the payload data.

Here is how it proceeds. Let w be a string of l bits which is the payload data we wish to turn into a balanced string (or block). First, the technique locates an appropriate *flipping point* inside of w. String w sees its prefix logically reversed ($0 \leftrightarrow 1$) up to the flipping point and becomes w'. The flipping point is selected such that w' is balanced. Such a flipping point always exists, as proved by Knuth. Then, a short prefix p is prepended before w' in order to indicate where the flipping point is, so that decoding is possible. Prefix p is itself a balanced string. There is no need to perform recursive calculations to obtain p. There are only as many prefixes as there may be flipping points, so they can be kept in a pre-calculated lookup table.

In the problem of DNA coding, we can perform balancing of the GC contents exactly the way Knuth's technique does it. Indeed, symbols `A` and `T` can be viewed as zeros and `C` and `G`, as ones. However, our task here consists not just into making the string balanced but also into preserving the RLL-3 property. Let us consider an RLL-3-obeying string w. Flipping some prefix of w to obtain w' might, with some probability, create a run that is longer than 3 at the flipping point. That must be avoided. Moreover, even if we correctly flip w to obtain w', we might also create too long a run when we prepend a prefix p to w'. That also must be avoided.

Here is how we proceed to avoid these two risks. First, we must prepare more prefixes than in Knuth's technique. In fact, we prepare four prefixes per flipping point, instead of just one. Of course, each prefix obeys RLL-3 and the balance of its GC contents. Also, we make sure that, for any given flipping point, the four prefixes end with a distinct symbol. Otherwise, we do not care which prefix is assigned to which flipping point.

For a given flipping point, say f, let $p'\texttt{A}$, $p''\texttt{C}$, $p'''\texttt{G}$, and $p''''\texttt{T}$ be the four prefixes that it possesses. Any of them means that the flip has occurred at point f. However, when prefix $p'\texttt{A}$ or $p''\texttt{C}$ is used, it means that the flip is made using the mapping $\texttt{A} \leftrightarrow \texttt{C}$ and $\texttt{T} \leftrightarrow \texttt{G}$. Otherwise, when prefix $p'''\texttt{G}$ or $p''''\texttt{T}$ is used, it means that the flip is made using the mapping $\texttt{A} \leftrightarrow \texttt{G}$ and $\texttt{T} \leftrightarrow \texttt{C}$. The use of one or the other mapping ensures that flipping does not create a long run at f.

Finally, we explain why we prepare two prefixes for a given flipping point and a given mapping. Having two different prefixes, each ending with a different symbol, ensures that we can prepend one of the two prefixes without creating a long run at the concatenation point.

This way, our second step ensures that we obtain a codeword drawn from Σ that both obeys RLL-3 and the GC-content balance.

References

[1] Krishna Gopal Benerjee, Sourav Deb, and Manish K. Gupta. On conflict free DNA codes. Technical Report 1902.04419v2, ArXiV, March 2019.

[2] Meinolf Blawat, Klaus Gaedke, Ingo Hütter, Xiao-Ming Chen, Brian Turczyk, Samuel Inverso, Benjamin W. Pruitt, and George M. Church. Forward error correction for DNA data storage. *Procedia Computer Science*, 80:1011–1022, 2016.

[3] J. Bornholt, R. Lopez, D. M. Carmean, L. Ceze, G. Seelig, and K. Strauss. A DNA-based archival storage system. In *Proc. 21st Int. Conf. Architectural Support Program. Lang. Operating Syst.*, pages 637–649, 2016.

[4] D. Cartwright and T. C. Gleason. The number of paths and cycles in a digraph. *Psychometrika*, 31(2):179–199, 1966.

[5] G. M. Church, Y. Gao, and S. Kosuri. Next-generation digital information storage in DNA. *Science*, 337(6102):1628, 2012.

[6] Y. Erlich and D. Zielinski. DNA fountain enables a robust and efficient storage architecture. *Science*, 355(6328):950–954, 2017.

[7] N. Goldman et al. Towards practical, high-capacity, low-maintenance information storage in synthesized DNA. *Nature*, 494:77–80, January 2013.

[8] R. N. Grass, R. Heckel, M. Puddu, D. Paunescu, and W. J. Stark. Robust chemical preservation of digital information on DNA in silica with error-correcting codes. *Angew. Chem. Int. Ed.*, 54(8):2552–2555, 2015.

[9] K. A. S. Immink and K. Cai. Design of capacity-approaching constrained codes for DNA-based storage systems. *IEEE Commun. Lett.*, 22(2):224–227, February 2018.

[10] Oliver D. King. Bounds for DNA codes with constant GC-content. *The Electronic Journal of Combinatorics*, 10:33–45, September 2003.

[11] Donald E Knuth. Efficient balanced codes. *IEEE Trans. on Information Theory*, 32(1):51–53, 1986.

[12] K. P. Murphy. *Machine Learning—A Probabilistic Perspective.* MIT Press, Cambridge, MA, USA, 2012.

[13] Wentu Song, Kui Cai, Mu Zhang, and Chau Yuen. Codes with run-length and GC-content constraints for DNA-based data storage. *IEEE Communications Letters*, 22(10):2004–2007, October 2018.

[14] S. M. Hossein Tabatabaei Yazdi, Yongbo Yuan, Jian Ma, Huimin Zhao, and Olgica Milenkovic. A rewritable, random-access DNA-based storage system. *Scientific Reports*, 5(14138), September 2015.