Enhanced Context Sensitive Flash Codes*

Gilad Baruch[1], Shmuel T. Klein[1] and Dana Shapira[2]

[1]Department of Computer Science, Bar Ilan University, Ramat-Gan 52900, Israel
[2]Department of Computer Science, Ariel University, Ariel 40700, Israel
Email: gilad.baruch@biu.ac.il, tomi@cs.biu.ac.il, shapird@g.ariel.ac.il

A major property of flash memory is that a 0-bit can be changed into a 1-bit, but the symmetric task of switching from a 1-bit to a zero may only be performed in blocks, and is therefore often prohibited. This led to the development of rewriting codes using the same storage space more than once, subject to the constraint that 0-bits can be changed into 1-bits, but not vice versa. Context sensitive rewriting codes extend this idea by incorporating also information gathered from surrounding bits. Several new context sensitive rewriting codes are presented and analyzed, some of which are better than the state of the art for sparse input. Empirical simulations show a good match with the theoretical results.

Received 00 Month 2020; revised 00 Month 2020

1. INTRODUCTION

Flash memory is one of the most popular storage media today [2, 3]. They can be found in our computers, cell phones and many other devices we use on a daily basis. One of the distinctive features of flash devices is that writing zeros or ones is not symmetrical: changing a 0 into a 1 is cheap and can be performed for each individual bit, whereas the switch from 1 to 0 is only possible by erasing entire blocks (of size 0.5MB or more), and is considered as being so expensive that one tries to avoid it, or at least, delay it as much as possible, contrarily to magnetic memory used so far.

This asymmetric behavior triggered the development of so-called *rewriting codes*, see, for example, [4, 5], which try to reuse the *same* storage space, after a block of bits has already been used to encode some data in what we shall call a *first round* of encoding. When new data should be encoded in a *second round*, the question is how to use the same bits again, without having to erase the entire block before rewriting. The problem can be generalized to three or more writing rounds, all with the same constraint of changing only 0s to 1s.

In fact, Rivest and Shamir [6] suggested a simple way to use three bits of memory to encode two rounds of the four possible values of two bits, long before flash memory became popular. They called these special codes *Write-Once Memory* (WOM), and we shall refer to the Rivest-Shamir code below as RS-WOM.

The efficiency of a given rewriting code may be measured by a compression ratio, referred to as *sum-rate* in the rewriting codes literature. It is defined as the number of provided *information bits* divided by the number of actually used *storage bits*. The number of information bits is in fact the information content of the data, whereas the number of storage bits depends on the way the data is encoded. For a standard binary encoding, information and storage bits are equivalent, giving a baseline of 1. For rewriting codes, we use the combined number of storage bits of all (two or more) writing rounds, thus the above mentioned RS-WOM-code yields a ratio of $\frac{4}{3} = 1.333$. For two rounds, the theoretical best possible ratio is $\log 3 = 1.585$, see [7], and the best ratio achieved so far is 1.509 [8].

Many rewriting codes, and RS-WOM in particular, treat each encoded element independently of those preceding it. A new paradigm of *context sensitive* rewriting codes was introduced in [9] and extended and analyzed in [10], suggesting to use a Fibonacci encoding in the first round. Such a binary encoding has the property that it contains no adjacent 1-bits [11], which means that every 1-bit must be followed by a zero. This can then be exploited in a second round to store new information in these 0-bits, which can be located using their context. The resulting compression ratio, though, was only 1.19 in the best case and 1.145 at average, which is inferior even to the simple RS-WOM.

In fact, the features of Fibonacci codes have been employed in a variety of other applications such as the robustness to errors [12], direct access [13], fast decoding, compressed matching [11, 14, 15], and compressed data structures [16]. The present work is yet another application of Fibonacci related encodings. It introduces several new context sensitive

* This is an extended version of a paper that has been presented at the International Conference on Implementation and Application of Automata (CIAA'19) in 2019, and appeared in its proceedings [1].

rewriting codes and shows their performance either analytically or by means of empirical tests. They improve the previously known codes but still do not always outperform RS-WOM. The main contribution is the development of the new methods themselves, showing several techniques how the Fibonacci based rewriting codes can be extended. We did, so far, not succeed in deriving a new method that consistently outperforms the best state of the art compression ratios for all possible data densities of the input, but other researchers might find some new variants that do, following similar ideas as those to be presented below. For sparse data, that is, when the probability of a 1-bit is low, several of our suggested new methods are better than the non context sensitive state of the art.

The next section recalls some details of the Fibonacci WOM codes. Section 3 presents enhanced context sensitive flash codes. Experimental results are presented in Section 4, and Section 5 concludes.

2. FIBONACCI WOM CODES

Any integer can be represented as a binary string in many different ways. The standard representation uses the powers of 2 as basis elements, whereas Fibonacci codes are based on the famous Fibonacci sequence, defined by $F_i = F_{i-1}+F_{i-2}$ for $i \geq 1$, and the boundary conditions $F_0 = 1$ and $F_{-1} = 0$.

Any integer x can be decomposed into a sum of distinct Fibonacci numbers, and can therefore be represented by a binary string $c_r c_{r-1} \cdots c_2 c_1$ of length r, called its Fibonacci or Zeckendorf representation [17], such that $x = \sum_{i=1}^{r} c_i F_i$. The representation of x will be unique if one starts with the *largest* Fibonacci number F_r smaller or equal to x and then continues recursively with $x - F_r$. For example, $77 = 55+21+1 = F_9+F_7+F_1$ so its binary Fibonacci representation would be 101000001. As a result of this encoding procedure, there are never consecutive Fibonacci numbers in any of these sums, or, equivalently, the corresponding binary representation does not contain adjacent 1s.

Fibonacci WOM codes are constructed in three stages. In the first step, the n bits of the block are transformed into a block of size $r = 1.44n$ by considering the n bits as the standard binary representation of some integer and recoding this integer into its Fibonacci representation. The resulting block will be longer, since more bits are needed, but generally also sparser, because of the property of prohibiting adjacent 1s. When the data is not needed anymore and can be overwritten, the next essential step is to fill in a maximal number of 1-bits without violating the non-adjacency property of the Fibonacci encoding. This means that in a run of zeros of odd length $2i + 1$, every second zero is turned on, and this is true also for a run of zeros of even length $2i$, except that for the even length, the last bit is left as zero, since it is followed by a 1. As a result of this filling strategy, the data block still does not have any adjacent 1s, but the lengths of the 1-limited zero-runs are now either 1 or 2, and the length of the leading run is either 0 or 1.

Finally, in the third step new data is encoded in the bits immediately to the right of every 1-bit. Since it is known that these positions contained only zeros at the end of step 2, they can be used at this stage to record new data, and their location can be identified. It has been shown that the compression efficiency of the Fibonacci WOM code is 1.194, 1.028, and 1.145, in the best, worst and average cases. In the following sections we show how the compression performance can be improved by extending the above idea.

3. ENHANCED CONTEXT SENSITIVE FLASH CODES

3.1. Fibonacci + 2 ⟶ 1

The storage penalty incurred by passing from the standard binary representation to the Fibonacci representation is a factor of $\log_\phi 2 = 1.44$, for any block size n, where $\phi = 1.618$ is the golden ratio obtained by taking the ratio of two consecutive Fibonacci numbers F_{k+1}/F_k and letting $k \to \infty$. Thus each of the n bits in the first round represents, on the average, only $\frac{1}{1.44}$ of a data bit of the original data.

The best case of the WOM code suggested in [10] occurs when every second bit in the Fibonacci representation is a 1-bit, and therefore it is followed by a 0-bit that can serve as a data bit. In this case $\frac{n}{2}$ data bits can be written in the second round, giving a total compression ratio of

$$\frac{1}{n}\left(\frac{1}{1.44}n + \frac{1}{2}n\right) = 1.194. \quad (1)$$

The following simple method achieves the same ratio, but not only on a single best case input, but for *all* possible outcomes of the first writing round, which, as before, is based on writing the data in its Fibonacci representation. The second step, however, treats every non-overlapping pair of successive bits separately. There are only three kinds of possible pairs: `00`, `01` and `10`. If one wishes to write `0`, the pair is left unchanged, that is, `00`, `01` and `10` all represent the value `0` in the second round. In case one wishes to output a `1`, the pair is overwritten by the pair `11`.

Each bit in the second round is thus encoded using 2 of the n bits, which again yields the same compression ratio as in (1).

3.2. Ternary + 2 ⟶ 1

A further improvement may be based on the awareness that the above method does not take advantage of the fact that the pair `01` is never followed by `10`, suggesting to relax the requirements of the Fibonacci representation used in the first round. Instead of prohibiting the appearance of the substring `11` altogether, which is equivalent to using a Fibonacci

encoding, we forbid the occurrence of the pattern `11` only at even indices in the string (indices are numbered starting with 0), but allow `11` to appear at odd indices. In other words, if we parse the string in pairs and therefore consider only pairs starting at even indices, `01` may be followed by `10`, since in this case the `11` formed by the concatenation of `01` and `10` occurs at an odd index and is therefore permitted. In fact, using this encoding, every number is now represented in a ternary code using the symbols `00`, `01` and `10`, and this is an improvement over the Fibonacci encoding.

In the second round, the bit stream is parsed into pairs of bits just as in the second round of the Fibonacci + 2 → 1 method of the previous subsection, so that the second round again adds $\frac{1}{2}$ to the compression ratio. To calculate the improved contribution of the first round, note that a string of k trits (ternary digits) can be used to store numbers between 0 and $3^k - 1$ in ternary representation. If each trit is encoded by two bits, an n-bit number in binary representation uses $\log(3^{\frac{n}{2}}) = \frac{n}{2}\log_2 3 = \frac{1.58n}{2} = 0.792n$ bits in the first round. The total compression ratio is thus

$$\frac{1}{n}\left(0.792n + \frac{1}{2}n\right) = 1.292. \qquad (2)$$

3.3. Fibonacci + Lookahead

We revert back to the Fibonacci encoding for the first round, and suggest a different way to exploit our knowledge that `01` is not followed by `10`. The idea is to apply a lookahead technique to the currently processed bit, and act according to both its value and the value of the new bits we wish to write.

The second round starts again by parsing the bit stream into bit-pairs, having only three possibilities `00`, `01` and `10`. As in the Ternary +2 → 1 method, we shall encode a `1`-bit in the second round by `11`, and a `0`-bit by either `00`, `01` or `10`. In addition, we now identify and exploit certain bits that can be used as data-bits because of their context, thereby increasing the total number of bits we are able to write in the second round.

Denote the current pair to be overwritten by A. If it is a `0`-bit that needs to be written, the following cases are considered:

1. if A = `10`: the pair is left unchanged, and no further bits can be treated in this iteration.
2. if A = `01`: `01` is used to represent `0`, but we use the fact that the following bit is a `0`, so it can be used, alone, as a data bit (and be left as a `0` or be turned into a `1`);
3. if A = `00`: in this case, we may look even further, and consider the following *two* bits after A, denote them as B.
   (a) If B = `00`, the current pair A is left as `00`, and the two bits of B can be used as data bits.
   (b) if B = `01`, the current pair A is turned from `00` into `01`, which brings us back to case 2., and the next `0`-bit is a single data bit. Note that in this case, the 1-bit of B will be processed again in the next iteration.
   (c) if B = `10`, the current pair A is turned from `00` into `10`, corresponding to case 1. above. We then already know that the following pair is also `10`, but this one will be treated only in the next iteration.

The decoding, accordingly, mainly parses bit-pairs: if the pair is `11` or `10`, the output is `1` and `0`, respectively; if the pair is `01` or `00`, the output is `0`, and it is followed, respectively, by one or two data-bits, each of which may contain `0` or `1`, independently from their context. This decoding procedure for the second round is summarized in the automaton presented in Figure 1. The decoded values are given in rectangles, and the decoding starts at the initial state, identified by the gray node on the left. An outgoing edge, labeled by d, is used to identify a data bit: this means that the current bit, either 0 or 1, serves itself as output, and not just as a codeword.

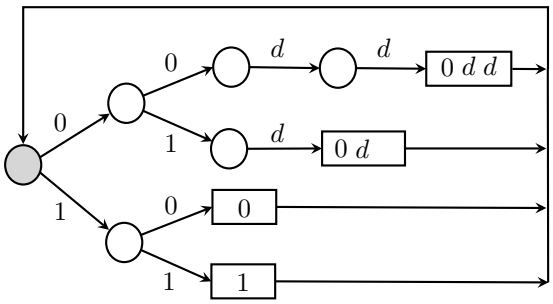

FIGURE 1. Fibonacci + Lookahead decoding automaton

As example, assume that in the first round we are interested in storing the value 4,340. The corresponding Fibonacci representation, `100000010000100010`, appears on the top line of Figure 2. The bits considered in lookahead mode (two bits after 00 and a single bit after 01) are overlined. Data bits are boxed. Note in particular the second appearance of 00: two bits are examined by lookahead, but only the first is actually used as a data bit, and the second is processed again in the next iteration.

Suppose the new data to be stored is the number 1,325 in its standard binary form, that is `10100101101`, presented on the second line. The third line of Figure 2 are the bits that are actually stored in the second round.

As a decoding example, consider the binary stream `110010011011111011`. Following the decoding automaton, the input is parsed as `11 0010 011 011 11 10 11` in which spaces are inserted for clarity each time the automaton returns to its initial state, and the decoded output is `1 010 01 01 1 0 1`.

In the best case, the output of the first round is a string consisting only of zeros, and we wish to write data

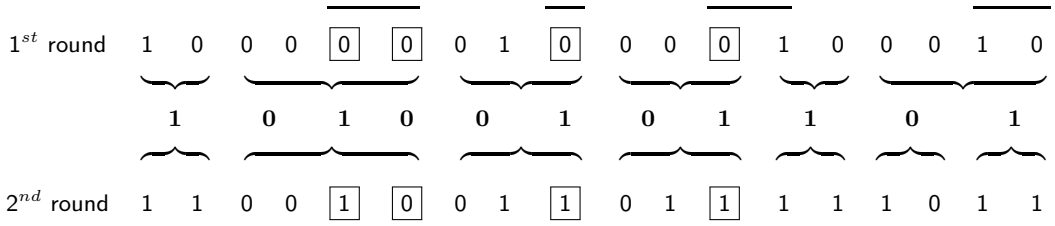

FIGURE 2. Fibonacci + Lookahead encoding example

in the second round in which the bits in all positions with index $1 + 3k$ for $k \geq 0$, are zeros. The string of zeros will then be parsed in quadruples, where each pair `00` is translated into one bit, and then the following two zeros are encoded as individual data bits. Thus, every 4-tuple gives rise to three data bits. The ratio is therefore $1/(\log_\phi 2) + 3/4 = 1.444$.

In the worst case, the output of the first round is a string of alternating ones and zeros of the form $101010\cdots$. In this case, the parsing will be into a sequence of `10` pairs, each of which will be left unchanged to encode a 0-bit, or changed into `11` to encode a 1-bit. The ratio is therefore, as seen before, $1/(\log_\phi 2) + 1/2 = 1.194$.

For the average case, we need to know the distribution of the pairs `00`, `01` and `10` in a Fibonacci encoded strings. Denote the probability of occurrence of these pairs by p_{00}, p_{01} and p_{10}, respectively. Consider the parsing into pairs of an infinite stream of bits that has been generated under the Fibonacci constraint that no adjacent 1's appear. If we shift the parsing by a single bit, all `10` pairs turn into `01` pairs and vice versa. On the other hand, such a shift should not affect the overall probabilities of the occurrences of the different pairs, so we may conclude that $p_{01} = p_{10}$. However, since every 1-bit is followed by 0, p_{10} is equal to p_1, the probability of a single 1, which has been shown in [18] to be $p_1 = \frac{1}{2}\left(1 - \frac{1}{\sqrt{5}}\right) = 0.2764$. We can thus also derive $p_{00} = 1 - 2p_1 = 0.447$.

To evaluate the average compression ratio, we assume that the data we wish to write has a 1-bit density of q, with $0 < q \leq \frac{1}{2}$. For random data, $q = \frac{1}{2}$, which is not unrealistic, as this will be the case for most compressed or encrypted files. If $q > \frac{1}{2}$, we may just encode the 1's complement of the data. Following the encoding details above, the contribution of the second round to the average compression ratio will be

$$\frac{1}{2}q + \left[p_{10}\frac{1}{2} + p_{01}\frac{2}{3} + p_{00}\left(p_{00}\frac{3}{4} + p_{01}\frac{2}{3} + p_{10}\frac{1}{2}\right)\right](1-q) = 0.617 - 0.117q.$$

In particular, for $q = \frac{1}{2}$, we get an average compression ratio of

$$1/\log_\varphi 2 + 0.558 = 1.253. \quad (5)$$

3.4. Fibonacci + Inspect

We now take the previous idea of one step forward, and propose to combine the lookahead with the $2 \to 1$ technique, which actually simplifies the processing, because no data bits are interleaved. Denote the value of the currently processed bit by C.

As above, if it is a `1`-bit that needs to be written in the second round, the following pair of bits, whose value is either `00`, `01` or `10`, is turned into `11`. If, on the other hand, we wish to write a `0`-bit, then if $C = 0$, this single bit suffices for the encoding. If $C = 1$, it must be followed by a zero, so the pair `10` can be used to encode the `0` value. Decoding of the second round is according to the automaton of Figure 3.

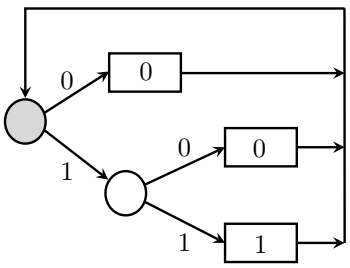

FIGURE 3. Fibonacci + Inspect decoding automaton.

As example, assume that in the first round we are interested in storing the value 112. The corresponding Fibonacci representation, `1001000010`, appears on the top line of Figure 4. Suppose the new data to be stored is the number 38 in its standard binary form, that is `100110`, presented on the second line. The third line of Figure 4 are the bits that are actually stored in the second round.

For decoding, consider the binary stream output of the second writing round, `1101011110`, of Figure 4, which is parsed as `11 0 10 11 11 0` by the automaton of Figure 3, yielding the decoded output `1 0 0 1 1 0`, as expected.

In the worst case, every bit to be written in the second round will require two bits, either because a `1` is to be written, or because the currently seen pair is `10`, so we get a ratio of 1.19 as already shown.

For the average case, we use the same notation as above to denote the distribution of the pairs `00`, `01` and

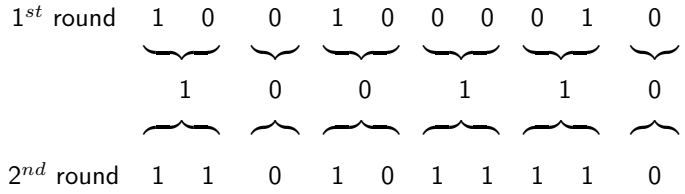

FIGURE 4. Fibonacci + Inspect encoding example

.

10 in a Fibonacci encoded strings, and assume again a 1-bit density of q, with $0 < q \leq \frac{1}{2}$, for the data in the second round.

The decoding of the second round is done by iterations processing either a single bit or a pair of bits. We first need to know the probability of the event F, that the first bit of a given decoding iteration is a 0-bit. This will be evaluated by conditioning on the bits written in the preceding iteration. Let G stand for the event that the last bit written in the previous iteration of the second round was a 0. We have that

$$P(F) = P(F|G)\,P(G) + P(F|\overline{G})\,P(\overline{G}).$$

But G occurs if and only if the bit written in the previous iteration was a 0, so we know that $P(G) = 1-q$ and $P(\overline{G}) = q$.

If at the end of the second round, the previous bit was a 0, this was also true at the end of the first round, since 1s can not be turned into zeros. Therefore, the event $F|G$ is equivalent to having seen a 0-bit after a 0-bit in the Fibonacci encoding, and using our previous notation, we get that the probability $p_{0\text{-}0}$ of this event is

$$\begin{aligned} p_{0\text{-}0} = P(F|G) &= \frac{p_{00}}{p_{00}+p_{01}} = \frac{1-2p_1}{1-p_1} \\ &= \frac{1}{\sqrt{5}\,\frac{1}{2}\left(1+\frac{1}{\sqrt{5}}\right)} = \frac{1}{\frac{\sqrt{5}+1}{2}} = \frac{1}{\phi} = 0.618. \end{aligned}$$

$P(F|\overline{G})$ is the probability of writing now a zero bit knowing that the last bit written in the previous iteration of the second round was a 1. We thus know that the bit value written in the second round was a 1 and has been encoded by overwriting either 00, 01 or 10 by the pair 11. Denote by R the value of the bit pair which has been overwritten. If $R = 01$, the following bit must be a zero, so the probability $P(F|\overline{G} \wedge (R = 01)) = 1$. In the other cases, $R = 00$ and $R = 10$, and the last bit written was a 0, we thus get that $P(F|\overline{G} \wedge (R = 00 \vee R = 10)) = p_{0\text{-}0}$, the probability of writing a 0 after a 0. Putting it together, we derive

$$\begin{aligned} P(F) &= p_{0\text{-}0}(1-q) + \Big(p_{01}\cdot 1 + (p_{10}+p_{00})p_{0\text{-}0}\Big)\,q \\ &= 0.618 + 0.105\,q. \end{aligned}$$

To calculate the expected number of bits $E(N)$ to be written in the second round, note that we write a single bit only if the current bit was a 0 at the end of the first round, and we wish to write a 0-bit. Denote this event as Y, then we have $P(Y) = P(F)(1-q)$. In fact, F has been defined relative to the second round, but as mentioned, a zero in some bit position at the end of the second round implies that this bit was also a 0 at the end of the first round. If Y does not occur, two bits will be written, so the expected number of bits is $P(Y) + 2(1-P(Y)) = 2 - P(Y)$, and substituting the values above, we get

$$\begin{aligned} E(N) &= 2 - (0.618 + 0.105\,q)(1-q) \\ &= 0.105\,q^2 + 0.513\,q + 1.382. \end{aligned}$$

Though this is a quadratic equation, its graph is, on the given range $[0, \frac{1}{2}]$, very close to a straight line and thus well approximated by a linear equation $E(N) \simeq 0.5655\,q + 1.377$. This yields as average compression ratio $1/(\log_\phi 2) + 1/E(N)$, and in particular, for $q = \frac{1}{2}$, we get 1.295, and for $q \to 0$, the ratio approaches 1.418, which is better than RS-WOM.

3.5. Fibonacci + $3 \longrightarrow 2$

We now extend the methods by treating larger blocks. Instead of encoding single bits by pairs, we aim at encoding bit-pairs by triplets, as done in the RS-WOM code. Each such triplet is interpreted as one of the four possible bit pairs: 00, 01, 10 or 11, and is transformed into another bit triplet representing the following bit pair to be written in the second round.

The first round of RS-WOM encodes the four possible pair values by either 000, 001, 010 or 100. We use again the standard Fibonacci encoding, which yields one more possible triplet: 101. For the second round, the data is parsed into packages of three consecutive bits, and the translation from the given to the newly generated triplet is done according to graph given in Figure 5. We use a color code to help the reader, where red, blue, yellow and green nodes represent the pairs 00, 01, 10 and 11. To enable viewing the differences on non-colored output, we also label the nodes with the initials R, B, Y and G of their colors. Any permutation could be used to match colors and pairs, as long as it is fixed throughout.

The nodes on the left hand side of Figure 5 represent the possible triplets resulting from the Fibonacci encoding of the first round. A transformation from triplet x to triplet y is indicated by a directed edge

(x, y), and all these transformations are according to the flash memory constraint that a 0 can be turned into a 1, but not a 1 into a 0. The white node, representing 000, has four outgoing edges, one to each color. The yellow node, representing 100, has only three outgoing edges, to the three colors which differ from yellow. Thus if we want to represent the yellow pair, the corresponding triplet is left unchanged, which is similar to the second round encoding of RS-WOM. Similarly, the green and blue nodes, representing 010 and 001, have also only three outgoing edges, to their complementing colors.

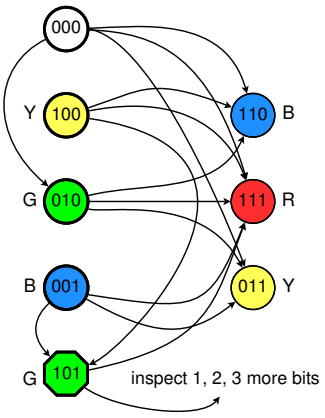

FIGURE 5. Transitions for Fib + 3 → 2 encoding.

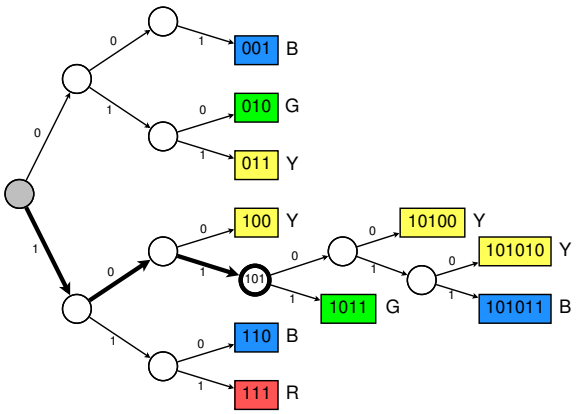

FIGURE 6. Decoding automaton for the second round of Fibonacci + 3 → 2 encoding.

If the input triplet is 101, it needs a special treatment, which is why the corresponding node appears as an octagon in Figure 5 rather than as a circle. The problem is that it is the sole possible triplet having only a single 0-bit, so there are only two options for encoding in the second round: either leave the triplet as 101, or transform it into 111. To overcome this difficulty, since we need 4 options to be encoded, we inspect and use some of the consecutive bits. Denote the three bits immediately following the triplet 101 as b_1, b_2 and b_3, and consider the following cases:

1. If the color we wish to encode is **red**, 101 is turned into 111;
2. otherwise, if the color we wish to encode is **green**, 101 is turned into 1011, that is, the following bit b_1, which can be either 0 or 1, is set to 1. The reason that b_1 could also be 1 is that the origin of the triplet 101 might be the encoding of yellow as 100 in the first round.
3. otherwise (yellow or blue), we know that at the end of the first round, the block was 101, implying that $b_1 = 0$, which will be left unchanged. If the color we wish to encode is yellow, inspect the following bit b_2, which can be either 0 or 1. If $b_2 = 0$, leave it and use this to indicate that yellow has been chosen. If $b_2 = 1$, then we know that $b_3 = 0$. Therefore
   (a) If the color we wish to encode is **yellow**, leave $b_3 = 0$;
   (b) otherwise (**blue**), turn $b_3 \leftarrow 1$.

Note that if we see 101 at the end of the second round, its origin is either (1) a 101 triplet already at the end of the first round, or (2) it may have been obtained from a transformation of 100 or 001 to encode the color green. In both cases, we need to inspect the following bits. For case (2), only one more bit is necessary, for case (1), we need one, two or three additional bits.

The decoding automaton corresponding to this encoding can be seen in Figure 6. The initial state is the gray node at the root of the tree, and the back edges from the leaves to the initial state have been omitted. The special node 101 and the path leading to it are emphasized. We omit the analyses of the methods of this and the following sub-section, but bring empirical test results in the experimental section thereafter.

3.6. Fibonacci + 5 ⟶ 3

A further extension of this 3 → 2 approach to deal with even larger blocks can be to conceive a 5 → 3 method that will process blocks of 5 bits of the output of the first round in order to encode each of the 8 possible triplets of the new data to be written in the second round. Only 13 of the 32 possible 5-bit strings comply with the Fibonacci constraint of avoiding adjacent 1s, and Figure 7 is one of the possible transition graphs similar to the one of Figure 5, but treating eight instead of just four colors: magenta, violet, cyan, yellow, red, green, blue and ochre, indicated in the figure and below by their initials M, V, C, etc., and representing, respectively, the triplets $000, 001, \ldots, 111$. Of course, any other assignment of the colors to the triplets could be used.

The elements in the gray rectangle in the center of Figure 7 are the 13 possible 5-tuples that can be obtained after having used a Fibonacci encoding in the first round. Similar to the case of the triplet 101 for the 3-to-2 method, there is again a special case, the 5-tuple 10101, which is the only one with just two zeros. It can thus only produce four out of the eight required possibilities for the second round, and we need,

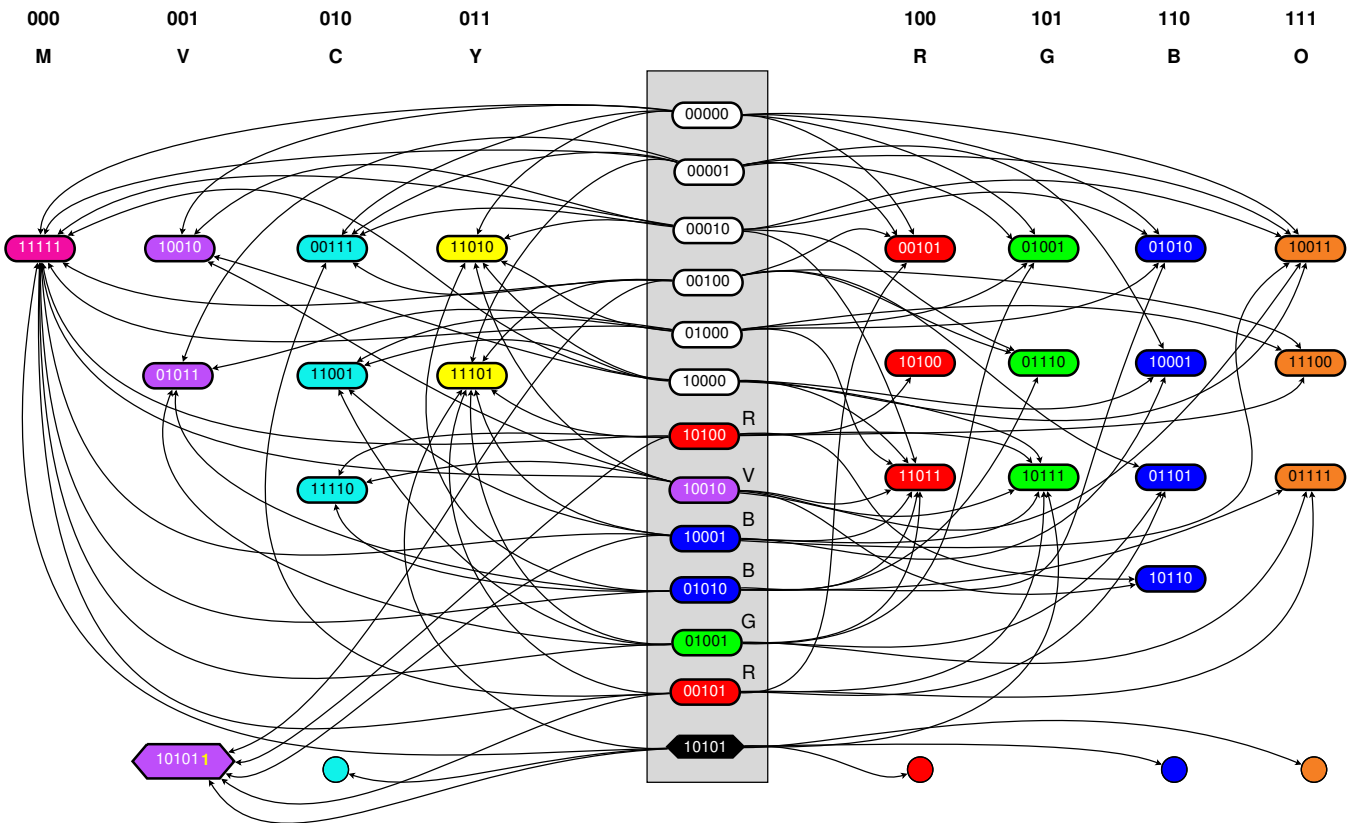

FIGURE 7. Transitions for Fibonacci + 5 → 3 encoding.

as above, an alternative treatment for this case. Like before, this case is visually distinguished by being the only one represented in a hexagonal shaped box and with black background.

There are 22 possible 5-tuples at the end of the second round, including the 7 lower elements of the middle column in the gray rectangle, and they are organized here in eight columns, each corresponding to one of the colors. There are thus several alternatives to encode a color, for example, both 11010 and 11101 will represent Yellow. Magenta is only represented by 11111; no alternative is needed in this case, since every 5-tuple can be transformed into 11111 without violating the flash memory constraint. From each of the elements in the middle column, there are eight outgoing edges, one to each of the possible colors. These edges describe the encoding possibilities. For example, if the input block contains 01001, then if we want to encode magenta, the output will be 11111; if we want to encode violet, the output will be 01011; similarly, the six remaining transitions are to 11001 for cyan, 11101 for yellow, 11011 for red, 01001 for green (that is, the input block remains unchanged), 01101 for blue and 01111 for ochre. Note that all these transitions comply with the constraint of changing only zeros to 1s and never 1s to 0s, and this is true also for all the other transitions represented by the edges in Figure 7.

The special case 10101 can change to 10111 (green), 11101 (yellow) or 11111 (magenta). It could also be left as is to encode violet, but this will not be enough. In fact, the 5-tuple 10101 has two distinctive roles: on the one hand, it serves as one of the alternatives to encode the color violet in the second round if the 5-tuple in the first round was 00100, 10100, 10001, 00101 or 10101 itself; on the other hand 10101 could be one of the possible 5-tuples of the first round, and then we have to enable transitions to all 8 colors. In particular, there are missing transitions emanating, in Figure 7, from the black node corresponding to 10101, those to colors cyan, red, blue and ochre, and they are indicated by edges to small circles at the bottom of the corresponding columns.

This problem is solved by again inspecting the following bits as we did for the 3 → 2 encoding; up to 7 additional bits, denoted b_1 to b_7, will be necessary to deal with all the possibilities. Figure 8 displays the relevant part of the corresponding decoding automaton. The coding alternatives rely on the fact that if a 1-bit is encountered, we know that the following bit must be zero, so it can be used as data-bit. The branchings corresponding to such data bits are emphasized in Figure 8 as boldfaced, broken arrows, and are labelled by green 0s and 1s.

Refer as example to the root of the subtree, representing the element 10101. It has outgoing edges labelled both 0 and 1, because 10101 could have been

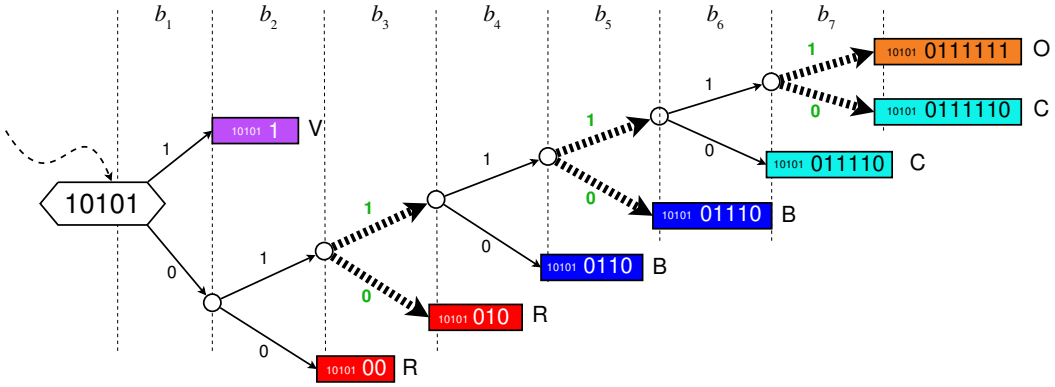

FIGURE 8. Part of the decoding automaton for the second round of Fibonacci + 5 → 3 encoding.

obtained from a transition from 10100 or 00100, so the following bit is not necessarily a zero. If $b_1 = 1$, we use the entire string 101011 including b_1 to represent V; if $b_1 = 0$, we also inspect b_2: if $b_2 = 0$, the string 1010100 including b_1b_2 is used to represent R; if $b_2 = 1$, then we know that $b_3 = 0$. In this case, we can decide about the value of the next bit according to the color we wish to encode. If it is R, we leave $b_3 = 0$, otherwise, to encode one of the remaining colors B, C or O, we set $b_3 = 1$. The decisions at other internal nodes in Figure 8 are derived similarly.

Note that as a result of the additional bits we have to inspect in order to accommodate all the coding alternatives, the color violet will be encoded by 6 bits 101011 and not just by 5 when the origin of the transition is one of the 5-tuples 00100, 10100, 10001, 00101 or 10101 itself. This is the reason for the different representation of 101011 in the column of the color violet in Figure 7.

3.7. 2.5-ary + Inspect

The following variant generalizes the Fibonacci approach for the first round and can then be combined with any of the suggested encodings for the second round, 2 → 1, 3 → 2, 5 → 3, Inspect, or Lookahead. The idea is to enforce the non-adjacency property of 1-bits by simply inserting a 0-bit after each 1-bit. In other words, we encode, in the first round, a 0 by itself, but a 1 by the pair 10. The output is then a bit sequence with the same property as the Fibonacci encoding, but we gain the additional property that there is no need to handle entire blocks, and the data can be processed as a stream, similar to the RS-WOM code.

This first round can then be combined with any of the above mentioned schemes for the second round, and we present the best combination, using the Lookahead method described in Section 3.4 in the second round. That is, if a 1 bit is to be written in the second round, the following pair of bits is turned into 11, and if a 0 bit is to be written, it is encoded as 0 if the following bit is already a 0 and as 10 if the following bit is a 1.

Since the ternary approach processes 2 bits to get 3 values and a binary approach processes one bit to get 2 values, the current method which processes one or two bits to get 2 values is some compromise, so we call it 2.5-ary. At first sight, this 2.5-ary approach seems to be disadvantageous, because the storage overhead incurred by passing from the standard binary representation to this variant is a factor of 1.5 for an evenly distributed bit-stream, instead of only 1.44 for the Fibonacci encoding. However, the probability of a 1 bit may vary, and it is not necessarily equal to $\frac{1}{2}$ as for random, compressed or encrypted data. A case in point would be the MNIST database[3] of handwritten digits that is commonly used in the Machine Learning and Computer Vision communities; the average 1-bit probability in MNIST is about 0.11.

For evenly distributed inputs the worst case is, again, when every bit to be written in the second round requires two bits and the ratio is $\frac{1}{n}\left(\frac{1}{1.5}n + \frac{1}{2}n\right) = 1.167$. However, the situation is much better for the average case.

The compression ratio is evaluated as a function of two parameters: the probabilities of a 1-bit in the input stream of the first round, p_{f}, and in the input stream of the second round, p_{s}. We need, however, also the probability $p_{\mathsf{f}}^{\mathsf{out}}$ at the *output* of the first round. The number of 1s at the output of the first round remains the same as for its input, but the expected length of the encoding has changed: by substituting a 1 by two bits and a 0 by a single bit, the expected expansion was by a factor of $1(1-p_{\mathsf{f}}) + 2p_{\mathsf{f}} = 1 + p_{\mathsf{f}}$, so we conclude that $p_{\mathsf{f}}^{\mathsf{out}} = \frac{p_{\mathsf{f}}}{1+p_{\mathsf{f}}}$, and the contribution of the first round to the compression ratio of the 2.5-ary method is $\frac{1}{1+p_{\mathsf{f}}}$.

As to the second round, two bits are overwritten for writing a 1-bit, and when writing a 0-bit in the second round, two bits are overwritten only in case 10

[3] http://yann.lecun.com/exdb/mnist/

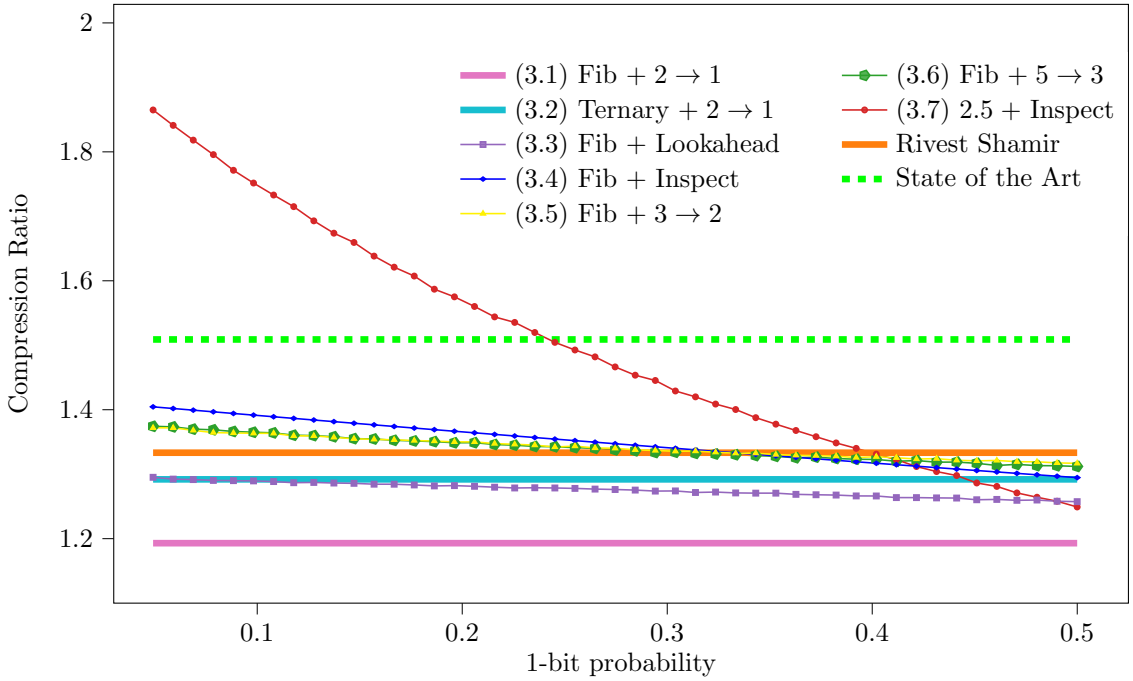

FIGURE 9. Compression performance on randomly generated data as a function of 1-bit probabilities.

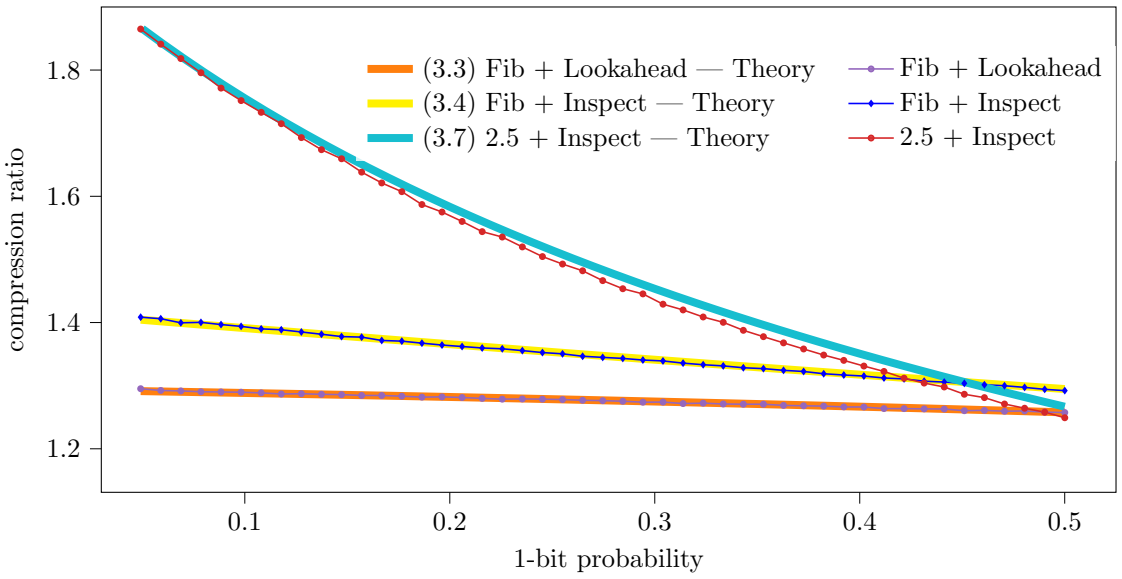

FIGURE 10. Random with Theory.

is encountered, and just a single bit otherwise. This sums up to an expected number of bits N_s written in the second round of

$$E(N_s) = 2\,p_{\mathsf{s}} + (1-p_{\mathsf{s}})\big(2p_{\mathsf{f}}^{\mathsf{out}} + 1\,(1-p_{\mathsf{f}}^{\mathsf{out}})\big)$$
$$= 2\,p_{\mathsf{s}} + (1-p_{\mathsf{s}})\left(\frac{1+2p_{\mathsf{f}}}{1+p_{\mathsf{f}}}\right).$$

The compression ratio of the 2.5-ary + Inspect method is thus $1/1+p_{\mathsf{f}} + 1/E(N_s)$. For example, if we assume random data with $p_{\mathsf{f}} = p_{\mathsf{s}} = \frac{1}{2}$, we get $\frac{2}{3} + \frac{3}{5} = 1.267$. However, if the data of both rounds is taken from the MNIST dataset with $p_{\mathsf{f}} = p_{\mathsf{s}} = 0.11$, the achieved compression ratio is 1.735.

4. EXPERIMENTAL RESULTS

We have run the following simulation tests to check our theory. For each value i, $4 \le i \le 50$, we have randomly generated 100 independent bitvectors with probability $q = \frac{i}{100}$ for the occurrence of a 1-bit. The number of bits in each of the vectors was set as $\frac{1000}{1.44} = 694$. Considering each vector as a binary number of 694 bits, each number was transformed

					3.1	3.2	3.3	3.4	3.5	3.6	3.7
	1	01	10	00	F–2–1	T–2–1	F–L	F–I	F–3–2	F–5–3	2.5–I
Theoretic	0.2764	0.2764	0.2764	0.447	1.194	1.292	1.253	1.295	–	–	1.267
Simulated	0.2759	0.2758	0.2761	0.448	1.194	1.291	1.257	1.293	1.322	1.319	1.309

TABLE 1: *Comparing theoretic and simulated probabilities and compression ratios.*

into its Fibonacci encoding, simulating a 1000 bit output of a first encoding round. We then applied the various second round encodings of the previous section, again on randomly generated data, and recorded the actual number of written bits. The numbers were then averaged for each value of q, which yields the results plotted in Figure 9 for values of $q \in [0.04, \frac{1}{2}]$. There is one plot for each of the methods presented in Sections 3.1 to 3.7, the first two of which are straight lines, since their performance does not depend on the 1-bit probability of the input. For comparison, lines indicating the methods of Rivest and Shamir (at 1.333) and the state of the art (1.509) have also been added.

Figure 10 repeats the plots of those methods, of sections 3.3, 3.4 and 3.7, for which the theoretical analysis has been given, and shows them together with the curves of the corresponding functions which appear as thicker lines. We see that for Fibonacci + Lookahead or Inspect, there is a perfect overlap, and for 2.5ary + Inspect, there is a good match.

Table 1 compares the analytically derived probabilities with the empirical occurrence probabilities on the simulated data. The first columns show the probabilities of a 1-bit, and of the pairs 00, 01 and 10, and the next columns give the performance of the seven new methods. For the methods depending on varying 1-bit probabilities, the numbers given correspond to $q = \frac{1}{2}$ for 3.3 to 3.6, and $p_f = p_s = \frac{1}{2}$ for 3.7. The similarity of the two lines of the table supports the accuracy of the model assumed in our analysis.

The performance has also been tested on some real, not randomly generated, data. We took the first bits of each of the six categories of the Pizza & Chili Corpus[4] and partitioned them into 100 blocks so as to let each block produce a string of 600 storage bits in a first round encoding. In addition, we added also a sample of the same size from the MNIST database mentioned earlier. For Fibonacci encoding, a block was of length $417 = 600/1.44$ data bits. The bits immediately following those used for the first round were then considered as the data to be written in the second round, and we counted the number of these data bits that could be encoded.

Table 2 brings the compression ratios for each of these files, as well as their average. The methods of RS-WOM and of sections 3.1 and 3.2 are omitted from the table, as they do not depend on the probability of occurrence of 1-bits. The files are sorted by decreasing 1-bit probability, which is given in the first column. We again get a very good match with the Theoretic values, obtained by plugging the average 1-bit density $q = p_f = p_s = 0.380$ into the corresponding formulæ. We see that the ratios for the dblp file, whose 1-bit probability 0.493, are close to those achieved in the simulations of Table 1, as if the data had been randomly generated.

[4] http://pizzachili.dcc.uchile.cl/texts.html

		3.3	3.4	3.5	3.6	3.7
	1-bits	F–L	F–I	F–3–2	F–5–3	2.5–I
dblp	0.493	1.257	1.292	1.318	1.308	1.252
english	0.458	1.260	1.299	1.322	1.310	1.306
pitches	0.432	1.259	1.304	1.321	1.313	1.304
proteins	0.397	1.269	1.317	1.319	1.315	1.348
sources	0.396	1.263	1.306	1.323	1.312	1.295
dna	0.374	1.270	1.325	1.326	1.325	1.378
MNIST	0.111	1.330	1.456	1.459	1.406	1.731
Average	0.380	1.272	1.328	1.341	1.327	1.373
Theoretic	0.380	1.268	1.323	–	–	1.370

TABLE 2: *Real data compression ratios.*

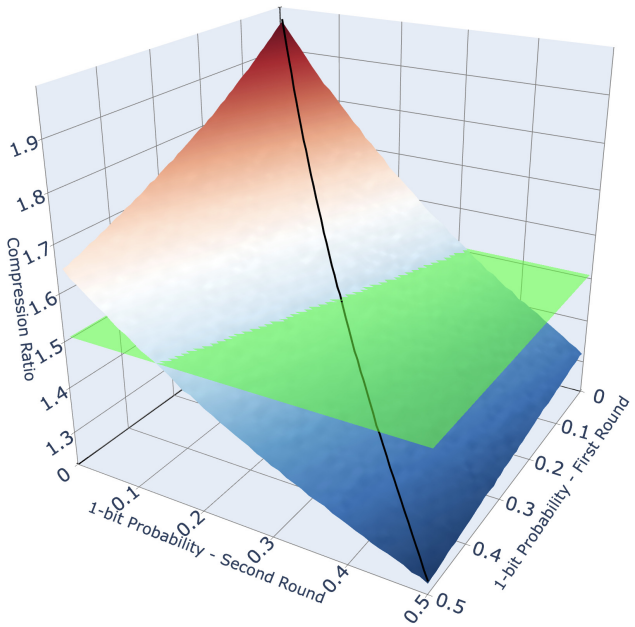

FIGURE 11. Compression ratio of 2.5-ary + Inspect as a function of the 1-bit probabilities in the first and second rounds.

Figure 11 is a 3-D plot illustrating the compression ratios for the 2.5-ary + Inspect method of Section 3.7.

To derive it, we randomly generated data with the desired probabilities for the first and second rounds and checked the resulting compression performance. Each pair of probabilities $(p_{\mathsf{f}}, p_{\mathsf{s}})$ was tested 100 times and the results were averaged. The input probabilities are given on the x- and y-axes, and the corresponding compression ratios appear according to the scale on the z-axis. The green plane corresponds to state of the art compression ratio, $z = 1.509$, and the black bold line crossing the surface is its intersection with the plane defined by $x = y$, corresponding to a scenario in which $p_{\mathsf{f}} = p_{\mathsf{s}}$, that is, the probabilities of a 1-bit are identical in both rounds.

5. CONCLUSION

We have presented several new techniques for extending context sensitive rewriting codes. Their performances are better than those of the methods of [10], but are still below the best known alternatives in the state of the art. Contrarily to other rewriting codes that are designed to yield a good compression ratio regardless of the 1-bit density of the input stream, some of the new methods presented herein take advantage of a possible non-uniformity of the input data, as may be the case for certain applications. Therefore, even though some of the compression ratios calculated above are higher than 1.509 and even than the information-theoretic upper bound of 1.585, we obviously do not claim having improved on the state of the art, since another model has been used.

It should, however, be noticed, that our challenge here is different from a situation that arises quite often in the development of new algorithms, where much effort is invested to improve a given technique, known to be currently the best. We do not try to ameliorate the performance of one of the state of the art methods, but suggest altogether different approaches. We thus see our contribution in the development of the techniques themselves. Being independent from the currently better state of the art methods, similar ideas to those we suggested may possibly lead to improved performances that could be better than the presently best known ones.

DATA AVAILABILITY STATEMENTS

The data underlying this article are available in the Pizza & Chili Corpus at http://pizzachili.dcc.uchile.cl/texts.html and in the MNIST database at http://yann.lecun.com/exdb/mnist/.

REFERENCES

[1] Gilad Baruch, Shmuel T. Klein, and Dana Shapira, "New approaches for context sensitive flash codes," in *Implementation and Application of Automata - 24th International Conference, CIAA 2019, Košice, Slovakia, July 22-25, 2019, Proceedings*, 2019, pp. 45–57.

[2] Mahmud Assar, Siamack Nemazie, and Petro Estakhri, "Flash memory mass storage architecture," 1995, US Patent 5,388,083, issued Feb. 7, 1995.

[3] Eran Gal and Sivan Toledo, "Algorithms and data structures for flash memories," *ACM Comput. Surv.*, vol. 37, no. 2, pp. 138–163, 2005.

[4] Anxiao Jiang, Vasken Bohossian, and Jehoshua Bruck, "Rewriting codes for joint information storage in flash memories," *IEEE Trans. Information Theory*, vol. 56, no. 10, pp. 5300–5313, 2010.

[5] Brian M. Kurkoski, "Rewriting codes for flash memories based upon lattices, and an example using the E8 lattice," in *IEEE Globecom Workshop on Applications of Communication Theory to Emerging Memory Technologies, ACTEMT 2010,* Miami, Florida, 6–10 December 2010. 2010, pp. 1861–1865, IEEE.

[6] Ronald L. Rivest and Adi Shamir, "How to reuse a 'write-once' memory," *Information and Control*, vol. 55, no. 1-3, pp. 1–19, 1982.

[7] Amir Shpilka, "New constructions of WOM codes using the Wozencraft ensemble," *IEEE Trans. Information Theory*, vol. 59, no. 7, pp. 4520–4529, 2013.

[8] Yeow Meng Chee, Han Mao Kiah, Alexander Vardy, and Eitan Yaakobi, "Explicit and efficient WOM codes of finite length," *IEEE Trans. Inf. Theory*, vol. 66, no. 5, pp. 2669–2682, 2020.

[9] Shmuel T. Klein and Dana Shapira, "Boosting the compression of rewriting on flash memory," in *Data Compression Conference, DCC 2014, Snowbird, UT, USA, 26-28 March, 2014*, 2014, pp. 193–202.

[10] Shmuel T. Klein and Dana Shapira, "Context sensitive rewriting codes for flash memory," *Comput. J.*, vol. 62, no. 1, pp. 20–29, 2019.

[11] Shmuel T. Klein and Miri Kopel Ben-Nissan, "On the usefulness of Fibonacci compression codes," *Comput. J.*, vol. 53, no. 6, pp. 701–716, 2010.

[12] Alberto Apostolico and Aviezri S. Fraenkel, "Robust transmission of unbounded strings using Fibonacci representations," *IEEE Trans. Information Theory*, vol. 33, no. 2, pp. 238–245, 1987.

[13] Shmuel T. Klein and Dana Shapira, "Random access to Fibonacci encoded files," *Discrete Applied Mathematics*, vol. 212, pp. 115–128, 2016.

[14] Shmuel T. Klein and Dana Shapira, "Compressed matching for feature vectors," *Theor. Comput. Sci.*, vol. 638, pp. 52–62, 2016.

[15] Shmuel T. Klein and Dana Shapira, "Compressed pattern matching in JPEG images," *Int. J. Found. Comput. Sci.*, vol. 17, no. 6, pp. 1297–1306, 2006.

[16] Ekaterina Benza, Shmuel T. Klein, and Dana Shapira, "Smaller compressed suffix arrays," to appear in *Computer Journal*, vol. 63, 2020.

[17] Edouard Zeckendorf, "Représentation des nombres naturels par une somme des nombres de Fibonacci ou de nombres de Lucas," *Bull. Soc. Roy. Sci. Liège*, vol. 41, pp. 179–182, 1972.

[18] Shmuel T. Klein, "Should one always use repeated squaring for modular exponentiation?," *Inf. Process. Lett.*, vol. 106, no. 6, pp. 232–237, 2008.