A General Method for the Development of Constrained Codes

Boris Ryabko, *Member, IEEE*

***Abstract*—Nowadays there are several classes of constrained codes intended for different applications. The following two large classes can be distinguished. The first class contains codes with local constraints; for example, the source data must be encoded by binary sequences containing no sub-words 00 and 111. The second class contains codes with global constraints; for example, the code-words must be binary sequences of certain even length where half of the symbols are zeros and half are ones. It is important to note that often the necessary codes must fulfill some requirements of both classes. In this paper we propose a general polynomial complexity method for constructing codes for both classes, as well as for combinations thereof. The proposed method uses the Cover enumerative code, but calculates all the parameters on the fly with polynomial complexity, unlike the known applications of that code which employ combinatorial formulae. The main idea of the paper is to use dynamic programming to perform calculations like: how many sequences with a given prefix and a given suffix length satisfying constraints exist. For the constraints under consideration, we do not need to know the entire prefix, but much less knowledge about the prefix is sufficient. That is, we only need a brief description of the prefix.**

***Index Terms*—Constrained codes, run-length limited codes, cover algorithm, polynomial complexity algorithm.**

I. Introduction

IN MODERN transmission and storage systems, source information is converted using data compression methods, self-correcting codes and constrained codes. The purpose of data compression and correction coding is to reduce the amount of data before transmission and to include additional data to correct the information after transmission or storage, respectively. The purpose of constrained encoding is to transform the original data before transmission to avoid all (or most) errors. Of course, all the codes mentioned above can be used together to improve performance.

Constrained codes have been intensively researched and applied in practice since the middle of the 20th century, when various hard and optical discs became widespread [1]. Nowadays, the constrained codes are used in many kinds of storage devices and numerous data transmission systems [1], [2], [3], [4] and these applications are based on profound theoretical results developed in numerous publications, see [1], [2], and [4] for reviews.

The so-called runlength-limited codes (RLL) were perhaps the first class of constrained codes [2], [3], [4]. These codes are defined via constraints on the length of runs of ones and/or zeros. In this paper we consider a somewhat more general code set that includes RLL as a subset, but we take the liberty to use the same abbreviation RLL because RLL codes are widely used in practice and are well known; at the same time, the proposed extension of this set is very simple and natural.

So, the proposed RLL class can be defined as follows: there exists a finite set of forbidden words (or patterns) $U = \{u_1, u_2, \ldots\}$ and any codeword must not contain any forbidden subwords. For example, if $u_1 = 00, u_2 = 111$, the codeword set of the three-letter constrained code is as follows: $010, 011, 101, 110$ (hence, this code can be used to encode 2-bit source words). Various RLL codes have been developed for many sets of forbidden patterns designed to meet the requirements of different storage devices and transmission systems, and there are now simply realisable and efficient algorithms for many RLL codes, see [2], [3], and [4] for a review.

Another large class of the constrained codes requires some constraints for the codeword as a whole. To describe several such codes, it will be convenient to represent the codewords over the alphabet $\{-1, +1\}$. So, the so-called balanced code is probably the first code of this type proposed by Gorog [6]. A balanced code (BC) of even length n is defined as $\left\{x_1 x_2 ... x_n : \ \sum_{i=1}^{n} x_i = 0\right\}$. Sometimes a more general problem is considered where the sum is not zero but is bounded by some (small) number α, that is, $|x_1 + \ldots + x_n| \le \alpha$. Later these codes were investigated by many researchers, and some results can be found in [2], [3], [5], [7], [8], [9], and [10] for general alphabets.

Another object of study is the limited running sum (LRS) code. By definition, for any of its codewords $x = x_1 \ldots x_n$ the current sum must satisfy $|x_1 + x_2 + \ldots + x_k| \le \delta$ for any $k = 1, 2, \ldots, n$, where $\delta > 0$ is a parameter. Note that for a codeword $x = x_1 \ldots x_n$ and any $k, l, k \le l$, we have $|x_k + x_{k+1} + ... + x_l| \le 2\delta$. So, the sum is limited for any so-called "sliding window" $x_k ....x_l$. For small δ and large n, LRS codes have a small zero frequency (dc) in the spectrum, and this property

Received 17 June 2024; revised 4 December 2024; accepted 13 March 2025. Date of publication 18 March 2025; date of current version 23 April 2025. This work was supported in part by the State Assignment of Ministry of Science and Higher Education of Russian Federation for Federal Research Center for Information and Computational Technologies and in part by the State Assignment of Siberian State University of Telecommunications and Informatics (SibSUTIS) under Grant 071-03-2024-008.

The author is with the Federal Research Center for Information and Computational Technologies, 630090 Novosibirsk, Russia, also with the Siberian State University of Telecommunications and Informatics, 630009 Novosibirsk, Russia, and also with the Novosibirsk State University, 630090 Novosibirsk, Russia (e-mail: boris@ryabko.net).

Communicated by I. Tal, Associate Editor for Coding and Decoding.
Digital Object Identifier 10.1109/TIT.2025.3552660

is very desirable for many storage devices and communication systems. This is why LRS codes have been developed by many researchers, see [1], [2], [3], and [5] for a review.

We combine the BC and LRS codes into one class by the following definition of the set of words $x_1 \dots x_n$ from the alphabet A:

$$T_n(\delta_1,\delta_2,\alpha,\beta)=\Big\{x_1\dots x_n : \delta_1 \le \sum_{i=1}^{k} x_i \le \delta_2,\ k=1,\dots,n-1 \text{ and } \alpha \le \sum_{i=1}^{n} x_i \le \beta,\ \delta_1 \le \alpha \le \beta \le \delta_2\Big\}. \quad (1)$$

Indeed, for $\alpha = \delta_1, \beta = \delta_2$ ws obtain LRS constrained codes and for $\alpha = \delta_1 = n \min_{a\in A} a$, $\beta = \delta_2 = n \max_{a\in A} a$ we obtain BC constrained codes.

The fourth class of codes with constraints is the so-called codes with energy constraints [11]. In this case, it is convenient to use the alphabet $\{0, 1\}$ and it is assumed that any 1 carries a unit of energy. (So, a sequence of letters from $\{0, 1\}$ conveys not only information but also energy.) In this problem, the density of units (i.e., the rate of energy transferred) must be bounded within some limits.

In this paper, we consider the following formal model: there exists a source of sequences in the alphabet A consisting of integers (of type $\{0, 1\}$ or $\{-1, +1\}$) and these sequences of some length n must be encoded by a certain constrained code, that is, a sequence x encoded by a codeword $c(x)$ of a constrained code such that $c(x) \neq c(y)$ if $x \neq y$. If we denote the set of all possible codewords by C, it is obvious that $|C| \ge 2^n$. (Here and below $|u|$ is the length of u if u is a word, and the number of elements if u is a set.)

In 1973, Cover proposed the so-called enumerative source coding [12], which has been widely used for constrained coding [2], [4], [13], [14]. It is worth noting that this enumerative coding was used prior to Cover's paper for the case of encoding $0-1$ words of a certain length n with a fixed number m of ones (and obviously $n - m$ zeros) [15], [16], [17].

Generally speaking, Cover's code can be used to create any constrained code, but such applications require some a priori combinatorial analysis. For example, the code for the mentioned problem about $0-1$ sequences with a fixed number of units is based on binomial coefficients and Pascal's triangle, whereas for other constrained codes the combinatorial analysis and the obtained combinatorial formulas are more complicated [4], [14].

Results are now available for some pairs of constraints, especially combining RLL with some others, and in all cases the resulting codes are based on rather complex elaborated combinatorial formulas [4], [14], [18], [19]. Apparently, the absence of known combinatorial formulas and methods of their development is a significant obstacle for construction of new constrained codes for various problems.

In this paper we propose a new approach to constructing codes with constraints in which the required parameters are computed in polynomial time and no special combinatorial analysis is required. The following is an example of a "complex" problem which can be solved by the suggested method without combinatorial analysis. We need to construct a constrained code for an n-letter sequence $x_1 \dots x_n$ of $\{0, 1\}$ for which i) the density of units is at least $1/2$, ii) the sum of units does not exceed $\lfloor 2n/3 \rfloor$, iii) $|\sum_{i=1}^{k} x_i - k/2| \le 20$ for $k = 1, \dots, n$ and iv) the words 0011 and 01010 should be excluded. (Thus, this problem includes all four constraints mentioned above.)

The rest of the paper consists of the following. Part 2 contains the description of the Cover method. The part 3 contains description of the limited running sum (LRS) code. Parts 4 and 5 contain description of codes with constraints for the two other problems described above. Part 6 describes codes for the combinations of constraints described above and studied in Parts 3 and 5. In Conclusion we talk about the complexity of the proposed algorithms, some simplifications and generalisations of the described codes, as well as the scope of the proposed method.

II. The Cover Method

In [12] Cover suggested the following general method of an enumerative encoding. There is an m-letter alphabet $A = \{a_1, a_2, \dots, a_m\}$ and let A^n be a set of words of length n over A. Every subset $S \subset A^n$ is called a source. An enumerative code f is given by two mappings $f_c : S \to \{0, 1\}^l$, where $l = \lceil \log |S| \rceil$ and $f_d : f_c(S) \to S$, so that $f_d(f_c(s)) = s$ for all $s \in S$. The map f_c is called an encoder and f_d is called a decoder. It is assumed that the alphabet A consists of numbers. It should be noted that in some sense the names "encoder" and "decoder" are interchangeable. The definition given above is often used in some data compression tasks, while the opposite definition is natural for some others.

Let us describe an enumerative code from [12].

Encoder: Let $N(x_1 \dots x_k)$ be the number of words which belong to S and have the prefix $x_1 \dots x_k$, $k = 1, 2, \dots, n-1$. For $x_1 x_2 \dots x_n \in S$ define the code word $f_c(x_1 \dots x_n)$ by

$$code(x_1 \dots x_n) = \sum_{i=1}^{n} \sum_{a<x_i} N(x_1 \dots x_{i-1} a). \quad (2)$$

Decoder: Let us describe the decoder. Denote $\alpha = code(x_1 \dots x_n)$, and $b_{m+1} = N(\epsilon) + 1$, where $N(\epsilon) = \sum_{a\in A} N(a) (= |S|)$ (ϵ is the common representation of the empty string). Then calculate $b_1 = N(a_1), b_2 = N(a_2), \dots, b_m = N(a_m)$. If $\sum_{s=1}^{j} b_s \le \alpha < \sum_{s=1}^{j+1} b_s$ then the first letter x_1 is a_j. In order to find x_2 the algorithm calculates $\alpha = \alpha - N(a_j), b_j = N(x_1 a_j), j = 1, \dots, m$. If $\sum_{s=1}^{k} b_s \le \alpha < \sum_{s=1}^{k+1} b_s$ then the second letter x_2 is a_k, and so on.

Note that the complexity of the Cover method is determined by the complexity of computing $N(\,)$, and hence, developing a simple method for computing $N(\,)$ is the main problem. Currently, the use of Cover's method is based on combinatorial formulas to compute $N(\,)$, as shown in the following example of encoding binary words of n-letters with a given number of units ν. That is, the alphabet $A = \{0, 1\}$, set S contains words for which $\sum_{k=1}^{n} x_i = \nu$. In this case

$$N(x_1 \dots x_{k-1} 0) = \binom{n-k}{\nu - \sum_{i=1}^{k-1} x_i}. \quad (3)$$

Using this formula and (2) we obtain

$$code(x_1 \dots x_n) = \sum_{k=1}^{n} x_k \binom{n-k}{\nu - \sum_{i=1}^{k-1} x_i}.$$

(This solution was found before the Cover code was described, see [15], [16], [17]). The set of binomial coefficients (3) can be stored in memory or computed on the fly as needed.

Nowadays combinatorial formulas for many codes with constraints are known and widely used in practice. But sometimes such formulas can be very complicated, and for some interesting problems these formulas are not developed at all.

In this paper we propose a direct computation of the sums $N(\,)$, not based on any combinatorial formulae. These $N(\,)$ can then be stored in memory or computed on the fly. More precisely, we describe polynomial methods for computing the values of $N(\,)$ for the various constrained codes discussed in the introduction.

III. Polynomial Complexity Method for BC and LRS Codes

In this part we consider the set $T_n(\delta_1, \delta_2, \alpha, \beta)$ of sequences $x_1 \dots x_n$, x_i over A, see (1). It is worth noting that the size of this set grows exponentially. For example, for $A = \{-1, 1\}$, $|T_n(-1, 1, -1, 1)| = 2^{\lceil n/2 \rceil}$. So, our goal is to develop a simple method for calculating $N(x_1 \dots x_r)$ for $r = 1, \dots, n$ for $x_1 \dots x_n \in T_n(\delta_1, \delta_2, \alpha, \beta)$, because this is a key part of the Cover encoding and decoding method (described in the previous section).

Algorithm 1 Computation of N

```
Input: x_1 ... x_r
Output: N(x_1 ... x_r)
Create table S[i, j], i = 1, ..., n − r + 1, j = δ_1, ..., δ_2
Fill S with zeros
z_r ← Σ_{i=1}^{r} x_i
S[1, z_r] ← 1
for i ← 1 to n − r do
    for k ← δ_1 to δ_2 do
        for a ∈ A do
            if δ_1 ≤ k + a ≤ δ_2 then State S[i + 1, k + a] ← S[i + 1, k + a] + S[i, k]
            end if
        end for
    end for
end for
N ← 0
for i ← α to β do
    N ← N + S[n − r + 1, i]
end for
```

First, let us give some comments to make the description of the algorithm (denoted as Algorithm 1) clear. The key observation is the following: the number of trajectories with the prefix $x_1 \dots x_k$ depends only on the sum $x_1 + \dots + x_k$ and hence the algorithm does not need to store the word $x_1 \dots x_k$. This reduces the required memory from exponential to polynomial. The algorithm computes the number of trajectories (or words) that start with $z_r = \sum_{i=1}^{r} x_i$. The table $S[i, j], i = 1, \dots, n-r+1, j = \delta_1, \dots, \delta_2$ contains the numbers of valid trajectories of type $z_r x_{r+1} \dots x_{r+i}$, $i = 1, \dots, n-r$, $j \in \{\delta_1, \delta_1 + 1, \dots, \delta_2\}$. In general, when the algorithm goes from $r+i$ to $r+i+1$, we must extend all current trajectories by the selected letter $a \in A$ and update the table $S[,]$ with the revised number of trajectories

Theorem 1: Algorithm 1 correctly computes the number of words from $T_n(\delta_1, \delta_2, \alpha, \beta)$ with the prefix $x_1 \dots x_r, r < n$, that is $N(x_1 \dots x_r)$.

Proof: The algorithm sequentially computes the numbers of words $N(x_1 \dots x_k a)$ when $N(x_1 \dots x_k)$ is given, considering all words for which $\delta_1 \le \sum_1^i x_i \le \delta_2$, i.o. in the set $T_i(\delta_1, \delta_2, \alpha, \beta)$, (here $x_1, \dots, x_k, a \in A$, $i = r, r+1, \dots, n$). Then, on line 18 $N(x_1 \dots x_r)$ is computed for α and β.■

Let us consider a small example. Let $A = \{-1, +1\}$, $n = 6$, $x \in T_n(\delta_1, \delta_2, \alpha, \beta)$, where $\delta_1 = 0, \delta_2 = 3, \alpha = 0, \beta = 2$ and let the algorithm 1 be applied for calculation $N(+1\ -1\ +1)$. Then $z_3 = 1$, and the algorithm is carried out as follows:

$$S[1,\ ] = \begin{pmatrix} 0 \\ 0 \\ 1 \\ 0 \end{pmatrix} S[2,\ ] = \begin{pmatrix} 0 \\ 1 \\ 0 \\ 1 \end{pmatrix}$$

$$S[3,\ ] = \begin{pmatrix} 1 \\ 0 \\ 2 \\ 0 \end{pmatrix} S[4,\ ] = \begin{pmatrix} 0 \\ 3 \\ 0 \\ 2 \end{pmatrix}$$

Applying the last algorithm cycle from $\alpha = 0$ till $\beta = 2$ we obtain $N = 2 + 0 + 3 = 5$ and, hence, $N(+1\ -1\ +1) = 5$.

Now let us estimate the complexity of the Cover method of encoding and decoding if $N(x_1 \dots x_i)$ are calculated by the algorithm described above. During decoding and encoding the values $N(x_1 \dots x_i)$ are calculated and for any $N(\,)$ the summation $O(n)$ is required. So, taking into account that the length of $N(\,)$ is up to $O(n)$ we see that the total time (for $i = 1, \dots, n-1$ in (2)) is $O(n^3(\delta_2 - \delta_1)) + (\beta - \alpha))$. The required memory size for $S[,]$ is $O(n^2)$ cells of the length $O(n)$, but it is sufficient to store only two columns of the table S, and hence the required memory is $O(n^2)$. Taking into account that $N(x_1 \dots x_k)$ depends only on $\sum_{i=1}^{k} x_i$, we see that there are only $(\delta_2 - \delta_1 + 1)$ of different $N(\,)$ and all of them can be calculated in advance and stored in $O((\delta_2 - \delta_1)n^2)$ cells.

IV. Energy Constraints

In [11] the so-called subblock energy-constrained codes (SECC) and sliding window codes (SWCC) are considered for the $\{0, 1\}$ alphabet. To describe them, it will be convenient to call the number of units in the word u as the weight u and denote it by $w(u)$.

SECC are binary sequences of length $n = ml$, which are treated as a sequence of m l-bit sub-blocks (m and l are integers, and in what follows, we consider the case where l is a constant independent of n). The weight of each subblock is at least α and at most β, where α and $\beta, \alpha < \beta$, are some integers. The set of sequences belonging to the same block is $T_l(\alpha, \beta, \alpha, \beta)$ (see (1)), and hence the algorithm presented in the previous section can be applied.

Sliding window constrained codes are n-bit words $x_1 \dots x_n$ such that the weight of any l-bit subword $x_{i+1} \dots x_{i+l}$ is at least α and at most β, i.e. $\alpha \le w(x_{i+1} \dots x_{i+l}) \le \beta$ (here i can be any integer from $[0, n-l]$, not necessarily a multiple of l).

So, the following problem of encoding is considered. There is a set of sequences $x_1, \dots, x_n$, $x_i \in \{0,1\}$ and $\alpha \le \sum_{i=k}^{k+l-1} x_i \le \beta$ for any integer $k \in [1, \dots, n-l+1]$ and let denote the set of such words by $SWCC_n(l, \alpha, \beta)$.

A. The Algorithm for Sliding Window-Constrained Codes

To simplify the description of the algorithm, we will use an additional letter $\emptyset$ and hence the alphabet will be $\{\emptyset, 0, 1\}$. We also assume that the weight of $\emptyset$ is 0. The algorithm uses the table $S[i,u]$, where i is an integer, $i = 1, \dots, n-l+1$ and u is a word, $u \in \{\emptyset, 0, 1\}^l$. (This table will be used to store the number of trajectories.) Also we denote by $v(v)$ the number of the letters $\emptyset$ in the word v. Note that the letter $\emptyset$ is needed only for the case $r < l$.

Algorithm 2 Computation of N

1: **Input:** $x_1 \dots x_r$. Comment: $x_1 \dots x_r \in SWCC_r(l, \alpha, \beta)$. Otherwise, $N = 0$.
2: **Output:** $N(x_1 \dots x_r)$
3: **for** $i \leftarrow -l+1$ to 0 **do**
4: $x_i \leftarrow \emptyset$
5: **end for**
6: Create table $S[i,u]$, $i = 1, \dots, n-r+1$, $u \in \{\emptyset, 0, 1\}^l$
7: Fill S with zeros
8: $S[1, x_{r-l+1} x_{r-l+2} \dots x_r] \leftarrow 1$
9: **for** $i \leftarrow 1$ to $n-r$ **do**
10: **for** $u \in \{\emptyset, 0, 1\}^l$ **do**
11: $v \leftarrow u_2 u_3 \dots, u_{l-1}$
12: **if** $w(v0) \ge \alpha - v(v0)$ **then**
13: $S[i+1, v0] \leftarrow S[i+1, v0] + S[i,u]$
14: **end if**
15: **if** $w(v1) \le \beta$ **then**
16: $S[i+1, v1] \leftarrow S[i+1, v1] + S[i,u]$
17: **end if**
18: **end for**
19: **end for**
20: $N \leftarrow 0$
21: **for** $u \in \{0,1\}^l$ **do**
22: $N \leftarrow N + S[n-r, u]$
23: **end for**

Theorem 2: Algorithm 2 correctly computes $N(x_1, \dots, x_r)$ $(r < n)$, that is, the number of words with prefix $x_1, \dots, x_r$ from the set $SWCC_n(l, \alpha, \beta)$.

Proof: When the window is "moved", the algorithm sequentially counts the number of words in $SWCC_i(l, \alpha, \beta)$, with the prefix $x_1, \dots, x_r$, $i = r, r+1, \dots, n$, such that all words in $SWCC_i(\alpha, \beta)$ are counted. Then $N(x_1 \dots x_r)$ is computed in line 21.■

Let us estimate the complexity of this algorithm. Table $S[,]$ contains $3^l(n+l)$ cells and the computation time is proportional to the same value. It is clear that this algorithm can be significantly simplified if $\emptyset$ is removed, but the asymptotic complexity estimate will be the same, so we will not describe these simplifications.

V. The Method for RLL Codes

Let codewords of an RLL code be n-letter words over an alphabet $A = \{a_1, \dots, a_m\}, m \ge 2$, and $V = \{v_1, \dots, v_s\}, s \ge 1$, be the finite set of forbidden words, that is, any codeword $x_1 \dots x_n$ does not contain any $v \in V$ as a subword. Denote the set of such words by $RLL_n(V)$. (The set V is assumed to be independent of n.) As before, we will use Cover's method, and the description of the algorithm will be reduced to computing the numbers $N(x_1 \dots x_r)$, see the description in part 2. It is worth noting that the described algorithm will be very close to the algorithm for sliding window codes. In particular, we extend the alphabet $A = \{a_1, \dots, a_m\}$ to $A' = \{\emptyset, a_1, \dots, a_m\}$.

A. The Algorithm for RLL Codes

Let us define

$$l_v = |v|, \mu = \max_{v \in V} l_v. \tag{4}$$

The algorithm uses the table $S[i,u]$, where i is an integer, $i = 0, 1, \dots, n-r+1$ and u is a word, $u \in A'^{\mu}$.

Algorithm 3 Computation of N

1: **Input:** $x_1 \dots x_r$. Comment: $x_1 \dots x_r$ does not contain subwords from V. Otherwise, $N = 0$.
2: **Output:** $N(x_1 \dots x_r)$
3: **for** $i \leftarrow -\mu+1$ to 0 **do**
4: $x_i \leftarrow \emptyset$
5: **end for**
6: Create table $S[i,u]$, $i = 1, \dots, n-r+1$, $u \in A'^{\mu}$
7: Fill S with zeros
8: $S[1, x_{r-\mu+1} x_{r-\mu+2} \dots x_r] \leftarrow 1$
9: **for** $i \leftarrow 1$ to $n-r$ **do**
10: **for** $u \in A'^{\mu}$ **do**
11: **for** $a \in A'$ **do**
12: $\lambda \leftarrow$ **true**, $w \leftarrow u_2 u_3 \dots u_\mu a$
13: **for** $v \in V$ **do**
14: **if** $w_{\mu - l_v + 1} \dots w_\mu = v$ **then**
15: $\lambda \leftarrow$ **false**,
16: **end if**
17: **end for**
18: **if** $\lambda =$ **true then**
19: $S[i+1, w] \leftarrow S[i+1, w] + S[i,u]$
20: **end if**
21: **end for**
22: **end for**
23: **end for**
24: $N \leftarrow 0$
25: **for** $u \in A^{\mu}$ **do**
26: $N \leftarrow N + S[n-r+1, u]$
27: **end for**

Theorem 3: Algorithm 3 correctly computes $N(x_1, \dots, x_r)$ $(r < n)$, that is, the number of words with prefix $x_1, \dots, x_r$ from the set $RLL_n(V)$.

Proof: When the window is "moved" (in line 12), the algorithm sequentially counts the number of words in $RLL_i(V)$,

with the prefix $x_1, \ldots, x_r$, $i = r, r+1, \ldots, n$, such that all words in $RLL_i(V)$ are counted. Then $N(x_1 \ldots x_r)$ is computed in line 25. ∎

VI. Multiple Constraints

Quite often certain data transmission and storage systems require codes satisfying multiple constraints.Generally speaking, sometimes such problems can be solved by performing special combinatorial studies to find expressions for $N(\,)$ in the Cover method, see, for example, [2], Chapter 6, [4], [13], [14], [18], [19]. In this part, we show that the described methods of directly computing the values of $N(\,)$ can be easily used in the case of joint application of multiple constraints.

As an example, we consider the case where constraints on the sums of sequence letters and RLL constraints are given together. All other possible combinations of constraints (e.g., energy and RLL) are similar. So we consider the following set $T_n(\delta_1, \delta_2, \alpha, \beta)$, $\delta_1 \le \alpha \le \beta \le \delta_2$, of sequences $x_1 \ldots x_n$, x_i from some alphabet A, see (1). Also there is a set with RLL constrains, that is a set of forbidden words $V = \{v_1, \ldots, v_s\}$, $s \ge 1$, and any codeword $x_1 \ldots x_n$ does not contain any $v \in V$ as a subword. (This set was denoted by $RLL_n(V)$). So our goal is to build the constrained code for the set of sequences $\Delta = T_n(\delta_1, \delta_2, \alpha, \beta) \cap RLL_n(V)$. As before, we will use the Cover method and therefore only describe the computation of the numbers $N(x_1 \ldots x_r)$. To do this, we combine Algorithm 1 and Algorithm 3 as follows.

The algorithm is designed to compute the number of trajectories (or words) that belong to Δ. We consider the table $S[i, j, u]$, where $i = 1, \ldots, n-r$, $j = \delta_1, \ldots, \delta_2$, $u \in A'^{\mu}$, (μ defined in (4)), $A' = \{\emptyset, a_1, \ldots, a_m\}$.

This algorithm is a combination of Algorithms 1 and 3. Its correctness follows from Theorems 1 and 3.

Algorithm 4 Computation of N

```
1:  Input: x_1 ... x_r, Output: N(x_1 ... x_r)
2:  Create table S[i, j, u], i = 1, ..., n − r, j = δ_1, ..., δ_2,
    u ∈ A'^μ
3:  Fill S with zeros
4:  for i ← −μ + 1 to 0 do
5:      x_i ← ∅
6:  end for
7:  z_r ← Σ_{i=1}^{r} x_i,  S[1, z_r, x_{r−μ+1} x_{r−μ+2} ... x_r] ← 1
8:  for i ← 1 to n − r + 1 do
9:      for j ← δ_1 to δ_2 do
10:         for a ∈ A' do
11:             if δ_1 ≤ j + a ≤ δ_2 then
12:                 λ_1 ← true
13:             else
14:                 λ_1 ← false
15:             end if
16:             λ_2 ← true,  w ← u_2 u_3 ... u_μ a
17:             for v ∈ V do
18:                 if w_{μ−l_v+1} ... w_μ = v then
19:                     λ_2 ← false
20:                 end if
21:             end for
22:             if λ_1 & λ_2 = true then
23:                 S[i+1, j+a, w] ← S[i+1, j+a, w] + S[i, j, u]
24:             end if
25:         end for
26:     end for
27: end for
28: N ← 0
29: for j ← α to β do
30:     for u ∈ A^μ do
31:         N ← N + S[n − r + 1, j, u]
32:     end for
33: end for
```

VII. Conclusion

In this part we first will briefly evaluate the complexity of the developed codes. First of all, we note that there are many obvious simplifications of the described methods. Thus, in all algorithms the table $S[i, j]$ can contain only two rows corresponding to i, the third and fourth algorithms can be described without an extra letter $\emptyset$, etc. This was done to avoid unnecessary small details and hence make the description shorter and clearer.

Next, we briefly discuss asymptotic estimates of the memory and time required for the developed algorithms (2). First of all, we note that the Cover method requires $O(n)$ summation operations with $O(n)$ integers of $O(n)$-bit length (because, as we showed in the introduction, the number of trajectories grows exponentially even in the simplest cases). So, the running time and memory size of the Cover method is $O(n^2)$.

Now let us evaluate the complexity of computing the values of $N(\,)$ used in the algorithms described above, but first note that the following two ways of using these values are possible: either the set of values $N(\,)$ can be calculated in advance, stored and used several times for encoding and decoding different $x_1 \ldots x_n$, or $N(\,)$ can be computed again for any $x_1 \ldots x_n$. The upper estimate can be obtained by assuming that δ_1 and δ_2 in the algorithms are equal to $O(n)$. In this case, the computation time of $N(\,)$ values is $O(n^3)$ and the memory size is $O(n^2)$ bits, assuming that the length of $N(\,)$ is $O(n)$ bits.

The developed algorithms are not symmetric in the following sense: If, for example, $N(x_1 \ldots x_n)$ is computed, then in all described algorithms the length of summands increases from a few bits to $O(n)$ bits. The work [20] describes an algorithm for the Cover method, where most of the operations are performed on numbers of the same length, which significantly reduces the complexity. It seems natural to reorganise the algorithms developed above in a similar way in order to reduce the complexity, but for this purpose a special new algorithm must be developed.

Next we discuss the scope of the described method. The key part of the Cover method is the computation of the number of words with prefix $x_1 \ldots x_k a$ if the number of words with prefix $x_1 \ldots x_k$ is known, that is, the computation of $N(x_1 \ldots x_k a)$ if $N(x_1 \ldots x_k)$ is given (here $x_1, \ldots, x_k, a \in A$), see (2). In all the constrained code problems we have considered, we do not store $N(x_1 \ldots x_k)$ for all words $x_1 \ldots x_k$. Instead, when we compute $N(x_1 \ldots x_k a)$, we do not use the entire $x_1 \ldots x_k$ prefix,

but only use the knowledge of a small summary of the prefix. This is a key observation, since the required memory size then decreases from exponential ($|A|^k$) to at most polynomial. (Thus, for constraints BC and RLC and $A = \{0, 1\}$, it is sufficient to store one $N(x_1 \dots x_k)$ for all words $x_1 \dots x_k$ of the same weight.)

The rest of the computations are based on dynamic programming.

A natural question is how many constrained codes can be realized by a similarly simple method. Let us consider this question informally. Clearly, there are 2^{2^n} subsets of $\{0, 1\}^n$. The Kolmogorov complexity of almost all those subsets will be $2^n(1 + o(1))$ according to the uniform distribution on the subsets (see [21], [22]). In particular, this means that the size of the program that would implement the Cover method must be at least $2^n(1 + o(1))$ for almost all these sets i.e., it grows exponentially with n.

On the other hand, the above applications of the method show that it works for many constrained sets if, informally speaking, the description of these constraints is in some sense short and simple. Perhaps this observation can be useful in practice, but a formal treatment of this problem belongs to the theory of complexity of algorithms.

REFERENCES

[1] K. A. S. Immink, "Innovation in constrained codes," *IEEE Commun. Mag.*, vol. 60, no. 10, pp. 20–24, Oct. 2022.

[2] K. A. S. Immink, *Codes for Mass Data Storage Systems*. Denver, CO, USA: Shannon Foundation, 2004.

[3] B. H. Marcus, R. M. Roth, and P. H. Siegel, "An introduction to coding for constrained systems," Lect. Notes, Oct. 2001.

[4] A. Hareedy, B. Dabak, and R. Calderbank, "The secret arithmetic of patterns: A general method for designing constrained codes based on lexicographic indexing," *IEEE Trans. Inf. Theory*, vol. 68, no. 9, pp. 5747–5778, Sep. 2022.

[5] D. Bar-Lev, A. Kobovich, O. Leitersdorf, and E. Yaakobi, "Optimal almost-balanced sequences," 2024, *arXiv:2405.08625*.

[6] E. Gorog, "Redundant alphabets with desirable frequency spectrum properties," *IBM J. Res. Develop.*, vol. 12, no. 3, pp. 234–241, May 1968.

[7] N. Alon, E. E. Bergmann, D. Coppersmith, and A. M. Odlyzko, "Balancing sets of vectors," *IEEE Trans. Inf. Theory*, vol. 34, no. 1, pp. 128–130, Jan. 2010.

[8] D. Knuth, "Efficient balanced codes," *IEEE Trans. Inf. Theory*, vol. IT-32, no. 1, pp. 51–53, Jan. 1986.

[9] K. A. S. Immink and J. H. Weber, "Very efficient balanced codes," *IEEE J. Sel. Areas Commun.*, vol. 28, no. 2, pp. 188–192, Feb. 2010.

[10] J. H. Weber, K. A. Schouhamer Immink, P. H. Siegel, and T. G. Swart, "Perspectives on balanced sequences," 2013, *arXiv:1301.6484*.

[11] T. T. Nguyen, K. Cai, and K. A. S. Immink, "Efficient design of subblock energy-constrained codes and sliding window-constrained codes," *IEEE Trans. Inf. Theory*, vol. 67, no. 12, pp. 7914–7924, Dec. 2021.

[12] T. Cover, "Enumerative source encoding," *IEEE Trans. Inf. Theory*, vol. IT-19, no. 1, pp. 73–77, Jan. 1973.

[13] A. Hareedy and R. Calderbank, "LOCO codes: Lexicographically-ordered constrained codes," *IEEE Trans. Inf. Theory*, vol. 66, no. 6, pp. 3572–3589, Jun. 2020.

[14] O. F. Kurmaev, "Constant-weight and constant-charge binary run-length limited codes," *IEEE Trans. Inf. Theory*, vol. 57, no. 7, pp. 4497–4515, Jul. 2011.

[15] T. J. Lynch, "Sequence time coding for data compression," *Proc. IEEE*, vol. 54, no. 10, pp. 1490–1491, 1966.

[16] L. D. Davisson, "Comments on 'Sequence time coding for data compression,"' *Proc. IEEE*, vol. 54, no. 12, p. 2010, Dec. 1966.

[17] V. F. Babkin, "A method of universal coding with non-exponent labour consumption," *Probl. Inform. Transmiss.*, vol. 7, pp. 13–21, Jun. 1971.

[18] O. Ytrehus, "Upper bounds on error-correcting runlength-limited block codes," *IEEE Trans. Inf. Theory*, vol. 37, no. 3, pp. 941–945, May 1991.

[19] O. F. Kurmaev, "Enumerative coding for constant-weight binary sequences with constrained run-length of zeros," *Problems Inf. Transmiss.*, vol. 38, no. 4, Oct. 2002, Art. no. 249254.

[20] B. Ryabko, "The fast enumeration of combinatorial objects," 1998, *arXiv: cs/0601069*.

[21] T. M. Cover and J. A. Thomas, *Elements of Information Theory*. New York, NY, USA: Wiley, 2006.

[22] M. Li and P. Vitnyi, *An Introduction to Kolmogorov Complexity and Its Applications*. New York, NY, USA: Springer, Nov. 5, 2008.

Boris Ryabko (Member, IEEE) has published about 200 articles on information theory and its applications to cryptography, mathematical statistics, biology, and linguistics. In 1979, he proved a theorem on the equivalence of channel capacity and redundancy of a universal code. In 1980, he discovered the "book stack" code, rediscovered in 1986 by Bentley, Slaytor, Tarjan, and Wei under the name "move-to-front transform." In 1986, he discovered a logical connection between Hausdorff dimensionality and Kolmogorov complexity. He also discovered optimal universal codes for various classes of sources, including the "adaptive binary tree" data structure (published in IEEE TRANSACTIONS ON INFORMATION THEORY in 1992) and later in 1994 published by Fenwick (now often called Fenwick tree or binary indexed tree). In the 1980's, biologist Zhanna Reznikova and he published several articles in which they experimentally proved that some species of ants can transmit up to six bits of information and can add and subtract small numbers using an abstract language (not chemical tracks). The experiments are based on ideas from information theory.