Introducing loop compression for encoding de Bruijn sequences

Oscar Cabrera

ocg.steppenwolf@gmail.com

Independent Researcher

March 2025

Abstract

In order to compress de Bruijn sequences of length N over any alphabet of size k, a method is developed (*Loop Compression*) using an efficient algorithm with linear time and space complexity.

1 Introduction

Given a de Bruijn sequence S over an alphabet of size k and length $N = k^n$, and knowing that traditional compression methods are unable to compress S, may be interesting to design an efficient algorithm to compress such sequences. First of all, let's remember that a de Bruijn sequence of order n over an alphabet of size k is a cyclic sequence in which every possible string of length n occurs exactly once as a substring. Some examples of binary de Bruijn sequences are: 01, 0011, 00010111, 0000100110101111... There are many known ways to create de Bruijn sequences [1]. Initially, it may seem shocking that this type of sequences can be compressed, but to see that it is not only possible but feasible to find an efficient method to compress them, we will explain in detail how we got there.

We already know the number of total (i.e. not unique) de Bruijn sequences of length N:

$$c_{db}(k,n) = (k!)^{k^{n-1}} = (k!)^{\frac{N}{k}} \tag{1}$$

We can imagine a list with all de Bruijn sequences, as a subset of all sequences of N symbols, with the goal of achieving the maximum possible average compression for that list. As an example, let's analyze the case $N = 2^2$ shown in Table 1, with one unique de Bruijn sequence and four in total.

Table 1: Compression methods for $N = 2^2$

Decimal index	Sequence	Constant compression	MSS compression	Optimal encoding
0	0011	00	-/0	-
1	0110	01	1	0
2	1001	10	10	1
3	1100	11	11	00

Basically, we would have three possibilities to compress each de Bruijn sequence:

- Constant compression: we encode each sequence using the number of symbols needed to encode the last sequence, this way we get a constant compression rate.
  In our example, only two symbols are needed.
- MSS compression: we encode each sequence using the number of symbols needed to encode its respective index (i.e. until the Most Significant Symbol).
  In our example, we can encode the first sequence using either a zero or nothing.
- Optimal encoding: we optimally encode the index associated to each sequence, that is to say, taking advantage of every encoding of size zero and up.
  In our example, we can encode using every encoding of zero and one symbols, and with one encoding of two symbols.

The third possibility is the best one, but as it improves the average compression almost insignificantly respecting constant compression, we won't enter in details and we will focus on the nice constant compression as a good reference point for our analysis. In fact, as in principle we will treat all de Bruijn sequences in the same way (equiprobability of sequences), it makes sense to encode each sequence in the necessary number of symbols to express all of them, instead of encoding the most probable sequences with fewer symbols.
Then, using constant compression, the number of symbols needed to express any de Bruijn sequence belonging to the subset of c_{db} sequences is:

$$s_{db}(k,n,b) = H + \lceil \log_b(c_{db}) \rceil = H + \left\lceil \log_b\left((k!)^{\frac{N}{k}}\right) \right\rceil = H + \left\lceil \frac{N}{k} \log_b(k!) \right\rceil \tag{2}$$

In the previous expression, b is the size of the output alphabet (in practice, common values will be k, 2 or 256) and H is the number of symbols (usually a constant value) using that output alphabet to store the header.
Since our sequences have $N = k^n$ symbols over an alphabet of size k, we can calculate the ratio between the number of necessary symbols and the original number of symbols N, and therefore, subtracting it from 1, also the constant compression rate:

$$r_{db}(k,n,b) = 1 - \frac{s_{db}}{\lceil \log_b(k^N) \rceil} = 1 - \frac{H + \left\lceil \frac{N}{k} \log_b(k!) \right\rceil}{\lceil N \log_b(k) \rceil} \tag{3}$$

Let us remember again that (3) will be the value that we will assume as a reference, but it is not the maximum possible value for the mean compression rate.

So far everything is fine, but in order to reach the constant compression rate seen in (3) we need to be able to associate a unique identifier $i \in [0 \,..\, c_{db} - 1]$ to each de Bruijn sequence, expressed in $\lceil \log_b(c_{db}) \rceil$ symbols using an alphabet of size b. We have two alternatives:

1. Associate the identifier using brute force: in general this is possible but not realistic, so it is discarded due to exponential time complexity.
2. Indexing: it was evaluated in a previous investigation [2]; here we are interested in *full indexing*, and although it is better than using brute force, it is not very efficient either. *Partial indexing* is useless here.

We want an efficient method with linear time complexity $O(N)$, so we will have to think of a smarter way.

2 Loop compression

2.1 Algorithm

The proposed algorithm has been developed as part of the investigation under *Myshella*, which is a personal project with a broad and idealistic objective: to bring order into chaos.

Well, once we have discarded the use of indexing, we have to find an alternative and efficient method to associate a unique identifier to each de Bruijn sequence, even though this means not reaching our reference mean compression rate seen in (3). Maybe we can deduce something by looking at expression (1): it tells us that we could associate for each de Bruijn sequence a combination of $\frac{N}{k}$ symbols over an alphabet of size $k!$.
The method presented in this text has been called *Loop Compression*, and consists of encoding the sequences using something we will call *buckets*, inspired by hash tables. Let's see how to manage these buckets.

Definition 2.1. *A bucket is a black box with k inputs, k outputs and $k!$ possible configurations for the relations between input and output codes $I_i \to O_j$; the real configuration for a bucket t will be given by its associated value T_t (Figure 1).*

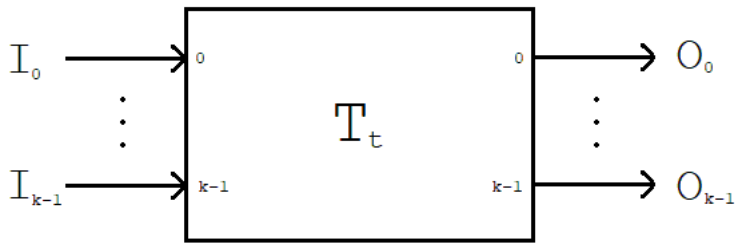

Figure 1: Black box for bucket t

Theorem 2.1. *Given a sequence of length $N = k^n$, there are $\frac{N}{k}$ buckets in total.*

Proof. Let's think about what happens when we have a code of n symbols read from a de Bruijn sequence S, and we are going to read the next symbol:

1. Remove the least significant symbol from the code by shifting the code one symbol (in other words: dividing the code by k).
2. Append the new read symbol in the most significant position of the new code.

It means that, as the removed symbol doesn't count, we have k^{n-1} possible values for the intermediate code before we append the most significant symbol. This intermediate value will be the bucket identifier t, so we will have $\frac{N}{k}$ buckets in total. □

Note that the total number of combinations of bucket values T_t, taking into account all buckets, is like having $\frac{N}{k}$ symbols in base $k!$, which is equal to (1). An important observation is that not all of these combinations generate valid de Bruijn sequences in base k, but only one in N does.

Proposition 2.1. *Given an input code $I_i \in [0\,..\,N-1]$ of n symbols over an alphabet of size k, where $i \in [0\,..\,k-1]$, the bucket index $t \in [0\,..\,\frac{N}{k}-1]$ is:*

$$t = \frac{I_i}{k} \tag{4}$$

Proposition 2.2. *Given a bucket t (Figure 1), the input and output codes will have the following values:*

$$\left.\begin{aligned} I_0 &= t \cdot k + 0 \\ I_1 &= t \cdot k + 1 \\ &\dots \\ I_{k-1} &= t \cdot k + (k-1) \end{aligned}\right\} \Rightarrow \begin{cases} O_0 = t + 0 \cdot \frac{N}{k} \\ O_1 = t + 1 \cdot \frac{N}{k} \\ \dots \\ O_{k-1} = t + (k-1) \cdot \frac{N}{k} \end{cases} \tag{5}$$

For instance, for $N = 2^3$ we get:

$$\left.\begin{aligned} t = 0 : I_0 &= 0 \\ I_1 &= 1 \end{aligned}\right\} \Rightarrow \begin{cases} O_0 = 0 \\ O_1 = 4 \end{cases} \qquad \left.\begin{aligned} t = 1 : I_0 &= 2 \\ I_1 &= 3 \end{aligned}\right\} \Rightarrow \begin{cases} O_0 = 1 \\ O_1 = 5 \end{cases}$$

$$\left.\begin{aligned} t = 2 : I_0 &= 4 \\ I_1 &= 5 \end{aligned}\right\} \Rightarrow \begin{cases} O_0 = 2 \\ O_1 = 6 \end{cases} \qquad \left.\begin{aligned} t = 3 : I_0 &= 6 \\ I_1 &= 7 \end{aligned}\right\} \Rightarrow \begin{cases} O_0 = 3 \\ O_1 = 7 \end{cases}$$

Theorem 2.2. *Given a bucket t, each relation $I_i \to O_j$ depends on the bucket value $T_t \in [0\,..\,k!-1]$.*

Proof. Knowing all the relations $I_i \to O_j$ in a bucket, we can construct a combination. In that combination we will see every output value (potentially sorted in $k!$ different possible ways, see Table 2); in other words, there are $k!$ possible configurations to relate input values I_i to output values O_j (note: we need at least $k-1$ relations to set the bucket value T_t). □

Table 2: Combinations of relations

I_0	I_1	...	I_{k-1}	T_t
O_0	O_1	...	O_{k-1}	0
...	...	...	...	...
O_{k-1}	O_{k-2}	...	O_0	$k!-1$

Obviously, once O_j is calculated from I_i, we will make $I_i = O_j$ and apply (4) to continue with the sequence encoding. For this task, in general, we will use an efficient algorithm

of lexicographic rank/unrank [3] in order to encode (calculate bucket value T_t) and decode (calculate combination relating input and output values... in reality, we use indices i and j rather than the input and output values themselves). Note that for $k = 2$ there is no need to use that algorithm because the implementation may be easily optimized (Figure 2).

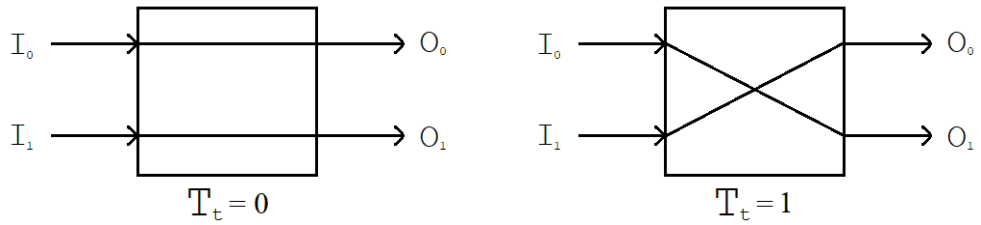

Figure 2: All possible configurations for bucket t using $k = 2$

We have seen that, although it will be necessary to append some additional symbols in the encoding, this is not dramatic because what interests us most is maximizing the mean compression rather than reaching the maximum compression for a specific sequence.
We are going to see two variants or methods to implement the algorithm, without including optimizations; although both have a time and space complexity $O(N)$, in practice the intensive operations using big numbers makes the difference between them.

2.1.1 Normal

A) Encoding
We have the input sequence S using an alphabet of size k, and we want to encode it to the output sequence S' using an alphabet of size b. The procedure is as follows:

1. Make a sweep over input sequence S with two objectives:
   - Get the numeric value of the initial code C_{init} using the first n symbols (they are in base k). This is necessary to know in which bucket the traversal begins.
   - Set the $\frac{N}{k}$ bucket values T_t, where $t \in [0 \,..\, \frac{N}{k} - 1]$, assigning default relations when necessary.
2. Calculate a value v using bucket values T_t and C_{init} in this way:

$$v = C_{init} \cdot (k!)^{\frac{N}{k}} + \sum_{i=0}^{\frac{N}{k}-1} T_i \cdot (k!)^i \tag{6}$$

3. Append to output sequence S' the header data of size H in base b.
4. Append to output sequence S' the value v using base b in $\lceil \log_b (v + 1) \rceil$ symbols ($+1$ because of v is zero-based, this way the logarithm is always defined).

In summary, in this method we don't store every bucket value, but a constructed value v. The number of stored symbols using the output alphabet of size b is:

$$s_{db}(k, n, b) = H + \lceil \log_b (v + 1) \rceil \tag{7}$$

Compare this with (2); note that we could store C_{init} separately, but using it for the calculation of v is slightly better for the compression rate.

B) Decoding
We have as input the encoded sequence S' using an alphabet of size b, which corresponds to a header of size H and the numerical value v, and we want to decode it to the output sequence S using an alphabet of size k. The procedure is as follows:

1. Read and process the H symbols in base b corresponding to the header.
2. Calculate the numeric value v from symbols of S' in base b, excluding the header.
3. Compute the numeric value of the initial code C_{init} in the original sequence S as follows:
$$C_{init} = \left\lfloor \frac{v}{(k!)^{\frac{N}{k}}} \right\rfloor \tag{8}$$
4. Using v and C_{init}, calculate the bucket values T_t applying a decomposition in the numerical system in base $k!$ of the remainder for the division seen in (8):
$$\sum_{i=0}^{\frac{N}{k}-1} T_i \cdot (k!)^i = v - C_{init} \cdot (k!)^{\frac{N}{k}} = v \bmod (k!)^{\frac{N}{k}} \tag{9}$$
5. Once C_{init} and all the bucket values T_t are known, construct the original sequence S in this way:
   - Append to output sequence S the value C_{init} using n symbols in base k.
   - Starting from the initial bucket corresponding to the input value C_{init} using expression (4), we travel through the buckets append to S a symbol j (in base k) corresponding to the relation $I_i \to O_j$. At the end of this sweep, we will have the original sequence S.

This method has the advantage of providing only the excess in symbols to include C_{init} in the encoding, allowing us to almost achieve our reference constant compression seen in (3)... but it may be slow due to the intensive use of operations with big numbers.
In Algorithm 1 and 2 we present very high level pseudocode versions for encoding and decoding, respectively.

Algorithm 1 Normal encoding

Input: S, k, n, N, b, H
Output: S'

1: $S' \leftarrow [\,]$ ▷ Initialize encoded sequence
2: $T[0, ..., \frac{N}{k} - 1] \leftarrow 0$ ▷ Create array for bucket values ($T_t = 0$ by default)
3: $(C_{init}, T) \leftarrow$ `SweepRead(`S`)` ▷ Set initial numeric code and bucket values from S
4: $v \leftarrow$ `CalcValue(`C_{init}`,`T`)` ▷ Compute (6) using C_{init} and bucket values
5: S'`.AppendHeader(`H`)` ▷ Store header in S' using H symbols in base b
6: S'`.AppendValue(`v`)` ▷ Store v in S' using base b

Algorithm 2 Normal decoding

Input: S', k, n, N, b, H
Output: S

1: $S \leftarrow [\,]$ ▷ Initialize decoded sequence
2: $T[0, ..., \frac{N}{k} - 1] \leftarrow 0$ ▷ Create array for bucket values ($T_t = 0$ by default)
3: `LoadHeader(`S'`)` ▷ Read and process the header of S' using base b
4: $v \leftarrow$ `ReadValue(`S'`)` ▷ Read numeric value v stored in S' using base b
5: $C_{init} \leftarrow$ `CalcInitCode(`v`)` ▷ Compute (8) using v
6: $T \leftarrow$ `CalcBucketValues(`S'`,`C_{init}`)` ▷ Compute bucket values using S' and C_{init}
7: $S \leftarrow$ `SweepWrite(`C_{init}`,`T`)` ▷ Reconstruct S from C_{init} and bucket values

2.1.2 Fast

In order to achieve a faster method than the previous one, in expression (2) we will do the following approach:

$$\lceil \log_b(c_{db}) \rceil \leq \frac{N}{k} \lceil \log_b(k!) \rceil \tag{10}$$

It is an approximation because, in general, the use of the right part of the inequation will be bigger than the left part, producing an excess of symbols and a lower compression rate; the exception is the case $k = b = 2$, where the inequation becomes an equation and there is no excess of symbols. The error regarding the exact value will satisfy:

$$\frac{N}{k} \lceil \log_b(k!) \rceil - \lceil \log_b(c_{db}) \rceil < \frac{N}{k} \tag{11}$$

A) Encoding
As in the previous method, we have the input sequence S using an alphabet of size k, and we want to encode it to the output sequence S' using an alphabet of size b. The procedure is as follows:

1. Make a first sweep over input sequence S with two objectives:
   - Get the numeric value of the initial code C_{init} using the first n symbols (they are in base k). This is necessary to know in which bucket the traversal begins.
   - Set the $\frac{N}{k}$ bucket values T_t, where $t \in [0\,..\,\frac{N}{k} - 1]$, assigning default relations when necessary.

   Observe that this first sweep is unnecessary when $k = 2$ because, in this case, a bucket value is automatically set the first time the bucket is visited... but in this text we will implement the algorithm for the general case, without optimizations.
2. Append to output sequence S' the header data of size H in base b.
3. Append to output sequence S' the numeric value C_{init} (in base b) using $\lceil \log_b(N) \rceil$ symbols.
4. Make a second sweep over input sequence S, append to output sequence S' each bucket value T_t (in base b) that is visited for the first time, using $\lceil \log_b(k!) \rceil$ symbols.

In summary, in this method we store every used bucket value. The maximum number of stored symbols using the output alphabet of size b is:

$$s_{db}(k, n, b) = H + \lceil \log_b(N) \rceil + \frac{N}{k} \lceil \log_b(k!) \rceil \tag{12}$$

Compare this with (2) and (7).

B) Decoding
We have as input the encoded sequence S' using an alphabet of size b, which corresponds to a header of size H, the numerical value C_{init} and the bucket values T_t, and we want to decode it to the output sequence S using an alphabet of size k. The procedure is as follows:

1. Read and process the H symbols in base b corresponding to the header.
2. Compute the numeric value of the initial code C_{init} from reading $\lceil \log_b(N) \rceil$ symbols in the input sequence S' (in base b).
3. Once C_{init} is known, construct the original sequence S in this way:
   - Append to output sequence S the value C_{init} using n symbols in base k.
   - Starting from the initial bucket corresponding to the input value C_{init} using expression (4), we travel through the buckets: when a bucket t is visited for the first time, we read $\lceil \log_b(k!) \rceil$ symbols in base b from input sequence S' to compute the bucket value T_t, appending to S a symbol j (in base k) corresponding to the relation $I_i \to O_j$. At the end of this sweep, we will have the original sequence S.

As we saw, for $k = b = 2$ this method is perfect (the only excess of symbols is due to the store of C_{init}), but for $k > 2$ we will have to accept a more or less significant excess in symbols. If we want an efficient algorithm, without an intensive use of operations with big numbers, a lower compression rate is not dramatic. The advantage is a true linear time and space complexity, $O(N)$.

In Algorithm 3 and 4 we present very high level pseudocode versions for encoding and decoding, respectively.

Algorithm 3 Fast encoding

Input: S, k, n, N, b, H
Output: S'

1: $S' \leftarrow [\,]$ ▷ Initialize encoded sequence
2: $T[0, ..., \frac{N}{k} - 1] \leftarrow 0$ ▷ Create array for bucket values ($T_t = 0$ by default)
3: $(C_{init}, T) \leftarrow$ `SweepRead(`S`)` ▷ Set initial numeric code and bucket values from S
4: S'`.AppendHeader(`H`)` ▷ Store header in S' using H symbols in base b
5: S'`.AppendValue(`C_{init}`)` ▷ Store C_{init} in S' using base b
6: `SweepWrite(`S'`,`S`,`C_{init}`,`T`)` ▷ Store in S' the values of visited buckets using base b

Algorithm 4 Fast decoding

Input: S', k, n, N, b, H
Output: S

1: $S \leftarrow [\,]$ ▷ Initialize decoded sequence
2: $T[0, ..., \frac{N}{k} - 1] \leftarrow 0$ ▷ Create array for bucket values ($T_t = 0$ by default)
3: `LoadHeader(`S'`)` ▷ Read and process the header of S' using base b
4: $C_{init} \leftarrow$ `ReadInitialValue(`S'`)` ▷ Read initial numeric code stored in S' using base b
5: S`.AppendValue(`C_{init}`)` ▷ Store C_{init} in S using base k
6: `SweepReadWrite(`S`,`S'`,`C_{init}`)` ▷ Reconstruct S in base k from visited buckets (base b)

2.2 Example

Let's see a little example with the following features, to show the algorithm in action:

- The numeric input values are: $k = b = 2$, $n = 4$, $N = 2^4$.
- The input de Bruijn sequence is $S = \overbrace{1}^{\text{MSB}}01001101111000\overbrace{0}^{\text{LSB}}$.
- We will make the sweep from LSB to MSB.
- To ensure that we will have compression we will take $H = 0$ (i.e. no header).
- Suppose that, in a hypothetical list of de Bruijn sequences of N symbols, sequence S corresponds to index 15 (zero-based).

Well, initially we can say that, according to (1), the total number of de Bruijn sequences is $c_{db}(2, 4) = 256$. In Table 3 we show the result of using the methods discussed in section 1 if we used full indexing.

Table 3: Compression methods for S

Decimal index	Sequence	Constant compression	MSS compression	Optimal encoding
15	1010011011110000	00001111	1111	0000

We would need $s_{db}(2, 4, 2) = 8$ bits to encode our sequence S using constant compression according to (2), 4 bits using MSS compression, and also also 4 bits using optimal compression.

Let's see now what we get applying the two methods of *Loop Compression*. Both methods share two things:

1. The same initial code of n symbols (in base k), which expressed in decimal numeric system is $C_{init} = 0$.
2. The same configuration for the buckets (Figure 3).

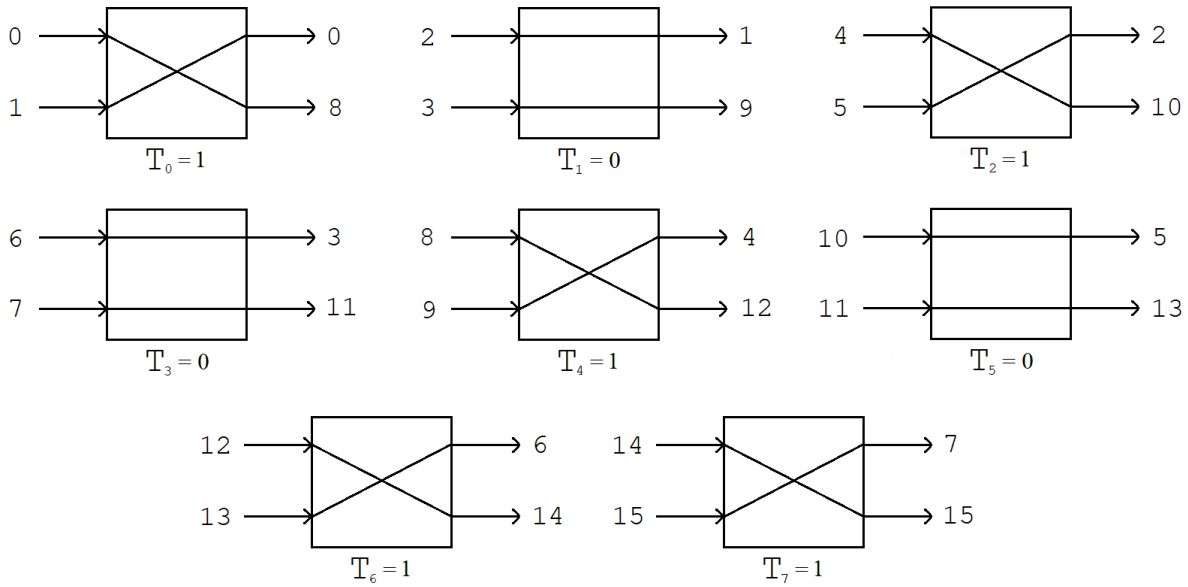

Figure 3: Configuration of buckets for both methods

2.2.1 Normal

A) Encoding

1. Make a sweep over input sequence S to obtain $C_{init} = 0$ and bucket values T_t (Figure 3).
2. Calculate the value v using bucket values T_t and C_{init}, according to (6):
$$v = 0 \cdot 2^8 + (2^0 + 2^2 + 2^4 + 2^6 + 2^7) = 213$$
3. Since $H = 0$, we don't have to append the header to output sequence S'.
4. According to (7), the number of symbols to append the value v to output sequence S' using base b is:
$$s_{db}(k, n, b) = \lceil \log_2(214) \rceil = 8$$

The encoded sequence will be: $S' = \underbrace{1}_{\text{MSB}} 101010 \underbrace{1}_{\text{LSB}}$.

B) Decoding

Here we have as input the encoded sequence $S' = \overbrace{1}^{\text{MSB}} 101010 \overbrace{1}^{\text{LSB}}$.

1. Since $H = 0$, we don't have to read the header.
2. The numeric value v from symbols of S' in base b, excluding the header, is $v = 213$.
3. The numeric value of the initial code C_{init} in the original sequence S is, according to (8):
$$C_{init} = \left\lfloor \frac{213}{2^8} \right\rfloor = 0$$

4. Using v and C_{init}, calculate the bucket values T_t according to (9):

$$\sum_{i=0}^{7} T_i \cdot 2^i = 213 - 0 \cdot 2^8 = 213$$

5. Once C_{init} and all bucket values T_t are known, we construct the original sequence S from the beginning, starting by appending the initial code of n symbols and traveling through the buckets. At the end of this sweep, we will have the original sequence $S = \underbrace{1}_{\text{MSB}}01001101111000\underbrace{0}_{\text{LSB}}$.

2.2.2 Fast

A) Encoding

1. Make a sweep over input sequence S to obtain $C_{init} = 0$ and bucket values T_t (Figure 3).
2. Since $H = 0$, we don't have to append the header to output sequence S'.
3. Append to output sequence S' the numeric value C_{init} (in base b) using $\lceil \log_2(16) \rceil = 4$ symbols.
4. Make a second sweep over input sequence S, appending to output sequence S' each bucket value T_t (in base b) that is visited for the first time, using $\lceil \log_2(2) \rceil = 1$ symbol (see Table 4).

Table 4: Encoding sweep over S using fast method

Input Code	Bucket t	$I_i \rightarrow O_j$	T_t	Output
-	-	-	-	0000 (C_{init})
0000	0	$0 \rightarrow 8$	1	1 (T_0)
1000	4	$8 \rightarrow 12$	1	1 (T_4)
1100	6	$12 \rightarrow 14$	1	1 (T_6)
1110	7	$14 \rightarrow 15$	1	1 (T_7)
1111	7	$15 \rightarrow 7$	1	-
0111	3	$7 \rightarrow 11$	0	0 (T_3)
1011	5	$11 \rightarrow 13$	0	0 (T_5)
1101	6	$13 \rightarrow 6$	1	-
0110	3	$6 \rightarrow 3$	0	-
0011	1	$3 \rightarrow 9$	0	0 (T_1)
1001	4	$9 \rightarrow 4$	1	-
0100	2	$4 \rightarrow 10$	1	1 (T_2)
1010	5	$10 \rightarrow 5$	0	-

The encoded sequence will be: $S' = \overbrace{1}^{\text{MSB}}0001111000\overbrace{0}^{\text{LSB}}$.

As optimization, observe that for $k = 2$ we always have $T_0 = T_{\frac{N}{k}-1} = 1$ to encode a de Bruijn sequence, so in our example we could skip T_0 and T_7 in the encoding.

B) Decoding

Here we have as input the encoded sequence $S' = \overbrace{1}^{\text{MSB}}0001111000\overbrace{0}^{\text{LSB}}$.

1. Since $H = 0$, we don't have to read the header.
2. Compute the numeric value of the initial code $C_{init} = 0$ from reading $\lceil \log_2(16) \rceil = 4$ symbols in the input sequence S' (in base b).
3. Append to output sequence S the value C_{init} using n symbols in base k. Then, starting from the initial bucket $t = 0$ corresponding to the input value $C_{init} = 0$ using expression (4), we travel through the buckets: when a bucket t is visited for the first time, we read $\lceil \log_2(2) \rceil = 1$ symbol in base b from input sequence S' to compute the bucket value T_t, appending to S a symbol j (in base k) corresponding to the relation $I_i \to O_j$ (see Table 5). At the end of this sweep, we will have the original sequence $S = \underbrace{1}_{\text{MSB}}01001101111000\underbrace{0}_{\text{LSB}}$.

Table 5: Decoding sweep over S using fast method

Code	Bucket t	Input T_t	$I_i \to O_j$	Output j
-	-	-	-	0000 (C_{init})
0000	0	1 (T_0)	$0 \to 8$	1
1000	4	1 (T_4)	$8 \to 12$	1
1100	6	1 (T_6)	$12 \to 14$	1
1110	7	1 (T_7)	$14 \to 15$	1
1111	7	-	$15 \to 7$	0
0111	3	0 (T_3)	$7 \to 11$	1
1011	5	0 (T_5)	$11 \to 13$	1
1101	6	-	$13 \to 6$	0
0110	3	-	$6 \to 3$	0
0011	1	0 (T_1)	$3 \to 9$	1
1001	4	-	$9 \to 4$	0
0100	2	1 (T_2)	$4 \to 10$	1

3 Beyond de Bruijn sequences

We have seen that our algorithm works with sequences of length $N = k^n$, which until now were nothing more than cycles of length N in a de Bruijn graph. Let's take in mind that each valid sequence is a Hamiltonian cycle in the corresponding de Bruijn graph $G(k, n)$, and an interesting property is that every Hamiltonian cycle in $G(k, n)$ is associated to an Eulerian cycle in $G(k, n-1)$. For instance, in order to generate de Bruijn sequences of length $N = 2^4$ we can find Eulerian cycles in $G(2, 3)$ (Figure 4). Remember that in an Eulerian cycle we visit each node exactly once, and in a Hamiltonian cycle we visit each edge once.

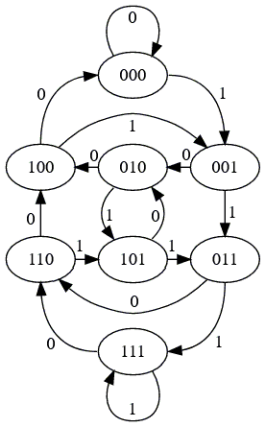

Figure 4: de Bruijn graph $G(2, 3)$

Therefore, the algorithm presented in this text must be able to automatically encode repeated cycles of any length from 1 to N, not just N. When a bucket value is unknown because at least $k-1$ relations have not appear, it will be completed by assigning default relations.

For instance, in the context of the example in section 2.2, if we had the input sequence $S = 0000000000000000$, which is clearly not a de Bruijn sequence, the algorithm would encode it without problems because it is the repetition of a cycle of length 1.

In Table 6 we can see the number of *simple cycles* $c(k, n, s)$ in a de Bruijn graph for some cases, where $s \in [1 \,..\, N]$ is the length of the cycle (for $k = 2$ see sequence A344018 in OEIS [4]). One way to calculate these values is through Johnson's algorithm to find simple cycles in a directed graph [5].

Table 6: Number of simple cycles $c(k, n, s)$ for some cycle lengths

N \ s	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
2^1	2	1	-	-	-	-	-	-	-	-	-	-	-	-	-	-
2^2	2	1	2	1	-	-	-	-	-	-	-	-	-	-	-	-
2^3	2	1	2	3	2	3	4	2	-	-	-	-	-	-	-	-
2^4	2	1	2	3	6	7	8	12	14	17	14	13	12	20	32	16
3^1	3	3	2	-	-	-	-	-	-	-	-	-	-	-	-	-
3^2	3	3	8	12	18	20	24	36	24	-	-	-	-	-	-	-
4^1	4	6	8	6	-	-	-	-	-	-	-	-	-	-	-	-
5^1	5	10	20	30	24	-	-	-	-	-	-	-	-	-	-	-
6^1	6	15	40	90	144	120	-	-	-	-	-	-	-	-	-	-

The total number of cycles $c_s(k, n)$ for a certain length s would be obtained by multiplying the corresponding value $c(k, n, s)$ by s:

$$c_s(k, n) = s \cdot c(k, n, s) \tag{13}$$

The total number of cycles $c_T(k,n)$ taking into account all sequence lengths would be obtained by adding the total number of cycles for each length:

$$c_T(k,n) = \sum_{s=1}^{N} c_s(k,n) = \sum_{s=1}^{N} s \cdot c(k,n,s) \tag{14}$$

Applying (13) and (14) over Table 6, we get the results shown in Table 7.

Table 7: Total number of simple cycles $c_s(k,n)$ and $c_T(k,n)$

N \ s	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	c_T
2^1	2	2	-	-	-	-	-	-	-	-	-	-	-	-	-	-	4
2^2	2	2	6	4	-	-	-	-	-	-	-	-	-	-	-	-	14
2^3	2	2	6	12	10	18	28	16	-	-	-	-	-	-	-	-	94
2^4	2	2	6	12	30	42	56	96	126	170	154	156	156	280	480	256	2024
3^1	3	6	6	-	-	-	-	-	-	-	-	-	-	-	-	-	15
3^2	3	6	24	48	90	120	168	288	216	-	-	-	-	-	-	-	963
4^1	4	12	24	24	-	-	-	-	-	-	-	-	-	-	-	-	64
5^1	5	20	60	120	120	-	-	-	-	-	-	-	-	-	-	-	325
6^1	6	30	120	360	720	720	-	-	-	-	-	-	-	-	-	-	1956

Observe that, as we already knew, the total number of cycles of length $s = N$ is equal to (1), that is, the total number of de Bruijn sequences. Then:

$$c_s(k,n)|_{s=N} = c_{db}(k,n) \tag{15}$$

Since what we are interested in is compressing a sequence without taking into account whether it is cyclic beyond the end of the sequence, Table 7 will be *truncated* as explained below, giving rise to Table 8, where truncated values $c'_s(k,n)$ and $c'_T(k,n)$ are shown. Then, the expression to calculate the total number of sequences of length N compressible by our algorithm will be:

$$c'_T(k,n) = \sum_{s=1}^{N} c'_s(k,n) = c_T(k,n) - x(k,n) \tag{16}$$

The term $x(k,n) \geq 0$ is the number of repeated sequences of length N, due to what was previously mentioned about the truncation. The difficulty lies in finding how to calculate $x(k,n)$ and the values $c'_s(k,n)$ or $c'_T(k,n)$ efficiently without having to count each cycle individually (avoiding exponential time complexity). This problem seems unsolved [6], but we can set an obvious bound:

$$c'_T(k,n) \leq c_T(k,n) \tag{17}$$

Table 8: Total number of truncated simple cycles $c'_s(k,n)$ and $c'_T(k,n)$

N \ s	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	c'_T
2^1	4	0	-	-	-	-	-	-	-	-	-	-	-	-	-	-	4
2^2	2	10	0	0	-	-	-	-	-	-	-	-	-	-	-	-	12
2^3	2	2	6	12	58	0	0	0	-	-	-	-	-	-	-	-	80
2^4	2	2	6	12	30	42	56	96	126	170	154	1116	0	0	0	0	1812
3^1	3	12	0	-	-	-	-	-	-	-	-	-	-	-	-	-	15
3^2	3	6	24	48	90	120	600	0	0	-	-	-	-	-	-	-	891
4^1	4	12	48	0	-	-	-	-	-	-	-	-	-	-	-	-	64
5^1	5	20	60	240	0	-	-	-	-	-	-	-	-	-	-	-	325
6^1	6	30	120	360	1440	0	-	-	-	-	-	-	-	-	-	-	1956

In Table 8 we can observe that $c'_s(k,n) = 0,\ \forall s > N-n$. This means that cycles larger than $N-n$ are truncated, since sequences of length N need at most cycles of length $N-n$ to be compressed by our algorithm.
In Table 9 we show the ratio $r(k,n)$ between the total number of cycles $c_T(k,n)$ and the total number of de Bruijn sequences $c_{db}(k,n)$ of length N, whose expression is:

$$r(k,n) = \frac{c_T(k,n)}{c_{db}(k,n)} \tag{18}$$

Table 9: Ratio $r(k,n)$

k \ n	1	2	3	4	5
2	4/2	14/4	94/16	2024/256	779930/65536
3	15/6	963/216	-	-	-
4	64/24	1629460/331776	-	-	-
5	325/120	-	-	-	-
6	1956/720	-	-	-	-
7	13699/5040	-	-	-	-
8	109600/40320	-	-	-	-
9	986409/362880	-	-	-	-

In Table 10 we show the ratio $r'(k,n)$ between the total number of compressible sequences $c'_T(k,n)$ and the total number of de Bruijn sequences $c_{db}(k,n)$ of length N, whose expression is:

$$r'(k,n) = \frac{c'_T(k,n)}{c_{db}(k,n)} \tag{19}$$

Table 10: Ratio $r'(k,n)$

k \ n	1	2	3	4	5
2	4/2	12/4	80/16	1812/256	723674/65536
3	15/6	891/216	-	-	-
4	64/24	1546516/331776	-	-	-
5	325/120	-	-	-	-
6	1956/720	-	-	-	-
7	13699/5040	-	-	-	-
8	109600/40320	-	-	-	-
9	986409/362880	-	-	-	-

Conjecture 3.1. *The limit of the ratio $r(k,n)$ and $r'(k,n)$ as k approaches infinity is:*

$$\lim_{k\to\infty} r(k,n) = \lim_{k\to\infty} r'(k,n) = e^n \tag{20}$$

Theorem 3.1. *Conjecture 3.1 is true, at least, for $n = 1$.*

Proof. For $n = 1$ we know the following:

1. The numerator of (18) and (19) corresponds to sequence A007526 in OEIS, obtaining:

$$c_T(k,1) = c'_T(k,1) = k! \sum_{i=0}^{k-1} \frac{1}{i!} \tag{21}$$

   Besides, $\frac{c_T(k,1)}{k} = \frac{c'_T(k,1)}{k}$ corresponds to sequence A000522 in OEIS.

2. The denominator of (18) and (19) is, according to (1):

$$c_{db}(k,1) = k! \tag{22}$$

3. Then, ratios (18) and (19) for $n = 1$ will be:

$$\frac{c_T(k,1)}{c_{db}(k,1)} = \frac{c'_T(k,1)}{c_{db}(k,1)} = \sum_{i=0}^{k-1} \frac{1}{i!} \tag{23}$$

   It is a known fact that the above equals e as k approaches to infinity:

$$\lim_{k\to\infty} \frac{c_T(k,1)}{c_{db}(k,1)} = \lim_{k\to\infty} \frac{c'_T(k,1)}{c_{db}(k,1)} = \lim_{k\to\infty} \sum_{i=0}^{k-1} \frac{1}{i!} = \sum_{i=0}^{\infty} \frac{1}{i!} = e \tag{24}$$

□

Note that for our objective of estimating the number of sequences compressible by the algorithm we are only interested in $r'(k,n)$. Assuming Conjecture 3.1 as true, since at the moment we don't know how to calculate $c'_s(k,n)$ or $c'_T(k,n)$, we will make the following estimate for (19) (the larger n, the better it seems):

$$r'(k,n) \approx e^{\left(\frac{k-1}{k}\right)n} \tag{25}$$

This way, we can get a rough idea of the total number of sequences of length N compressible by our algorithm instead of applying (16):

$$c'_T(k,n) \approx e^{\left(\frac{k-1}{k}\right)n} \cdot (k!)^{k^{n-1}} \tag{26}$$

4 Open problems

Prove or refute Conjecture 3.1 or complete Table 10 for $n > 1$ to see if the conjecture is justified.

5 Conclusion

Initially, the compression of de Bruijn sequences seemed a difficult task, but as the set of de Bruijn sequences is relatively small, the only difficulty in order to compress them was to model the internal structure of that set of sequences. This has been achieved by developing the *Loop Compression* shown here in two variants, both with time and space complexity $O(N)$. We have also seen that the algorithm is not applicable only for de Bruijn sequences, but also for all sequences with repeated cycles in a de Bruijn graph.
The direct application, obviously, is to use this compression in those cases in which it is required to store de Bruijn sequences, especially large ones.

References

[1] J. Sawada, A. Williams, and D. Wong. "A surprisingly simple de Bruijn sequence construction". Discrete Mathematics, 339(1):127-131, 2016.

[2] O. Cabrera. "On indexing de Bruijn sequences", 2024.
https://doi.org/10.31224/3650

[3] B. Bonet. "Efficient Algorithms to Rank and Unrank Permutations in Lexicographic Order". Workshop on Search in Artificial Intelligence and Robotics at AAAI, 2008.

[4] OEIS Foundation Inc. (2023), The On-Line Encyclopedia of Integer Sequences, Published electronically at http://oeis.org.

[5] D.B. Johnson. "Finding all the elementary circuits of a directed graph". SIAM Journal on Computing, 4(1):77–84, 1975.

[6] P.R. Bryant, J. Christensen. "The enumeration of shift register sequences". Journal of Combinatorial Theory, Series A, Volume 35, Issue 2, Pages 154-172, 1983.