On indexing de Bruijn sequences

Oscar Cabrera

ocg.steppenwolf@gmail.com

Independent Researcher

April 2024

Abstract

Indexing refers to associating a unique de Bruijn sequence with a numerical value, so that from either index value or sequence we can generate the associated sequence or index, respectively. The objective is to index the greatest number of this type of sequences, for any alphabet size, in the most efficient possible way.
As part of a larger project, a software has been specially designed to analyze these sequences and thus helping us to create and verify a model that meets the objective.

1 Introduction

1.1 Overview

Given an alphabet of size k, a de Bruijn sequence of order n is a cyclic sequence in which every possible string of length n occurs exactly once as a substring. We denote this sequence of length $N = k^n$ by $B(k, n)$. The number of unique de Bruijn sequences $B(k, n)$ is

$$\frac{(k!)^{k^{n-1}}}{k^n}.$$

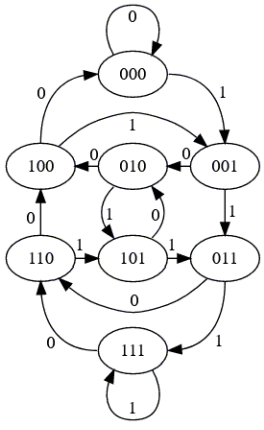


Figure 1: de Bruijn graph $G(2, 3)$

Some examples of binary de Bruijn sequences are: 01, 0011, 00010111, 0000100110101111... There are many known ways to create de Bruijn sequences [1], we can see indexing as another way to generate this kind of sequences. Each valid sequence is a Hamiltonian cycle in the corresponding de Bruijn graph $G(k, n)$. 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 1). Remember that in an Eulerian cycle we visit each node exactly once, and in a Hamiltonian cycle we visit each edge once.

Definition 1.1. *$I_k(n)$ is the number of unique de Bruijn sequences we can index using a method of our choice.*

Proposition 1.1. *Any valid indexing method associates a subset of $I_k(n)$ unique de Bruijn sequences to a set of indices. The indexing relation among index i and sequence S is given by function $f(S) = i$ and its inverse $f^{-1}(i) = S$. The method must satisfy the following requirements:*

1. $\exists x \in [0\,..\,(k!)^{k^{n-1}}/k^n - 1]\ \forall i \in [0\,..\,I_k(n) - 1]\colon f(S_x) = i \iff f^{-1}(i) = S_x$.
   Each index value inside the interval is associated to a unique de Bruijn sequence.
2. *Given* $i, j \in [0\,..\,I_k(n) - 1]$, $f^{-1}(i) = S_x$ *and* $f^{-1}(j) = S_y$, *then* $S_x \neq S_y\ \forall i \neq j$.
   The same unique de Bruijn sequence cannot be associated to more than one index value.

We can apply Hierholzer's algorithm [2] to generate a random de Bruijn sequence, but there is no clear way to index since, although its efficiency is maximum, it does not meet Proposition 1.1.

1.2 Preliminaries

Throughout this text we will make some assumptions. First of all, we normally will use the notation N instead of k^n, especially to express the computational complexity, since it will make things easier when we use other known algorithms; furthermore, it seems more natural, because the time complexity for generating a sequence will never be less than $O(N)$.

Besides, to make the analysis easier we will establish an initial substring of length n, and all generated sequences will begin with that substring.

In next sections we will use the same example to analyze the problem: a handable case for sequences of length $N = 2^4$ and working graph $G(2, 3)$ (Figure 1). In this case, the number of unique de Bruijn sequences is 16 (Figure 2).

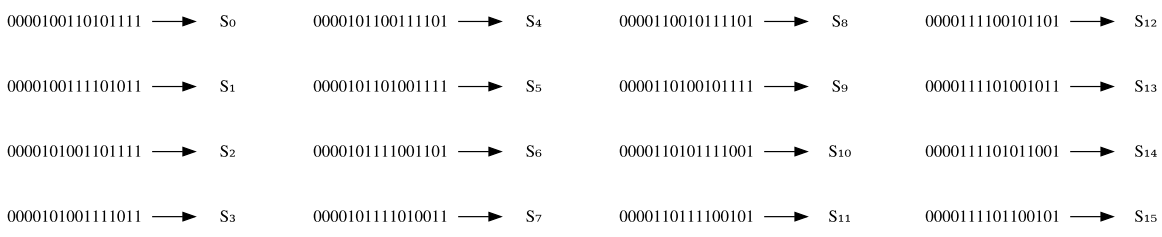


Figure 2: The 16 de Bruijn sequences S_x of length $N = 2^4$ sorted by initial substring 0000

As we walk through a path in graph $G(k, n-1)$ to obtain an Eulerian cycle, we will remove edges of the path. For reasons of clarity and compatibility with other algorithms we use, we will force that it is always possible to form an Eulerian cycle; to do this, we will add an extra node Z to interconnect initial node and latest node. For instance, let's suppose a partial valid path 11 starting from node 000, using the extra node Z applied to our example (Figure 3). Something very important is that, as we remove edges of the path, we won't consider isolated nodes (i.e. nodes without edges) as part of the graph, and they will be excluded from the analysis. To approach the analysis we will use two new concepts that will be useful to us.

Definition 1.2. *A Sequence Tree (ST) is a tree where all unique de Bruijn sequences are gathered along the nodes, removing redundant information.*

Pay attention to the ST for our example (Figure 4).

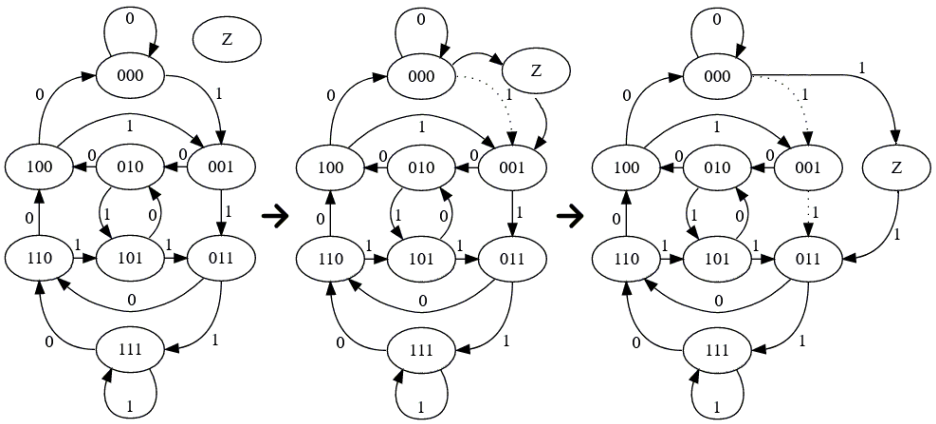


Figure 3: Simulating path 11 in graph $G(2,3)$ (removed edges are dotted), we can find an Eulerian cycle on each step

For simplicity, we will call *splitting nodes* to those branch nodes with more than one child node. Be careful with the following: the fact that a certain node in the ST is not a splitting node doesn't imply that the associated node in graph $G(k, n-1)$ has only one outcoming edge, but rather only one of its outcoming edges leads to, at least, one de Bruijn sequence.

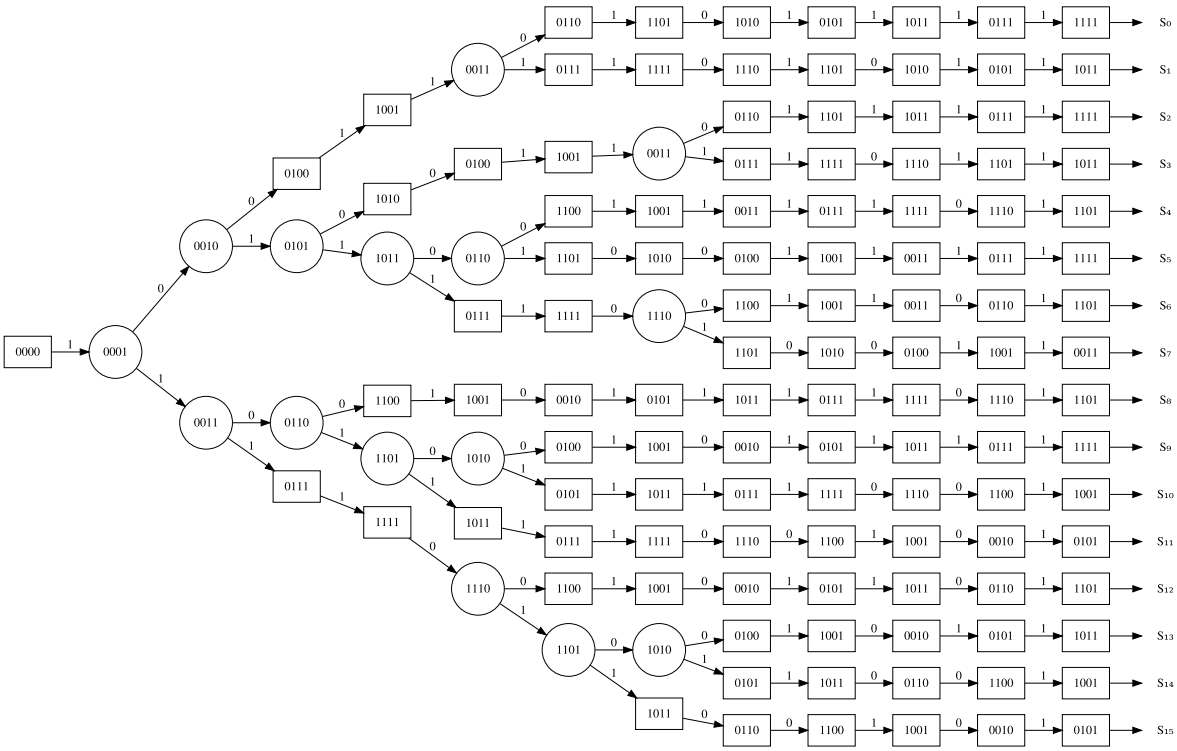


Figure 4: ST for $N = 2^4$ and initial substring 0000 (splitting nodes are shown rounded)

Definition 1.3. *A Sequence Number Tree (SNT) is a simplification of the ST, where we have removed all branch nodes with only one child node. Then, every node in a SNT is either a splitting node or a leaf node. Each splitting node is marked with the total amount of leaf nodes hanging from it, directly or indirectly.*

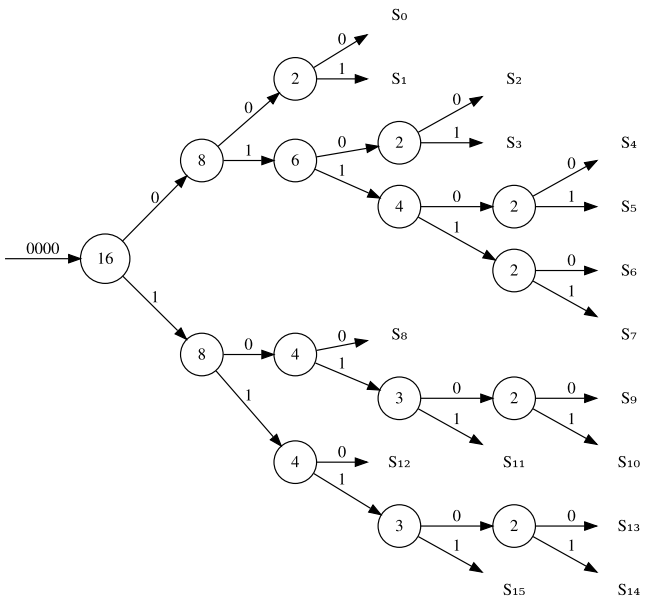


Figure 5: SNT for $N = 2^4$ and initial substring 0000

Look carefully the SNT for our example (Figure 5). Note that the levels of the SNT are reorganized, losing information about the position where a splitting node occurs; this is not worrying and will have its advantages in the analysis. The shape of the SNT may change depending on the chosen initial substring. Keep in mind that, in practice, we won't have to build either the ST nor the SNT, they are only presented here for analytic purposes.

2 Full indexing

Having the objective of indexing the greatest number of de Bruijn sequences in the most efficient possible way, we can wonder if indexing the total amount of sequences is possible. The answer is obviously affirmative, the question is whether it is efficient to do so.
As initial approach, brute force allows us to achieve full indexing, but in a totally inefficient way (exponential cost).

Algorithm 1 Update data indexing in node

Input: S, pos_S `(initially 0)`, $index$ `(initially 0)`
Output: pos_S, $index$ `(both updated)`

```
isSplitNode ← GetNumEC(ec)                    ▷ Get Eulerian cycles for each symbol
if isSplitNode then
    for c = 0 to k − 1 do
        if ec[symOrder[c]] > 0 then
            if symOrder[c] = S[pos_S] then
                pos_S ← pos_S + 1
                break
            index ← index + ec[symOrder[c]]    ▷ Add Eulerian cycles
```

Let's think about an alternative method: if on each splitting node visited in our path through the SNT (or in each node in the ST) we could be able to calculate the values inside its child nodes, then indexing would be trivial (see Algorithm 1 and 2 to update internal data when we find a splitting node).

Algorithm 2 Update data unindexing in node

Input: S `(initially empty)`, $index$
Output: S, $index$ `(both updated)`

```
1: isSplitNode ← GetNumEC(ec)                    ▷ Get Eulerian cycles for each symbol
2: if isSplitNode then
3:     for c = 0 to k − 1 do
4:         if ec[symOrder[c]] > index then
5:             S.Add(symOrder[c])
6:             break
7:         else
8:             index ← index − ec[symOrder[c]]   ▷ Substract Eulerian cycles
```

One way to do this is by using the BEST theorem [3] and the Kirchhoff's matrix tree theorem [4]. The BEST theorem gives the number of Eulerian cycles $ec(G)$ in directed graphs:

$$\mathrm{ec}(G) = t_w(G) \prod_{v \in V} \big(\deg(v) - 1\big)! \tag{1}$$

Here, $t_w(G)$ is the number of spanning trees of graph G, which are trees directed towards the root vertex fixed in w; $deg(v)$ can be either outdegree or indegree of vertex v; V is the set of vertices of our graph. No matter the vertex we choose into the connected graph G, the result will be the same... but, as we said in section 1.2, isolated vertices are excluded.
To calculate $t_w(G)$ we will apply the Kirchhoff's matrix tree theorem: its value is equal to any cofactor of the Laplacian matrix of G, and can be computed in polynomial time. Steps:

1. Construct the Laplacian matrix Q for graph G, excluding isolated vertices.

$$q_{i,j} = \begin{cases} \deg(v_i) - \mathrm{loops}(v_i) & \text{if } i = j \\ -m & \text{if } i \neq j \text{ and } v_i \text{ has } m \text{ edges towards } v_j \end{cases}$$

   For the degree of v_i, $\deg(v_i)$, we can either use indegree or outdegree. The amount $\mathrm{loops}(v_i)$ is the number of edges connecting vertex v_i to itself.

2. Construct a matrix Q' by deleting row w and column w from Q (we can use any other non-isolated vertex of our choice, instead of w).

3. Take the determinant of Q' to obtain $t_w(G)$.

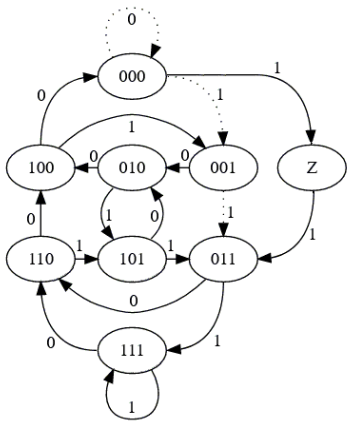


Figure 6: Simulating path 000011 in $G(2,3)$ (removed edges are dotted)

In our example, let's suppose we follow the path 00001 and want to evaluate the value of the child node in the SNT extending the path with the symbol 1, that is, the number of de Bruijn sequences we can constuct with the resulting graph (Figure 6). We know that the correct number of Eulerian cycles is 8 (Figure 2, 4 and 5).

Let's see what we obtain applying the two theorems mentioned above; we use indegree to calculate Q, we take Z as the ninth vertex, and for simplicity we remove the latest row and column of Q to obtain Q'. Finally, we only have to take the determinant of Q', apply (1) and compare the result with the expected value:

$$Q = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 \\ 0 & 1 & -1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 2 & 0 & -1 & -1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 2 & 0 & 0 & -1 & -1 & 0 \\ -1 & -1 & 0 & 0 & 2 & 0 & 0 & 0 & 0 \\ 0 & 0 & -1 & -1 & 0 & 2 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & -1 & -1 & 2 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & -1 & 1 & 0 \\ 0 & 0 & 0 & -1 & 0 & 0 & 0 & 0 & 1 \end{pmatrix}, \quad Q' = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & -1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 2 & 0 & -1 & -1 & 0 & 0 \\ 0 & 0 & 0 & 2 & 0 & 0 & -1 & -1 \\ -1 & -1 & 0 & 0 & 2 & 0 & 0 & 0 \\ 0 & 0 & -1 & -1 & 0 & 2 & 0 & 0 \\ 0 & 0 & 0 & 0 & -1 & -1 & 2 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & -1 & 1 \end{pmatrix}$$

$$t_w(G) = \left|Q'\right| = 8, \quad ec(G) = 8 \cdot (1!)^6 \cdot (0!)^3 = 8$$

Using this method, although the total cost has polynomial complexity, it is still unaffordable except for small cases: take the determinant has a space complexity of $O(N^2)$ and a time complexity typically of $O(N^3)$... by doing it N times, we have a global time complexity of $O(N^4)$. In practice it is even worse due to the need of using operations with support for big numbers. So it seems that, if we don't find a more efficient way to do it, we will have to settle for a partial indexing. As a final note, with full indexing we can obtain a lexicographically sorted indexing when we prioritize the symbols from least to greatest.

3 Partial indexing

3.1 Encoding

If we leave aside the full indexing, let's see if we can find some pattern or scheme in the de Bruijn sequences that allows us to efficiently index as many of them as possible and without the need of using support for big numbers.

Observing the SNT, we can consider the chosen symbols in the splitting nodes to represent the sequence encoding. In our example, we can directly construct all encodings for an initial substring 0000 (Figure 7), and they can be easily verified looking at the corresponding ST (Figure 4).

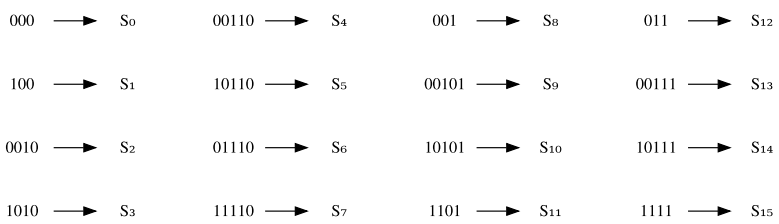


Figure 7: The 16 encoded values for de Bruijn sequences of length $N = 2^4$ with initial substring 0000

Definition 3.1. *Given k, n and the encoding C of a sequence S, we will call M to the length of C.*

Proposition 3.1. *In general, as we change the initial substring, C and M may also change for a certain sequence S, and values C_i $\forall i \in [0\,..\,I_k(n)-1]$ are not fully consecutive.*

In our example, we have four sequences for $M = 3$, four sequences for $M = 4$ and eight sequences for $M = 5$, therefore the minimum is 3 and the maximum is 5. Note that, although we could generate an exact indexing model for our example, it would generally not hold for other cases. Trying to find a pattern in the encoding values can be a really difficult task.

3.2 Indexing

Definition 3.2. *Given k and n, we call $M_{min}(k,n)$ and $M_{max}(k,n)$ to the minimum and maximum value of M, respectively, taking into account all SNT created by all possible initial substrings.*

Conjecture 3.1. *Given k and n, the values $M_{min}(k,n)$, $M_{max}(k,n)$ and the number of sequences for each value of $M \in [M_{min}\,..\,M_{max}]$ are independent of the initial substring.*

We can advance that the previous conjecture seems true in practice, but in our analysis we won't need to assume or prove it, because we are interested in the global minimum and maximum, no matter the initial substring we choose.
In our example, we have seen that $M_{min}(2,4) = 3$ and $M_{max}(2,4) = 5$; we do know it because we have generated the SNT, but in real cases we won't generate that tree... then, how can we know it in advance? If we suppose that $M_{min}(k,n)$ and $M_{max}(k,n)$ follow a pattern or have structure, we have two options: deduce them mathematically in a theorical way, or model them from a practical analysis by software. In this text we will approach the second option.

Proposition 3.2. *Given a tree, whose nodes are either leaf nodes or splitting nodes with exactly k child nodes, we will be able to safely index a number of leafs (i.e. sequences) equal to $k^{M_{min}}$, being M_{min} the minimum level of the tree where there is at least one leaf. Each valid index value belongs to interval $[0\,..\,k^{M_{min}}-1]$, filling the most significant positions with zeros if necessary. This is so because such a tree will always be a full tree up to level M_{min}.*

Theorem 3.1. *Encoding the selected symbols of the splitting nodes in a SNT, the biggest number of indexed de Bruijn sequences that we can guarantee with consecutive indices is:*

$$I_k(n) = 2^{M_{min}(k,n)} \tag{2}$$

Proof. Let's apply Proposition 3.2 to SNTs:

- Case $k = 2$: It fully satisfies the proposition, therefore $I_2(n) = 2^{M_{min}(2,n)}$.
  In our example, we have 8 indexed sequences using partial indexing (Figure 8). We represent indices in binary format, from 0 to 7, starting with MSB (leftmost) and finishing with LSB (rightmost).

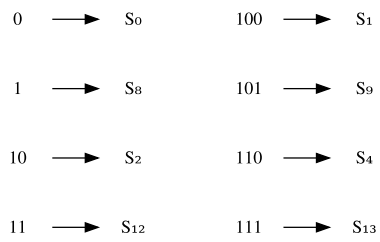


Figure 8: The 8 indexed values for de Bruijn sequences of length $N = 2^4$ with initial substring 0000

  Be careful with the growing direction of the weight of the symbols in the SNT: in our case it grows from the root node (LSB) onwards, filling with zeros the most significant bits to complete the path to reach a de Bruijn sequence.

- Case $k > 2$: It doesn't satisfy the proposition because splitting nodes can have from 2 to k child nodes. In order to be able to do a partial indexing in this case, we will binarize the tree to turn it into case $k = 2$: as to efficiently encode we would need to know the marked values inside splitting nodes (a high complexity task, as we saw in section 2), then we will use precalculated coding tables and we will assume that we always have the worst scenario (one bit encoding). To clarify the method let's leave aside our classical example for a moment and look at a specific example only for this time: there are 24 de Bruijn sequences of length $N = 3^2$ with initial substring 00 in lexicographic order (Figure 9).

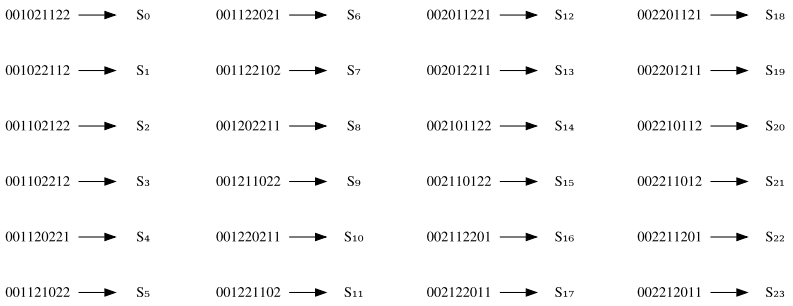
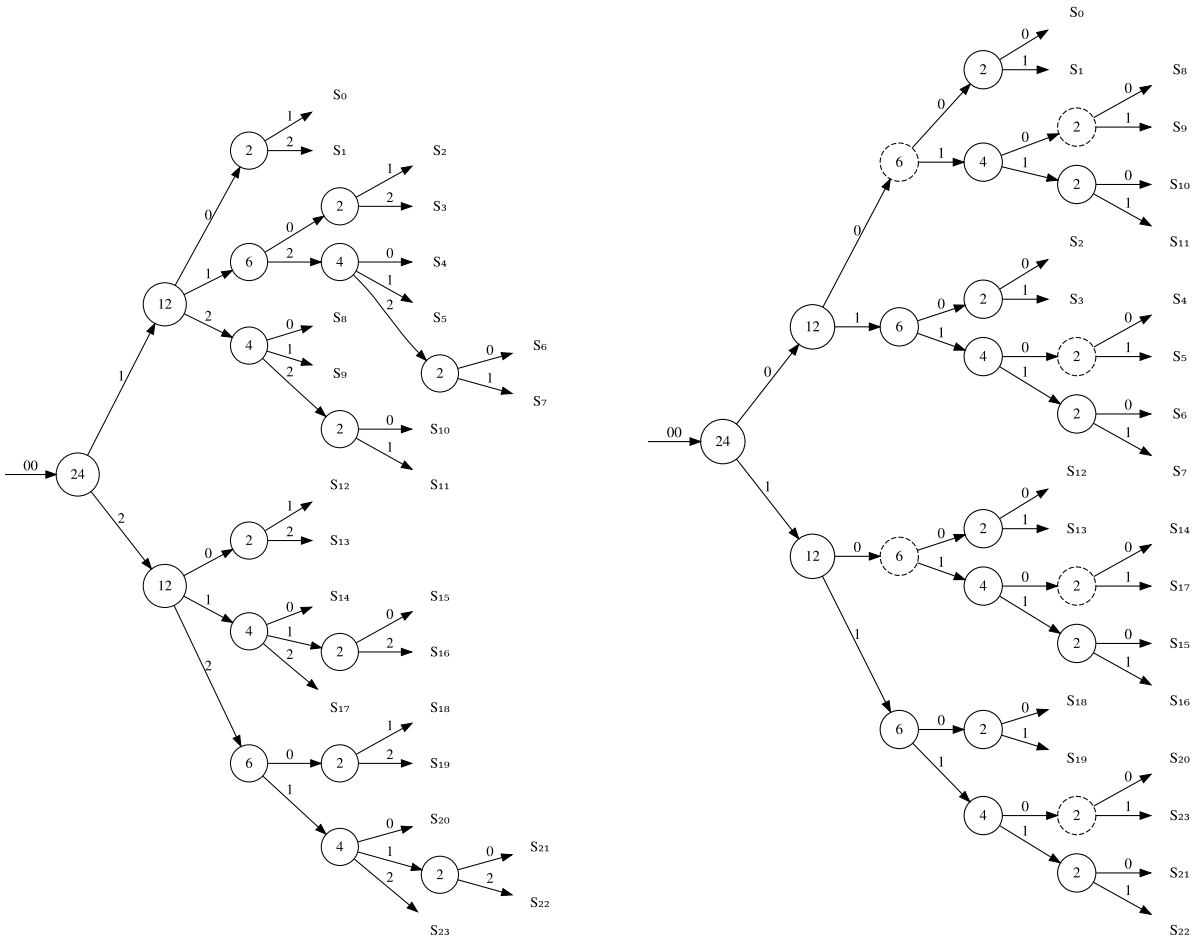


Figure 9: The 24 de Bruijn sequences of length $N = 3^2$ sorted by initial substring 00

  Again, we can build its associated SNT (Figure 10a), where we observe that $M_{min}(3, 2) = 3$, and its binarized SNT (Figure 10b). Note that in this binarized SNT we observe that $M_{min}(3, 2) = 4$, and dashed nodes mean unreal nodes, only shown to see how node



(a) SNT **(b)** Binarized SNT

Figure 10: Trees for $N = 3^2$ and initial substring 00

grouping is made during encoding. This binarized SNT is just one among all encoding possibilities... in fact, it is the best one because it always merges the two nodes with lowest value, providing a number of indexed sequences of 2^4. But, as we generally won't know the values marked in the nodes, we will suppose the worst scenario: a number of indexed sequences of 2^3 using partial indexing (Figure 11).

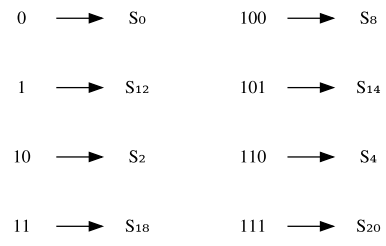


Figure 11: The 8 indexed values for de Bruijn sequences of length $N = 3^2$ with initial substring 00

Applying both cases, we come to the conclusion that (2) gives us a predictable and safe bound that assures us that we can index satisfying Proposition 1.1.

□

Now, we only need a way to deduce or model the funtion $M_{min}(k, n)$: as we'll see, we'll try to model it in section 4. Note that to carry out partial indexing it is crucial to verify if a node is a splitting node; we have two options to do it:

- Hierholzer's algorithm: if we can build an Eulerian cycle by at least two child nodes of the current node, then the current node is a splitting node. Time complexity: $O(N)$. Space complexity: $O(N)$. We get a global time complexity of $O(N^2)$. Not bad, but we can think of a better a way to improve it: having to build an Eulerian cycle is perhaps a waste (too much information that we don't need) and what is really interesting is to detect if by removing an edge is still possible to build an Eulerian cycle... of course, this detection must have a time complexity lower than $O(N)$ to get a global time complexity lower than $O(N^2)$.
- Fully Dynamic Connectivity (FDC): if after removing an edge the source node doesn't become isolated, for sure we will have to return to that node at some point of our path. It implies that the two nodes, whose shared edge we remove, must continue belonging to the same connected component. In fact, discarding isolated vertices, the graph will always have a single connected component; then, we can deduce that if we remove an edge that doesn't lead to a de Bruijn sequence, the graph will be divided into two disconnected subgraphs.

  Theorem 3.2. *Given a partial path on a de Bruijn graph, and using vertex Z for readapting the graph after the edge removal, when we fail trying to build an Eulerian cycle it always means that the main graph is not connected anymore.*

  Proof. A graph must satisfy two requirements so that we can apply Hierholzer's algorithm:

  - It must be an Eulerian graph. This requirement is satisfied since the beginning; in every vertex of the graph, indegree must be equal to outdegree in order to obtain an Eulerian cycle. It is always true, no matter the edge we remove, due to the use of vertex Z readapting the graph after each edge removal (review Figure 3).
  - It must be a connected graph. This requirement is satisfied at the beginning, because a de Bruijn graph is always initially connected, but it can't be guaranteed until the end: if we remove an edge that doesn't lead to an Eulerian cycle, the graph must become disconnected because the first requirement is always true.

  Hence, by checking connectivity between the two nodes after removing the shared edge, we can say whether the extended path is valid with complete certainty. □

  So after removing an edge we have to check if the two nodes are still connected (as long as the source node is not isolated), and do this with a time complexity lower than $O(N)$. For directed graphs, like our de Bruijn graphs, such efficient algorithm is not known...

but for undirected graphs there is an existing one [5]. Luckily, in our case, as we want to detect a disconnection between two subgraphs, we don't care if we take our graph as if it were undirected, we are only interested in that break of the graph. Then, we can use that efficient algorithm without problem. Using dynamic connectivity, we dynamically maintain information about the connected components of a graph. When edges can be either added or removed, as in our case, this can be called *fully dynamic connectivity*. If advancing by at least two child nodes of the current node the graph remains connected, then the current node is a splitting node. Time complexity: $O(\lg^2 N)$ amortized time per update (edge insertion or removal), and $O(\lg N)$ per connectivity query (it can be improved to $O(\lg N / \lg \lg N)$). Space complexity: $O(N \lg^3 N)$. We get a global time complexity of $O(N \lg^2 N)$, which is quite acceptable.

Going back to our classic example, let's suppose we have a valid path 0000111 and want to extend it with a 0 using FDC method (Figure 12). As we see, the main connected graph has been divided into two disconnected subgraphs containing non-isolated vertices, and one of them composed only of vertex 111. Since there is no way to return directly or indirectly to vertex 111 from vertex 110, we can't construct a de Bruijn sequence and therefore the extension of the path is invalid.

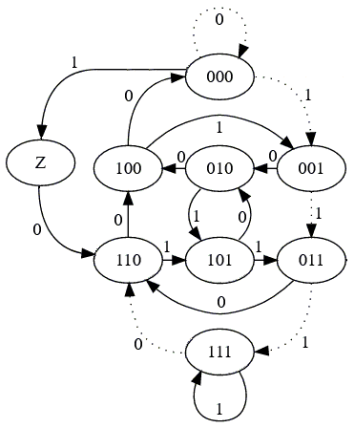


Figure 12: Simulation of path 00001110 in graph $G(2, 3)$ creates two disconnected subgraphs

Finally, very schematic indexing and unindexing algorithms are proposed (Algorithm 3 and 4). Thus, the critical work of partial indexing consists of determining which node edges lead us to de Bruijn sequences (Algorithm 5).

Algorithm 3 Partial indexing

Input: k, n, $initSubStr$, $index$, $method$
Output: S, $state$

```
1:  N ← k^n
2:  G ← CreateGraph(k,n − 1)                          ▷ Node Z used
3:  G.ApplyInitialSubString(initSubStr)
4:  pos_index ← 0, S ← initSubStr, state ← True
5:  for c = 1 to N − n do
6:      edges ← G.GetValidNodeEdges(method)            ▷ Algorithm 5
7:      (selEdge, pos_index) ← G.GetSelEdgeByIndex(edges,index,pos_index)
8:      if selEdge = None then
9:          S ← [ ], state ← False
10:         break
11:     else
12:         S.Add(edges[selEdge])
13:         G.SelectEdge(edges[selEdge])
```

Algorithm 4 Partial unindexing

Input: k, n, S, *method*
Output: *index*, *state*

$N \leftarrow k^n$
$initSubStr \leftarrow S[0, ..., n-1]$
$G \leftarrow$ `CreateGraph(`k`,`$n-1$`)` ▷ Node Z used
G`.ApplyInitialSubString(`$initSubStr$`)`
$pos_S \leftarrow n$, $index \leftarrow [\,]$, $state \leftarrow$ `True`
for $c = 1$ to $N - n$ **do**
    $edges \leftarrow G$`.GetValidNodeEdges(`*method*`)` ▷ Algorithm 5
    $(selEdge, pos_S, binCodif) \leftarrow G$`.GetSelEdgeByS(`$edges$`,`$S$`,`$pos_S$`)`
    if $selEdge =$ `None` **then**
        $index \leftarrow [\,]$, $state \leftarrow$ `False`
        break
    else
        $index$`.Add(`$binCodif$`)`
        G`.SelectEdge(`$Edges[selEdge]$`)`

Algorithm 5 Get valid edges of current node in SNT

Input: G, *method*
Output: *edges*

$edges \leftarrow [\,]$
$V_i \leftarrow G$`.GetCurrentNode()`
$nodeEdges \leftarrow G$`.GetNodeEdges()` ▷ symOrder[ ] used
for each $e \in nodeEdges$ **do**
    $V_f \leftarrow G$`.GetNodeUsingEdge(`e`)`
    if *method* = `FDC2` **then**
        $NC_i \leftarrow G$`.GetNumConnectedComponents()`
    G`.SelectEdge(`e`)`
    $state \leftarrow G$`.IsIsolated(`V_i`)`
    if $\neg$ *state* **then**
        if *method* = `Hierholzer` **then**
            $state \leftarrow G$`.Hierholzer()`
        else if *method* = `FDC1` **then**
            $state \leftarrow G$`.IsConnected(`V_i`,`V_f`)`
        else ▷ FDC2
            $NC_f \leftarrow G$`.GetNumConnectedComponents()`
            $state \leftarrow G$`.IsConnected(`V_i`,`V_f`)`
            if $NC_i = NC_f$ **then**
                $state \leftarrow$ `True`
    if *state* **then**
        $edges$`.Add(`e`)`
    G`.UnselectEdge(`e`)`

As a final note, with partial indexing we don't obtain a lexicographically sorted indexing when we prioritize the symbols from least to greatest.

4 Implementation: Myshella

Myshella is a personal project with a broad and idealistic objective: to bring order into chaos. It will be a highly appreciated help for the purpose of this text, indexing, because of the difficulty of theoretical analysis and fast increase in the number of sequences for different values of k and n. As part of this project, a C++ implementation has been developed for the analysis of different aspects of de Bruijn sequences, with the additional objective of finding a potential use in cryptography.

Some implementation features of *Myshella* related to this text:

1. Symbol sorting in the splitting nodes.
   It allows three types of sorting: increasing, decreasing and random.
2. Full indexing/unindexing of the sequences.
   It uses LU decomposition to take the determinant [6].
3. Encoding/decoding of the sequences.
4. Partial indexing/unindexing of the sequences.
   - Hierholzer's algorithm or FDC may be used. For the second case, we take advantage of an excellent open source implementation [7].
   - In order to detect if the graph becomes disconnected after an edge removal, there are two options: we can either call a function to see if source vertex and target vertex are still connected, with a time complexity of $O(\lg N)$, or call another function to compare the number of connected components in the graph after and before the edge removal, with a time complexity of $O(1)$; of course the second option is slightly better, but in practice we have a time complexity of $O(\lg^2 N)$ due to the edge insertion and removal, no matter the option we use.
   - When tree binarization is needed ($k > 2$), we use precalculated tables for Huffman coding [8], assuming symbol equiprobability.
5. Smart search of practical values for $M_{min}(k, n)$ and $M_{max}(k, n)$.
   Given a certain splitting node, we use the method seen in section 2 (BEST theorem and Kirchhoff's matrix tree theorem) to find the child nodes with the minimum or maximum number of Eulerian cycles. In order to speed up the process, we avoid the use of big numbers and we calculate the logarithm of the determinant. Even so, both factors are difficult to evaluate for high values of n.

In the previous study of partial indexing we have seen that the only thing we needed to find out was $M_{min}(k, n)$. Let's start the analysis using *Myshella* for $k = 2$, grouping practical results for functions $M_{min}(2, n)$ and $M_{max}(2, n)$ (Table 1).

Table 1: Reached values of $M_{min}(2,n)$ and $M_{max}(2,n)$ using *Myshella*

n	1	2	3	4	5	6	7	8	9
$M_{min}(2,n)$	0	0	1	3	9	19	45	93	197
$M_{max}(2,n)$	0	0	1	5	13	29	61	125	253

The numerical series for $M_{min}(2,n)$ and $M_{max}(2,n)$ match the sequence A329145(n) and max(A036563($n-1$), 0), respectively, in OEIS [9], having the following expressions:

$$M_{min}(2,n) = 2^{n-1} - N_2(n) + 1 \tag{3}$$

$$M_{max}(2,n) = \max(2^{n-1} - 3, 0) \tag{4}$$

$N_k(n)$ is the number of necklaces of length n over an alphabet of size k (where $\varphi(n)$ is Euler's totient function) [10]:

$$N_k(n) = \frac{1}{n}\sum_{d|n}\varphi(d)k^{n/d} = \frac{1}{n}\sum_{i=1}^{n}k^{\gcd(i,n)} \tag{5}$$

We can easily calculate some reference values of $N_k(n)$ for $k \in [2\,..\,6]$ (Table 2, where the appropriate numerical series in OEIS are included). We must take in mind that $N_1(n) = 1\ \forall n$.

Table 2: Some values of $N_k(n)$

k \ n	1	2	3	4	5	6	7	8	9	10	OEIS
2	2	3	4	6	8	14	20	36	60	108	A000031(n)
3	3	6	11	24	51	130	315	834	2195	5934	A001867(n)
4	4	10	24	70	208	700	2344	8230	29144	104968	A001868(n)
5	5	15	45	165	629	2635	11165	48915	217045	976887	A001869(n)
6	6	21	76	336	1560	7826	39996	210126	1119796	6047412	A054625(n)

Going further, we can group practical results for functions $M_{min}(k,n)$ and $M_{max}(k,n)$, respectively, using *Myshella* for some values of k and n (Table 3 and 4).

Table 3: Some values of $M_{min}(k,n)$

k \ n	1	2	3	4	5	6	7	8	9	10
2	0	0	1	3	9	19	45	93	197	*405*
3	1	3	11	36	*119*	*370*	*1163*	*3576*	*10987*	*33540*
4	2	8	35	146	*611*	*2502*	*10259*	*41756*	*169659*	*687398*
5	3	15	79	*405*	*2079*	*10565*	*53679*	*271815*	*1374599*	*6940581*
6	4	24	149	*909*	*5549*	*33689*	*204449*	*1238469*	*7495329*	*45317955*

Table 4: Some values of $M_{max}(k,n)$

k \ n	1	2	3	4	5	6	7	8	9	10
2	0	0	1	5	13	29	61	125	253	509
3	1	5	17	53	161	485	1457	4373	*13121*	*39365*
4	2	11	47	191	767	3071	*12287*	*49151*	*196607*	*786431*
5	3	19	99	499	2499	*12499*	*62499*	*312499*	*1562499*	*7812499*
6	4	29	179	1079	*6479*	*38879*	*233279*	*1399679*	*8398079*	*50388479*

A value in normal style means that *Myshella* has successfully reached the predicted value, and not exceeded (neither below $M_{min}(k,n)$ nor above $M_{max}(k,n)$); a value in italics means that it has not been reached in practice but, what is more important, it has not been exceeded either (an excess would invalidate the model automatically).

Let's explain where the practical and predicted values come from. The smart search of practical values for $M_{min}(k,n)$ and $M_{max}(k,n)$ is based on the following conjecture:

Conjecture 4.1. *There is at least one sequence with encoding length equal to $M_{min}(k,n)$ or $M_{max}(k,n)$ choosing always the child nodes with the minimum or maximum number of Eulerian cycles, respectively, through a path in the SNT.*

This conjecture, although it may be not true in general, allows us to reach in practice the predicted values in many cases, as it is shown in Table 3 and 4. Taking into account these practical values, we can try to generalize (3) and (4) to create the predictive model for partial indexing:

Hypothesis 4.1. *Given k and n, the exact expressions for values $M_{min}(k,n)$ and $M_{max}(k,n)$ are the following:*

$$M_{min}(k,n) = (k-1)k^{n-1} - N_k(n) + N_{k-1}(n) \tag{6}$$

$$M_{max}(k,n) = \begin{cases} \max(2^{n-1}-3, 0) & \text{if } k = 2 \\ (k-1)k^{n-1} - 1 & \text{if } k > 2 \end{cases} \tag{7}$$

This model seems to fit perfectly the practical values. Note that, in practice, grouping terms we calculate (6) in this way:

$$M_{min}(k,n) = (k-1)k^{n-1} - \frac{1}{n}\sum_{d|n} \varphi(d)[k^{n/d} - (k-1)^{n/d}] \tag{8}$$

In summary, (2) gives us the number of indexed sequences $I_k(n)$ using partial indexing, whose function $M_{min}(k,n)$ we have successfully modeled in (6) or (8), so we can generate with a global time complexity of $O(N \lg^2 N)$ any de Bruijn sequence associated to a certain index $i \in [0\,..\,I_k(n)-1]$ (and vice versa).
We must keep in mind that, although in general $I_k(n)$ is huge in absolute terms, it is an insignificant part of the total amount of unique de Bruijn sequences.

5 Conclusion

We have initially set the requirements that any valid indexing method must satisfy. As a consequence of the high complexity of the method used in this text, full indexing in practice is only possible for small cases. On the contrary, using partial indexing we have supposedly achieved the objective of indexing a relatively big number of unique de Bruijn sequences, for any alphabet size and in a quite efficient way. For that purpose we have modeled the structure of two properties of these sequences, $M_{min}(k, n)$ and $M_{max}(k, n)$ (lengths of encoded sequences using splitting nodes), thanks to an implementation of project *Myshella*.

As long as the model is correct, partial indexing offers another way to generate de Bruijn sequences, added to those already known, with acceptable efficiency. But before going further in the investigation of potential applications, a proof for the model is required.

What we have seen in this text could also be used in other cases where we want to index the Eulerian cycles for a certain family of graphs; although, in order to use partial indexing, we would have to know how to predict the value M_{min} for those kind of graphs.

6 Open problems

There are many related things pending investigation:

1. Proof for Hypothesis 4.1 to confirm the predictive model used in partial indexing. Otherwise, we should keep trying to reach in practice more predicted values of Table 3 and 4; that way, either the model would be more robust or we could find counterexamples to refute it.
2. Proofs for Conjecture 3.1 and 4.1. There is no need, but it would be interesting to know if they are true. Especially, the creation of a model for the number of de Bruijn sequences for each $M \in [M_{min} \,..\, M_{max}]$, given k and n.
3. Possible pattern in de Bruijn sequences to achieve an efficient full indexing. Alternatively, a way to make a kind of dynamic full indexing using the method seen in this text, saving operations.
4. For $k > 2$, an efficient coding in splitting nodes for increasing $M_{min}(k, n)$ in the binarized SNT, because right now we suppose the worst coding case.
5. Possible use of these indexing methods in cryptography.

References

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

[2] C. Hierholzer. "Ueber die Möglichkeit, einen Linienzug ohne Wiederholung und ohne Unterbrechung zu umfahren". Mathematische Annalen, 6:30–32, 1873.

[3] T. van Aardenne-Ehrenfest, N. G. de Bruijn. "Circuits and tress in oriented linear graphs". Simon Stevin, 28: 203–217, 1951.

[4] G. Kirchhoff. "Ueber die Auflösung der Gleichungen, auf welche man bei der Untersuchung der linearen Vertheilung Galvanischer Ströme geführt wird". Annalen der Physik und Chemie, vol. 72, pp. 497-508, 1847.

[5] J. Holm, K. de Lichtenberg, and M. Thorup. "Poly-logarithmic deterministic fully-dynamic algorithms for connectivity, minimum spanning tree, 2-edge, and biconnectivity". Journal of the ACM, 48(4):723–760, 2001.

[6] A. Schwarzenberg-Czerny. "On matrix factorization and efficient least squares solution". Astronomy and Astrophysics Supplement Series, 110: 405. 1995.

[7] T. Tseng. "Dynamic Connectivity". https://github.com/tomtseng/dynamic-connectivity-hdt

[8] D. Huffman. "A Method for the Construction of Minimum-Redundancy Codes". Proceedings of the IRE, 40 (9): 1098–1101. 1952.

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

[10] G. Pólya. "Kombinatorische Anzahlbestimmungen für Gruppen, Graphen und chemische Verbindungen". Acta Mathematica, 68 (1): 145–254. 1937.