Longest common substring regarding the set of de Bruijn sequences

Oscar Cabrera
ocg.steppenwolf@gmail.com

Independent Researcher

August 2024

Abstract

An algorithm is presented in order to, given any alphabet of size k, compute the longest common substring (LCS) between a sequence S of length N and the whole set of de Bruijn sequences for that length, understood as the maximum value between all LCS of S respecting every sequence of the set.

1 Introduction

Given any sequence S over an alphabet of size k and length $N = k^n$, may be interesting to compute the longest common substring between that sequence and the set of de Bruijn sequences for that length. First of all, let's clarify some concepts.
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. The number of total (i.e. not unique) de Bruijn sequences of length N is $(k!)^{k^{n-1}}$.

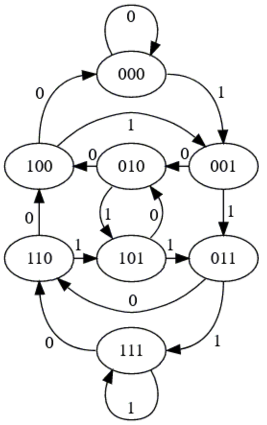


Figure 1: de Bruijn graph $G(2,3)$

Some examples of binary de Bruijn sequences are: 01, 0011, 00010111, 0000100110101111... There are many known ways to create de Bruijn sequences [1]. 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.

The Longest Common Substring (LCS) of two or more strings is one of the longest strings that is a substring of all of them. The LCS may not be unique, and there are algorithms to compute it efficiently [2], also for the cyclic LCS.

According to this, we want to compute the following:

Definition 1.1. *Given k, n and function $L(s_0, s_1)$ that gives the LCS between sequences s_0 and s_1, we define the LCS between any sequence S of length $N = k^n$ and the whole set D of* *de Bruijn sequences for that length by:*

$$L_D(S) = \{L(S, s_j),\ s_j \in D : \texttt{length}(L(S, s_j)) = \max_{s_i \in D}\{\texttt{length}(L(S, s_i))\}\} \qquad (1)$$

The literal implementation of (1) involves an exponential time complexity due to the fast growth for the number of de Bruijn sequences. It is not acceptable, so we will have to think of a smarter way.

Our desired algorithm must to avoid the one by one sequence comparison taking advantage of the properties of de Bruijn graphs; this way, we will make the comparison with all those sequences somehow 'at once', roughly said. The key is to express Definition 1.1 in other words: L_D is the set of longest substrings of S which can be completed to obtain at least one de Bruijn sequence of length N.

2 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. The method uses a kind of *dynamic construction*, and it works as follows:

1. Starting from an initial node, we go building an Eulerian path in the de Bruijn graph removing the edges labeled by the symbols of S, extending the current substring.
2. When we find a problem, the current substring finishes. Then we go restoring edges from the beginning of the substring until the problem is fixed.
3. Update the initial node. If it is outside the sequence, we exit. Otherwise, go to step 1 to continue extending the new substring.

The 'problem' we can find in step 2 can only be one of these:

a) The edge to remove doesn't exist because it has previously removed.

b) After removing the edge, it is not possible to build an Eulerian path, and the only reason possible for this is that the graph is not connected anymore; for detecting the graph disconnection we will use a technique already used in a previous article for project *Myshella* [3]. Let's make a brief explanation.
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 2.1. *Given a partial path on a de Bruijn graph, when we fail trying to build an Eulerian path it always means that the main graph is not connected anymore.*

Proof. Every de Bruijn graph must satisfy two requirements so that we can build an Eulerian cycle (i.e. a de Bruijn sequence):

- It must be an Eulerian graph at the beginning. This requirement is satisfied: in every vertex of the graph, indegree must be equal to outdegree in order to obtain an Eulerian cycle. We lose this property once we start to remove edges, but in the current graph we will always be able to build an Eulerian path between the initial node ($indegree - outdegree = 1$) and the current node ($outdegree - indegree = 1$), being every other node with equal indegree and outdegree. This way, an Eulerian cycle is always indirectly reachable.
- 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 path, 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 good time complexity. For directed graphs, like our de Bruijn graphs, such efficient algorithm is not known... but for undirected graphs there is an existing one [4]. 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.* 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 $O(\lg N / \lg \lg N)$).

A basic implementation for the explained method is shown in Algorithm 1. In function `ManageSubString` we can handle the new substring found (add to a list, manage the maximum length...). For the implementation of FDC, we take advantage of an excellent open source implementation [5].

Theorem 2.2. *Dynamic construction exemplified in Algorithm 1 always finds the exact L_D.*

Proof. As a consequence of Theorem 2.1, given an initial node v_i, FDC guarantees that we will obtain the maximum path possible according to sequence S and the de Bruijn graph, so it is a sure fact that FDC correctly delimits all the substrings. Then, soon or later the algorithm will detect the first node of each substring $s \in L_D$ as initial node for a substring. □

Note that we can easily know the length of the 'longest' substring on each position of sequence S from the substrings found using the algorithm: for every substring, we start indicating the maximum length on its initial position, and then we go decreasing it down to n (the minimum value possible) as we advance; finally, on each position we will take the maximum value. The exception is the case where S is already a de Bruijn sequence: then, the length for the longest substring on each position is N.

Algorithm 1 Computation of L_D

```
Input: $k$, $n$, $S$
Output: $L_D$
$N \leftarrow k^n$
$v_i \leftarrow$ CalcFirstNode($S[0,...,n-2]$)    ▷ Value for the first $n-1$ symbols of $S$
$G \leftarrow$ CreateGraph($k$,$n-1$)
$G$.SetInitialNode($v_i$)
$v_1 \leftarrow v_i,\ p \leftarrow p_i + n - 1$
while $p_i < N \wedge (p - p_i) < N$ do
    $sym \leftarrow S[p \mod N],\ v_2 \leftarrow$ CalcNextNode($v_1, sym$)
    $nc \leftarrow G$.GetNumberOfConnectedComponents()
    $rem \leftarrow G$.DeleteEdge($v_1, v_2, sym$)    ▷ Remove edge $v_1 \rightarrow v_2$ labeled by $sym$
    if $\neg\ rem$ then
        $conn \leftarrow$ True
    else
        $conn \leftarrow G$.IsIsolated($v_1$) $\vee$ ($nc = G$.GetNumberOfConnectedComponents())
    if $\neg\ rem \vee \neg\ conn$ then    ▷ Edge doesn't exist or graph is disconnected
        $allowAdd \leftarrow$ True
        ManageSubString($p_i, p - p_i$)    ▷ New substring of length $p - p_i$ at position $p_i$
        $vStart_1 \leftarrow v_1,\ vStart_2 \leftarrow v_2,\ sStart \leftarrow sym,\ pStart \leftarrow p$    ▷ Store current values
        $v_1 \leftarrow v_i,\ p \leftarrow p_i + n - 1$    ▷ Set starting values for the loop
        while $allowAdd$ do    ▷ Restore edges until problem is fixed
            $sym \leftarrow S[p \mod N],\ v_2 \leftarrow$ CalcNextNode($v_1, sym$)
            if $\neg\ rem$ then
                $allowAdd \leftarrow vStart_1 \neq v_1 \vee vStart_2 \neq v2 \vee sStart \neq sym$
                if $\neg\ allowAdd$ then    ▷ If we are done, check if graph is disconnected
                    $rem \leftarrow$ True
                    $conn \leftarrow G$.IsIsolated($vStart_1$) $\vee$ $G$.IsConnected($vStart_1, vStart_2$)
            if $allowAdd$ then
                $G$.AddEdge($v_1, v_2, sym$)    ▷ Restore edge $v_1 \rightarrow v_2$ labeled by $sym$
            if $\neg\ conn$ then
                $allowAdd \leftarrow \neg\ G$.IsConnected($vStart_1, vStart_2$)
            $p \leftarrow p + 1,\ v_1 \leftarrow v_2$
        $v_i \leftarrow v_1,\ p_i \leftarrow p - n + 1,\ v_1 \leftarrow vStart_2,\ p \leftarrow pStart + 1$ ▷ Initiate next substring
    else
        $p \leftarrow p + 1,\ v_1 \leftarrow v_2$    ▷ All ok, update position and node
if $p_i < N$ then
    ManageSubString($p_i, p - p_i$)    ▷ New substring of length $p - p_i$ at position $p_i$
$L_D \leftarrow$ GetMaxSubStrings()  ▷ Get all (or one of) the LCS stored by ManageSubString
```

Finally, keep in mind that, if we use a rotated version of S, the number of substrings found may change, but not the substrings in L_D. May be interesting to know the minimum number of substrings for S, independently of rotation.

Proposition 2.1. *Being $l[0, ..., N-1]$ a vector indicating the length for the substring on each position of S, and r the number of substrings found using dynamic construction, in order to know the number of substrings of S, independently of rotation, we have to make the following correction in the algorithm:*

$$l[N-1] = l[0] + 1 \Longrightarrow r \leftarrow r - 1$$

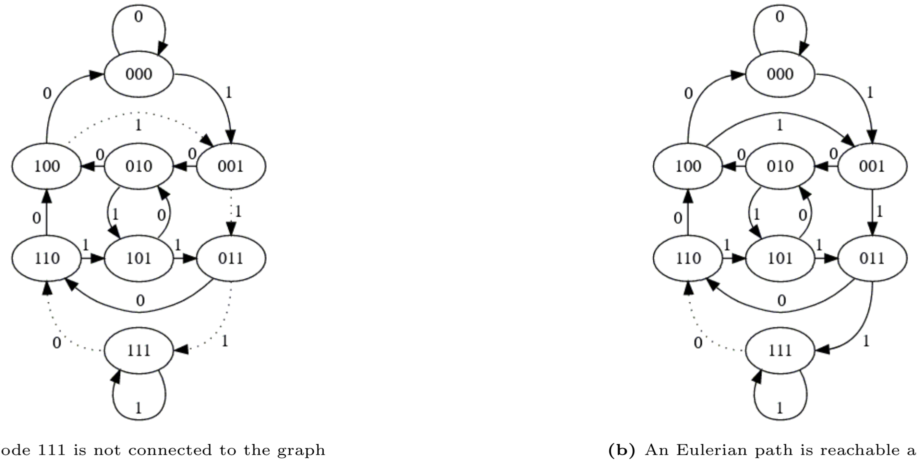
We will apply Proposition 2.1 when the first substring is included in the last substring. It means that we will have to remove the first substring found by the algorithm. In Algorithm 1, we can apply it as a final step (inside function `GetMaxSubStrings`, for instance).

The complexity of the algorithm is given by the predominant use of the FDC: $O(N \lg^3 N)$ in space and $O(N \lg^2 N)$ in time.

3 Example

Let's practice with an example to see the algorithm in action.

In this case, we use $k = 2$, $n = 4$, $N = 2^4 = 16$ and sequence $S = \overset{\substack{\text{LSB}\\ \downarrow}}{1}00111011000001\overset{\substack{\text{MSB}\\ \downarrow}}{0}$. We start with the de Bruijn graph $G(2, 3)$ (Figure 1). Be careful with the least/most significant bit in the sequence, the graph and the algorithm... the order is not important, but to be coherent.



(a) Node 111 is not connected to the graph

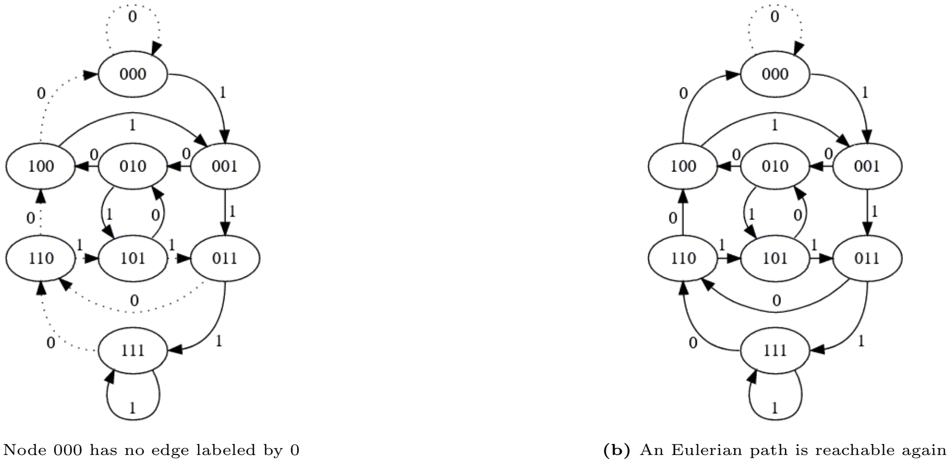
(b) An Eulerian path is reachable again

Figure 2: First substring is 100111.
(a) An Eulerian path is not possible. (b) Path is fixed.

1. First substring
   We start in the initial node 100 and then advance deleting edges labeled by 1, 1, 1 and 0 (Figure 2a). In the latest step, the algorithm will detect that the graph is not

connected anymore. It implies that the first substring is 100111 and the algorithm will restore deleted edges from the beginning of the path until the graph is connected again (Figure 2b).

2. Second substring
   Being the initial node 111 for this string, we continue from latest node 110 and then advance deleting edges labeled by 1, 1, 0, 0, 0, 0 and 0 (Figure 3a). In the latest step, the algorithm will detect that the current node 000 has no existing edge labeled by 0. It means that the second substring is 1110110000 and the algorithm will restore deleted edges from the beginning of the path until one of those edges be the problematic edge (Figure 3b).



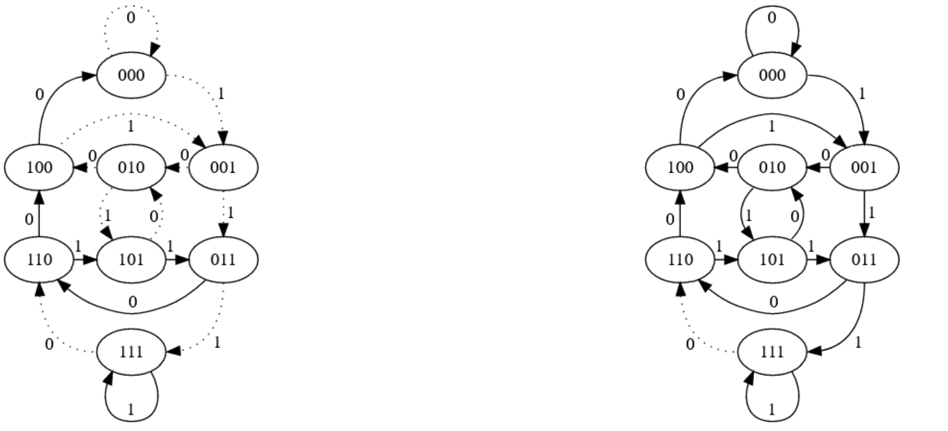
(a) Node 000 has no edge labeled by 0

(b) An Eulerian path is reachable again

Figure 3: Second substring is 1110110000.
(a) Extension of the path with a 0 is not possible. (b) Path is fixed.

3. Third substring
   Being the initial node 000 for this string, we continue from latest node 000 and then advance deleting edges labeled by 1, 0, 1, 0, 0, 1, 1, 1 and 0 (Figure 4a). In the latest step, the algorithm will detect that the graph is not connected anymore. It implies that the third substring is 000010100111 and the algorithm will restore deleted edges from the beginning of the path until the graph is connected again (Figure 4b).

4. End
   As the initial node is beyond the length N of our sequence S, the algorithm stops because there's no more useful substrings for computing L_D.
   In Table 1 we see a little summary of the results; a symbol in bold means the initial symbol of each substring (note that the third substring is cyclic). The value for L_D is the maximum length among all substrings found, so in this case $L_D = \{000010100111\}$ and the length of the LCS is 12 (marked in bold). Besides, we show an example of de Bruijn sequence completing the third substring.



(a) Node 111 is not connected to the graph

(b) An Eulerian path is reachable again

Figure 4: Third substring is 000010100111.
(a) An Eulerian path is not possible. (b) Path is fixed.

Table 1: Substrings found

	1	0	0	1	1	1	0	1	1	0	0	0	0	0	1	0	Length
Substring 1	**1**	0	0	1	1	1	-	-	-	-	-	-	-	-	-	-	6
Substring 2	-	-	-	**1**	1	1	0	1	1	0	0	0	0	-	-	-	10
Substring 3	1	0	0	1	1	1	-	-	-	-	**0**	0	0	0	1	0	**12**
Example	1	0	0	1	1	1	(1	0	1	1)	0	0	0	0	1	0	16

Table 2 shows the length of the maximum substring on each position of sequence S, where the value in bold means the initial length of the substring, and the length of the LCS is marked in bold in the last row.

Table 2: Length of maximum substring by position

	1	0	0	1	1	1	0	1	1	0	0	0	0	0	1	0
Substring 1	**6**	5	4	-	-	-	-	-	-	-	-	-	-	-	-	-
Substring 2	-	-	-	**10**	9	8	7	6	5	4	-	-	-	-	-	-
Substring 3	6	5	4	-	-	-	-	-	-	-	**12**	11	10	9	8	7
Length	6	5	4	10	9	8	7	6	5	4	**12**	11	10	9	8	7

Finally, observe that, according to Proposition 2.1, instead of three substrings we would have only the last two of them (the first one is included in the third substring), but L_D will remain the same.

4 Conclusion

Although, at first glance, the purpose of this text could seem a hard task, thanks to the relation and properties between de Bruijn sequences and graphs we have achieved the goal presenting an algorithm with a non-exponential time complexity.

References

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

[2] P. Charalampopoulos, T. Kociumaka, S.P. Pissis, J. Radoszewski. ”Faster Algorithms for Longest Common Substring”. In 29th Annual European Symposium on Algorithms (ESA 2021), Leibniz International Proceedings in Informatics (LIPIcs), Volume 204, 30:1-30:17, Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2021.
https://doi.org/10.4230/LIPIcs.ESA.2021.30

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

[4] 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.

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