An Efficiently Generated Family of Binary de Bruijn Sequences

Yunlong Zhu\textsuperscript{a}, Zuling Chang\textsuperscript{a,}\textsuperscript{∗}, Martianus Frederic Ezerman\textsuperscript{b}, Qiang Wang\textsuperscript{c}

\textsuperscript{a}School of Mathematics and Statistics, Zhengzhou University, 450001 Zhengzhou, China.
\textsuperscript{b}School of Physical and Mathematical Sciences, Nanyang Technological University, 21 Nanyang Link, Singapore 637371.
\textsuperscript{c}School of Mathematics and Statistics, Carleton University, 1125 Colonel By Drive, Ottawa, Ontario, K1S 5B6, Canada.

Abstract

We study how to generate binary de Bruijn sequences efficiently from the class of simple linear feedback shift registers with feedback function $f(x_0, x_1, \ldots, x_{n-1}) = x_0 + x_1 + x_{n-1}$ for $n \geq 3$, using the cycle joining method. Based on the properties of this class of LFSRs, we propose two new generic successor rules, each of which produces at least $2^{n-3}$ de Bruijn sequences. These two classes build upon a framework proposed by Gabric, Sawada, Williams and Wong in Discrete Mathematics vol. 341, no. 11, pp. 2977–2987, November 2018. Here we introduce new useful choices for the uniquely determined state in each cycle to devise valid successor rules. These choices significantly increase the number of de Bruijn sequences that can be generated. In each class, the next bit costs $O(n)$ time and $O(n)$ space for a fixed $n$.

Keywords: Binary periodic sequence, de Bruijn sequence, feedback shift register, successor rule, cycle joining method.

1. Introduction

A binary de Bruijn sequence of order $n$ is a $2^n$-periodic sequence in which each $n$-tuple occurs exactly once per period. There are $2^{2^n-n}$ such sequences \cite{2}. They have been studied for a long time as they appeared in multiple disguises \cite{21}. More details are supplied in Fredricksen’s survey \cite{13}. Certain families of such sequences have been found useful in far ranging application domains that include bioinformatics, communication systems, coding theory, and cryptography.

One can build de Bruijn sequences of order $n$ from the Hamiltonian paths of an $n$-dimensional de Bruijn graph over 2 symbols. This is equivalent to finding Eulerian cycles of an $(n-1)$-dimensional de Bruijn graph. While a complete enumeration of all such cycles can be done, for example by Fleury’s algorithm \cite{10}, this rather naive approach is highly inefficient in storage requirement. It remains a major objective to strike a good balance between minimizing the computational costs and maximizing the number of sequences that can be explicitly built. On top of this consideration, depending on the specific application domains, additional requirements

\textsuperscript{∗}Corresponding author

Email addresses: zhu2010_5460163.com (Yunlong Zhu), zuling_chang@zzu.edu.cn (Zuling Chang), fredezerman@ntu.edu.sg (Martianus Frederic Ezerman), wang@math.carleton.ca (Qiang Wang)
may be imposed. In cryptography, for instance, the preference is towards de Bruijn sequences with particular linear complexity profiles while in DNA fragment assembly certain substrings may be more or less desirable than others.

A well-known generic construction approach is the cycle joining method (CJM) (see, for examples, [13] and [16]). The main idea of this method is to join all cycles produced from a given Feedback Shift Register (FSR) into a single cycle by interchanging the successors of some pairs of conjugate states. A good number of CJM-based fast algorithms are already in the literature. Most of them produce a very limited number of sequences. Let us sample a few. As was shown in [11], one can generate the granddaddy de Bruijn sequence in \(O(n)\) time and \(O(n)\) space per bit. A related sequence, the grandmaa, was built in [6]. Etzion and Lempel proposed some algorithms to generate de Bruijn sequences based on the pure cycling register (PCR) and the pure summing register (PSR) in [8]. Their algorithms generate a remarkable number, an exponential of \(n\), of sequences at the expense of higher memory requirement to store some special states. Earlier, Huang gave another construction that joins the cycles of the complemented cycling register (CCR) in [18]. Jansen, Franx, and Boekee established a requirement to determine some conjugate pairs in [19], leading to another fast algorithm. In [25], Sawada, Williams, and Wong proposed a simple de Bruijn sequence construction, which turned out to be a special case of the method in [19]. Gabric, Sawada, Williams, and Wong generalized the last two results to form a simple successor rule framework that yield three new and simple de Bruijn constructions based on the PCR and the CCR in [14]. Further generalization to the constructions of \(k\)-ary de Bruijn sequences was done in [18] and [15]. Chang, Ezerman, Ke, and Wang recently proposed a new criteria for successor rules in [4]. They applied the criteria to efficiently construct numerous de Bruijn sequences based on the PCR and the PSR by imposing new relations on the respective generated cycles.

In this paper we provide more successor rules to generate a new family of de Bruijn sequences. We use the CJM to join all cycles generated by a special LFSR whose characteristic polynomial is \(f(x) = x^n + x^{n-1} + x + 1 \in \mathbb{F}_2[x]\) for \(n \geq 3\). The cycles generated by this LFSR correspond to the cycles of the PCR and CCR of order \(n - 1\). Sala, in a Master’s thesis [22], studied this LFSR and proposed a successor rule to generate de Bruijn sequences in \(O(n)\) time and \(O(n)\) space per bit. In a recent preprint [23], Sala, Sawada, and Alhakim noticed that the states in each cycle produced by this LFSR have the same run-length. They then proposed a new successor rule based on the so-called run-length order to generate the famous prefer-same de Bruijn sequence [7] using \(O(n)\) time per bit and only \(O(n)\) space. They named the LFSR the pure run-length register (PRR). This, along with the general framework to generate de Bruijn sequences proposed in [14] and [23], can be found in [24].

Our main contribution in this paper is to construct two new generic successor rules based on the special LFSR by applying the main results of [14]. The number of de Bruijn sequences generated by our new rules is exponential in the order \(n\) of the FSR. Our method runs in \(O(n)\) time and \(O(n)\) space per generated bit. More explicitly, we accomplish the following tasks.

1. We take a different point of view in studying the PRR of order \(n \geq 3\) from the one already done in [22]. Here we discuss the properties of the PRR of order \(n\) via the characteristic polynomials of the PCR and the CCR of order \(n - 1\).

2. Two new generic successor rules are presented based on the PRR to generate de Bruijn sequences. The correctness of the rules are demonstrated by applying the framework proposed in [14]. We introduce the notion of a critical set of spanning conjugate pairs to
effectively define a spanning tree of the cycles induced by the PRR. In each generic rule, a CCR-related cycle has a unique parent while a PCR-related cycle may have a variety of possible parents, depending on the specified critical set.

3. For each generic successor rule, we construct critical sets to uniquely identify the respective parents of the PCR-related cycles efficiently in both time and space. Thus, each rule leads to an exponential number of de Bruijn sequences that can be generated from among the generic successors. Each of the formulated algorithms can generate the next bit of a de Bruijn sequence in $O(n)$ time using $O(n)$ space.

In terms of organization, Section 2 gathers some preliminary notions and useful known results. Sections 3 and 4 provide the treatment on the two classes, respectively. We end with a brief discussion on the required computational resources, a few directions for follow-up investigation, and the C code of our basic implementation.

2. Preliminaries

An $n$-stage shift register is a clock-regulated circuit with the following properties. It has $n$ consecutive storage units. Each unit holds a bit. As the clock pulses the circuit shifts the bit in each unit to the next stage. The register turns into a binary code generator if one appends a feedback loop that outputs a new bit $s_0$ based on the $n$-stage initial state $s_0 = s_0, \ldots, s_{n-1}$. The corresponding Boolean feedback function $f(s_0, \ldots, s_{n-1})$ outputs $s_0$ on input $s_0$. A feedback shift register (FSR), therefore, outputs a binary sequence $s = \{s_i\} = s_0, s_1, \ldots, s_n, \ldots$ that satisfies the recursive relation $s_{n+\ell} = f(s_\ell, s_{\ell+1}, \ldots, s_{\ell+n-1})$ for $\ell = 0, 1, 2, \ldots$.

For $N \in \mathbb{N}$, if $s_{i+N} = s_i$ for all $i \geq 0$, then $s$ is $N$-periodic or with period $N$ and we write $s = (s_0, s_1, s_2, \ldots, s_{n-1})$. The least among all periods of $s$ is called the least period of $s$.

We call $s_i = s_i, s_{i+1}, \ldots, s_{i+n-1}$ the $i$-th state of $s$. The predecessor and the successor of $s_i$ are denoted, respectively, by $s_{i-1}$ and $s_{i+1}$. For $s \in \mathbb{F}_2^n$, let $\bar{s} := s + 1 \in \mathbb{F}_2^n$. The definition extends to any binary vector or sequence. If $s = s_0, s_1, \ldots, s_{n-1}, \ldots$, then $\bar{s} = \bar{s}_0, \bar{s}_1, \ldots, \bar{s}_{n-1}, \ldots$ For an arbitrary state $v = v_0, v_1, \ldots, v_{n-1}$ of $s$, the states $\tilde{v} := \bar{v}_0, \bar{v}_1, \ldots, \bar{v}_{n-1} \text{ and } \bar{v} := v_0, v_1, \ldots, v_{n-1}$ are the conjugate state and companion state of $v$, respectively. Hence, $(v, \tilde{v})$ is a conjugate pair and $(v, \bar{v})$ a companion pair.

Any FSR with feedback function $f$, on distinct $n$-stage initial states, generates distinct sequences that form a set $G(f)$ of cardinality $2^n$. All sequences in $G(f)$ are periodic if and only if the feedback function $f$ is nonsingular, that is, $f(x_0, x_1, \ldots, x_{n-1}) = x_0 + h(x_1, \ldots, x_{n-1})$ for some Boolean function $h(x_1, \ldots, x_{n-1})$ whose domain is $\mathbb{F}_2^{n-1}$ [12, p. 116]. Here we deal only with nonsingular feedback functions. An FSR is linear or an LFSR if its feedback function is $f(x_0, \ldots, x_{n-1}) = x_0 + a_1 x_1 + \cdots + a_{n-1} x_{n-1}$. The polynomial

$$f(x) = x^n + a_{n-1} x^{n-1} + \cdots + a_1 x + 1 \in \mathbb{F}_2[x]$$

is the characteristic polynomial of the LFSR. Otherwise, the FSR is nonlinear or an NLFSR. Further properties of LFSRs are treated in [17] and [20].
The left shift operator $L$ maps a periodic sequence
\[ s := (s_0, s_1, \ldots, s_{N-1}) \mapsto Ls := (s_1, \ldots, s_{N-1}, s_0), \]
with the convention that $L^0$ fixes $s$. The set
\[ [s] := \{ s, ls, \ldots, L^{N-1}s \} \]
is a shift equivalent class. Sequences in the same shift equivalent class correspond to the same cycle in the state diagram of the FSR. A cycle is a periodic sequence in a shift equivalent class. If an FSR with feedback function $f$ generates exactly $r$ distinct cycles $C_1, C_2, \ldots, C_r$, then its cycle structure is
\[ \Omega(f) = \{ C_1, C_2, \ldots, C_r \}. \]

A cycle can also be viewed as a set of $n$-stage states in the corresponding periodic sequence. When $r = 1$, the corresponding FSR is of maximal length and its output is a de Bruijn sequence of order $n$.

The unique lexicographically least $n$-stage state in each cycle $C \in \Omega(f)$ is designated as the cycle representative of $C$ in $\Omega(f)$. One can impose a lexicographic order $\prec_{\text{lex}}$ on the cycles, based on their representatives, by saying that $C_i \prec_{\text{lex}} C_j$ if and only if the cycle representative of $C_i$ is lexicographically smaller than that of $C_j$.

If two distinct cycles $C_i$ and $C_j$ in $\Omega(f)$ have the property that the state $v = v_0, v_1, \ldots, v_{n-1} \in C_i$ has its conjugate state $\hat{v} \in C_j$, then interchanging the successors of $v$ and $\hat{v}$ joins $C_i$ and $C_j$ into a single cycle. The feedback function of this new cycle is
\[
\hat{f} := f(x_0, x_1, \ldots, x_{n-1}) + \prod_{i=1}^{n-1} (x_i + \hat{x}_i).
\]

Similarly, if the companion states $v$ and $\hat{v}$ are in two distinct cycles, then interchanging their predecessors joins the two cycles. If either process continues until all of the cycles in $\Omega(f)$ can be joined into a single cycle, then we obtain a de Bruijn sequence. This construction is the cycle joining method (CJM).

Given an FSR with feedback function $f$, its adjacency graph $G_f$, or simply $G$ if $f$ is clear, is an undirected multigraph whose vertices correspond to the cycles in $\Omega(f)$. Two distinct vertices are adjacent if they share a conjugate (or companion) pair. The number of edges between them is the number of shared conjugate (or companion) pairs, with a specific pair assigned to each edge. We know from [1] that there is a bijection between the set of spanning trees of $G_f$ and the set of all inequivalent de Bruijn sequences constructible by the CJM on input $f$.

We now introduce two simple FSRs that we will often use. The pure cycling register (PCR) of order $n$ is an LFSR with feedback function and characteristic polynomial
\[ f_{\text{PCR}}(x_0, x_1, \ldots, x_{n-1}) = x_0 \text{ and } f_{\text{PCR}}(x) = x^n + 1. \]

Each cycle generated by the PCR of order $n$ is $n$-periodic and has the form $(c_0, c_1, \ldots, c_{n-1})$. The complemented cycling register (CCR) of order $n$ is an NLFSR with feedback function
\[ f_{\text{CCR}}(x_0, x_1, \ldots, x_{n-1}) = x_0 + 1. \]

Each cycle generated by the CCR is $2n$-periodic and has the form
\[ (c_0, c_1, \ldots, c_{n-1}, \bar{c}_0, \bar{c}_1, \ldots, \bar{c}_{n-1}), \]
that is, we always have $c_i = \bar{c}_{i+n}$ for all $i \geq 0$. The weight of a cycle $C$ or a state $v$, denoted respectively by $\text{wt}(C)$ and $\text{wt}(v)$, is the number of 1s in the cycle or the state. It is clear that the weight of each CCR cycle is $n$.

For a given order $n$, the usual cycle representatives of a cycle in $\Omega(f_{\text{PCR}})$ and a cycle in $\Omega(f_{\text{CCR}})$ are called the necklace and the co-necklace, respectively. It is known, for example in [14, Algorithm 2], that testing if a state is the necklace or the co-necklace of a cycle takes $O(n)$ time and $O(n)$ space. Most fast algorithms to generate de Bruijn sequences work on either the PCR or the CCR.

Although the CCR is not linear, the corresponding sequences can be generated by an LFSR. Each sequence $s = (s_0, s_1, \ldots, s_{2n-1})$ of the CCR of order $n-1$, with $n \geq 3$, satisfies

$$s_{n-1+i} = s_i + 1 \text{ and } s_{n+i} = s_{i+1} + 1, \text{ for } i \geq 0.$$ 

Combining them, we obtain the relation

$$s_{n+i} = s_i + s_{i+1} + s_{n-1+i}, \text{ for } i \geq 0.$$ 

Hence, $s$ can be generated by an LFSR of order $n$ with characteristic polynomial and feedback function

$$h(x) = x^n + x^{n-1} + x + 1 = (x^{n-1} + 1)(x+1) \text{ and } h(x_0, x_1, \ldots, x_{n-1}) = x_0 + x_1 + x_{n-1}. \quad (5)$$

The LFSR with characteristic polynomial $h(x)$ has several good properties. The authors of [23] used it to construct a successor rule that generates the prefer-same de Bruijn sequence in $O(n)$ time and $O(n)$ space per bit. They named the said LFSR the pure run-length register (PRR). We henceforth adopt the name and refer to $h(x)$ in Equation (5) as $f_{\text{PRR},n}(x)$. When $n$ is clear in the context, we also use the abbreviation $f_{\text{PRR}}(x)$. The same practice of specifying the order for precision when necessary applies to the other LFSRs.

**Lemma 1.** Each cycle in $\Omega(f_{\text{PRR}})$ of order $n$ is either a PCR cycle or a CCR cycle of order $n-1$.

**Proof.** We have seen that all sequences of the CCR of order $n-1$ can be generated by the PRR of order $n$. Hence, $G(f_{\text{CCR},n-1}) \subseteq G(f_{\text{PRR},n})$. For two LFSRs with respective characteristic polynomials $f_1(x)$ and $f_2(x)$, one has $G(f_1) \subseteq G(f_2)$ if and only if $f_1(x)$ divides $f_2(x)$ [17, Lemma 4.2 (a)]. Since $(x^{n-1} + 1)$ divides $f_{\text{PRR},n}(x)$, we have $G(f_{\text{PCR},n-1}) \subseteq G(f_{\text{PRR},n})$. Hence,

$$G(f_{\text{PRR},n-1}) \cup G(f_{\text{CCR},n-1}) \subseteq G(f_{\text{PRR},n}).$$

Since $G(f_{\text{PCR},n-1})$ and $G(f_{\text{CCR},n-1})$ are disjoint, with $|G(f_{\text{PCR},n-1})| = |G(f_{\text{CCR},n-1})| = 2^{n-1}$, and $|G(f_{\text{PRR},n})| = 2^n$, it is clear that

$$G(f_{\text{PRR},n}) = G(f_{\text{PRR},n-1}) \cup G(f_{\text{CCR},n-1}).$$

Thus, each cycle in $\Omega(f_{\text{PRR}})$ is a cycle either in $\Omega(f_{\text{PCR},n-1})$ or in $\Omega(f_{\text{CCR},n-1})$. \qed

**Remark 1.** Let an $n$-stage state $c_0, c_1, \ldots, c_{n-1}$ of a cycle $C$ in $\Omega(f_{\text{PRR}})$ be given. If $c_0 = c_{n-1}$, then, by the proof of Lemma 1, $C$ is a PCR cycle. Otherwise, $C$ is a CCR cycle. This is why we sometimes refer to the PRR in [23] as a pure and complemented cycling register (PCCR).
If \( \phi(\cdot) \) is the Euler totient function, then the number of cycles in \( \Omega(f_{\text{PRR},n}) \) is
\[
Z_{n} = Z_{n-1} + Z_{n-1}',
\]
where
\[
Z_{n-1} = \frac{1}{n-1} \left( \sum_{d | (n-1)} \phi(d) 2^{\frac{n-1}{d}} \right) \quad \text{and} \quad Z_{n-1}' = \frac{1}{2(n-1)} \left( \sum_{d | (n-1)} \phi(d) 2^{\frac{n-1}{d}} \right) \tag{6}
\]
are the respective number of cycles in \( \Omega(f_{\text{PCR},n-1}) \) and in \( \Omega(f_{\text{CCR},n-1}) \) \cite{13}. A proof for the formula of \( Z_{n} \) and a sketch of the proof for the formula of \( Z_{n}' \) were both due to Golomb \cite{16}. A more thorough discussion was supplied by Sloane in \cite{27, Section 3}.

We partition the cycles in \( \Omega(f_{\text{PRR},n}) \) into two parts, namely the PCR cycles \( \mathcal{P}_{1}, \ldots, \mathcal{P}_{Z_{n-1}} \) and the CCR cycles \( \mathcal{C}_{1}, \ldots, \mathcal{C}_{Z_{n-1}'} \). Excluding the cycle \( (1^{n-1}) \) whose representative is obviously \( 1^{n} \), the other cycle representatives must be \( n \)-stage states whose forms are either
\[
0, c_{1}, \ldots, c_{n-2}, 0 \quad \text{or} \quad 0, c_{1}, \ldots, c_{n-3}, 0, 1,
\]
where \( 0, c_{1}, \ldots, c_{n-2} \) is either the necklace in a PCR cycle or the co-necklace in a CCR cycle.

The PRR has yet another interesting property. The run-length encoding of an \( n \)-stage state \( v = v_{0}, v_{1}, \ldots, v_{n-1} \) is a compressed representation that stores, consecutively, the lengths of the maximal runs of each element. The run-length of \( v \) is the length of its run-length encoding. For example, the state \( 0001011101 \) has run-length encoding \( 311311311311311 \) and run-length 6. The following lemma was established in \cite{23}.

**Lemma 2.** All states in a cycle in \( \Omega(f_{\text{PRR},n}) \) have the same run-length for a given \( n \).

**Example 1.** Let \( n = 6 \). The 12 cycles in \( \Omega(f_{\text{PRR}}) \) consists of the 8 cycles generated by the PCR of order 5, namely
\[
\mathcal{P}_{1} := (00000), \mathcal{P}_{2} := (00001), \mathcal{P}_{3} := (00011), \mathcal{P}_{4} := (00101), \mathcal{P}_{5} := (00111), \mathcal{P}_{6} := (01011), \mathcal{P}_{7} := (01111), \mathcal{P}_{8} := (11111),
\]
and the 4 cycles generated by the CCR of order 5, namely
\[
\mathcal{C}_{1} := (0000011111), \mathcal{C}_{2} := (0001011101), \mathcal{C}_{3} := (0010011101), \mathcal{C}_{4} := (010101101).
\]
The cycles are presented in increasing lexicographical order within their respective types as
\[
\mathcal{P}_{1} \prec_{\text{lex}} \mathcal{P}_{2} \prec_{\text{lex}} \ldots \prec_{\text{lex}} \mathcal{P}_{8} \quad \text{and} \quad \mathcal{C}_{1} \prec_{\text{lex}} \mathcal{C}_{2} \prec_{\text{lex}} \ldots \prec_{\text{lex}} \mathcal{C}_{4}.
\]
The cycle representatives of \( \mathcal{P}_{1}, \ldots, \mathcal{P}_{8} \) and \( \mathcal{C}_{1}, \ldots, \mathcal{C}_{4} \) are, in that order,
\[
000000, 000010, 000110, 001010, 001110, 010110, 011110, 111111, 000001, 000101, 001001, 010101.
\]
In \( \mathcal{C}_{2} \), there are 10 distinct 6-stage states:
\[
000101, 001011, 010111, 011110, 111010, 110100, 101000, 010001, 100010, 100100
\]
and their run-lengths are, respectively,
\[
3111, 2112, 1113, 1131, 1311, 3111, 2112, 1113, 1131, 1311.
\]
All of them have the same run-length 4.
Gabric et al. in [14] and Sawada et al. in [25] proposed several fast algorithms to generate de Bruijn sequences. In those papers they ordered the cycles in $\Omega(f_{PCR})$ and in $\Omega(f_{CCR})$ lexicographically according to how each cycle’s necklace or co-necklace compares to one another, respectively. In each case, they replace the usual FSR-based generating algorithm by a well-chosen mechanism, called the successor rule $\rho(x_0, x_1, \ldots, x_{n-1})$, to generate the next bit. Given an FSR with a feedback function $f(x_0, x_1, \ldots, x_{n-1})$, the general thinking behind this approach is to determine some condition which guarantees that the resulting sequence is de Bruijn.

Let $A$ be a nonempty subset of the set of all states. For each state $c = c_0, c_1, \ldots, c_{n-1}$, a nontrivial successor rule $\rho$ is an assignment

$$\rho(c_0, c_1, \ldots, c_{n-1}) := \begin{cases} f(c_0, \ldots, c_{n-1}) & \text{if } c \in A, \\ f(c_0, \ldots, c_{n-1}) & \text{if } c \notin A. \end{cases}$$

(7)

To be precise, the successor of $c = c_0, c_1, \ldots, c_{n-1}$ is $c_1, \ldots, c_{n-1}, f(c_0, \ldots, c_{n-1})$ except when $c \in A$. When $c \in A$, the successor is $c_1, \ldots, c_{n-1}, f(c_0, \ldots, c_{n-1}) + 1$, that is, the last bit of the successor is the complement of the last bit of the successor when $c \notin A$. Our interest is to characterize a set $A$ whose corresponding successor rule ensures that the cycles can be joined to form a de Bruijn sequence. A framework to judge whether a successor rule can generate a de Bruijn sequence was provided in [14, Section 3]. We restate their result in our notation as follows.

**Theorem 3** (Theorem 3.5 in [14]). Let $\prec$ denote a valid order on the cycles of an FSR whose feedback function is $f(x_0, x_1, \ldots, x_{n-1})$. Let the cycles be ordered as $C_1 \prec C_2 \prec \ldots \prec C_r$. Let $A$ be a set that contains all states which constitute $r - 1$ conjugate pairs with the following properties.

For each cycle $C_i$ with $1 < i \leq r$, there exists a unique conjugate pair $(c, \hat{c})$ such that $c$ is in $C_i$, $\hat{c}$ is in $C_j$, where $j < i$, and both $c, \hat{c} \in A$. The successor rule $\rho$ in Equation (7) generates a de Bruijn sequence.

The theorem states that, if one can define a suitable order on the cycles generated by a given FSR, then a successor rule can be devised to generate a de Bruijn sequence by constructing a spanning tree.

If in each cycle $C_i$ with $1 < i \leq r$ one can uniquely determine a state whose conjugate state is in another cycle $C_j \prec C_i$, then the successor rule interchanges their respective successor states. The successor of any other state which is not identified as either the unique state in each $C_i$ or its conjugate state in $C_j$ remains to be the one assigned by the FSR with feedback function $f$. This process joins cycles $C_i$ and $C_j$. All of the cycles are eventually joined into a single cycle as $i$ goes from 2 to $r$.

The set $A$ in Theorem 3 determines $r - 1$ conjugate pairs that correspond to $r - 1$ edges in the adjacency graph $G$ that has $r$ vertices. If an edge connects two distinct cycles $C_i$ and $C_j$ with $r \geq i > j \geq 1$, then the direction of this edge is from $C_i$ to $C_j$. The condition guarantees that there is a unique path from each $C_i$ to $C_1$ for $1 < i \leq r$ and a rooted spanning tree is explicitly constructed. This ensures that the generated sequence is a de Bruijn sequence. Henceforth, we call such a set $A$ a critical set of spanning conjugate pairs or a critical set, in short.

The following result from [19] is useful to tell when a given cycle is lexicographically less than another cycle.

**Lemma 4.** [19] If $C$ is a cycle of an FSR whose cycle representative is a nonzero state $v$, then the companion state $\tilde{v}$ is in another cycle $C'$ with $C' \prec_{\text{lex}} C$. 

7
Sala used the PRR of order $n$ to construct a successor rule that generates a de Bruijn sequence in $O(n)$ time and $O(n)$ space per bit in [22]. The original proof was quite involved. Here we recall the result and supply a simpler proof.

**Theorem 5.** [22] For each state $c = c_0, c_1, \ldots, c_{n-1}$, let $\mathbf{u}_c := c_1, c_2, \ldots, c_{n-1}$. The successor rule

\[
\zeta(c) = \begin{cases} 
  c_0 + c_1 + c_{n-1} & \text{if } c \text{ is a necklace or a co-necklace}, \\
  c_0 + c_1 + c_{n-1} & \text{otherwise},
\end{cases}
\]

(8)

generates a de Bruijn sequence of order $n$.

**Proof.** Let the cycles in $\Omega(f_{\text{PRR},n})$ be ordered as $(0^n) = C_0 \prec_{\text{lex}} C_1 \prec_{\text{lex}} \ldots \prec_{\text{lex}} C_{2^n-1}$. For a given state $c$ in some $C_i$ with $i > 0$, let $c_n := c_0 + c_1 + c_{n-1}$ and $v := c_1, c_2, \ldots, c_n$. As we have discussed earlier, the $(n-1)$-stage state $\mathbf{u}_c$ is a necklace or a co-necklace if and only if the $n$-stage state $v$ is the cycle representative of $C_i$, which is unique. By Lemma 4 the companion state $\hat{v}$ must be in $C_{j} \neq C_i$ with $C_{j} \prec_{\text{lex}} C_{i}$, that is, $j < i$. Collecting the two states in the conjugate pair $(c, \hat{c})$ as $i$ ranges from 1 to $2^n - 1$ yields a critical set $A$. The desired conclusion follows.

Now that prior arts have been covered, we are ready to present our new successor rules. Each of the next two sections introduces a new generic class of successor rules. We use Theorem 5 to confirm that these two classes generate exponentially many de Bruijn sequences.

### 3. The first class of successor rules from the PRR

We begin by giving a general formula for successor rules to generate de Bruijn sequences based on the PRR before defining a new class of successor rules explicitly.

**Theorem 6.** For each state $c = c_0, c_1, \ldots, c_{n-1}$ produced by the PRR of order $n \geq 3$, we define the state $\mathbf{v}_c := c_1, \ldots, c_{n-1}, 1$. The states forming the conjugate pair $(c, \hat{c})$ belong to a critical set $A_\Psi$ if one of the followings holds.

(i) The state $\mathbf{v}_c$ is the cycle representative of a CCR cycle $\Psi$ if $c_1 = 0$.

(ii) The state $\mathbf{v}_c$ can be uniquely determined in a PCR cycle $\mathcal{P}$ if $c_1 = 1$.

The successor rule

\[
\Psi(c) := \begin{cases} 
  c_0 + c_1 + c_{n-1} & \text{if } c \in A_\Psi, \\
  c_0 + c_1 + c_{n-1} & \text{otherwise},
\end{cases}
\]

(9)

generates a de Bruijn sequence of order $n$.

**Proof.** Suppose that the cycles in $\Omega(f_{\text{PRR}})$ have been ordered lexicographically as

\[(0^{n-1}) = C_0 \prec_{\text{lex}} C_1 \prec_{\text{lex}} \ldots \prec_{\text{lex}} C_{2^n-1}.
\]

The state $\mathbf{v}_c$ that satisfies either one of the above conditions, depending on the value of $c_1$, determines a conjugate pair $(c, \hat{c})$. For $1 \leq i < 2^n$, Theorem 5 requires us to show that each $C_i$ has a uniquely identified state whose conjugate state is in $C_j$, with $C_{j} \prec_{\text{lex}} C_i$. This is equivalent to showing that each $C_i$ has a uniquely identified state whose companion state is in $C_j$, with
$C_i \prec_{\text{lex}} C_j$. We note that each $C_i$ must contain at least one state whose last bit is 1, that is, $C_i$ always contains a state that can serve as $v_c$.

If $v_c$ is the cycle representative of a CCR cycle $C$, then it is uniquely determined and its companion state $v_c$ must be in a PCR cycle $P$. By Lemma 4, we have $P \prec_{\text{lex}} C$.

Let $v_c$ be a uniquely determined state in a PCR cycle $P$. If $P = (1^{n-1})$, then $v_c = 1^n$ and $v_c = 1^{n-1}0$ is in the CCR cycle $(0^{n-1}1^n) \prec_{\text{lex}} (1^{n-1}) = P$. If $P \neq (1^{n-1})$, then the unique cycle representative of $P$ begins with a 0 and has the form

$$c_j, \ldots, c_{n-1}, 1, c_2, \ldots, c_j$$

for some $j$ in the range $2 \leq j < n$. The state $v_c$ is in a CCR cycle $C$ that contains the state

$$c_j, \ldots, c_{n-1}, 0, c_2, \ldots, c_j,$$

which is clearly lexicographically less than the cycle representative of $P$. Hence, $C \prec_{\text{lex}} P$. Thus, $\Psi$ generates a de Bruijn sequence less than the cycle representative of $P$. Hence, $C \prec_{\text{lex}} P$. 

We emphasize that the uniquely identified state in each CCR cycle must be the cycle representative. The uniquely determined state in each nonzero PCR cycle can be any state $v_c$ that can serve as $v_c$. Different ways to uniquely identify $c_1 = 1, c_2, \ldots, c_{n-1}, 1$ in each PCR cycle yield distinct successor rules. Each such rule generates a de Bruijn sequence. The rest of this section supplies two clusters of successor rules based on concrete choices for the critical set $\Lambda$.

Since $c_1 = 1$, one strategy is to uniquely identify the $(n-1)$-stage state $c_2, \ldots, c_{n-1}, 1$ in any nonzero cycle produced by the PCR of order $n-1$ with respect to its necklace. Every state in a PCR cycle can be transformed into the necklace by repeated left shift operations. Now we define a new operator $\Lambda$ which consists of a sequence of left shift operations so that the first 1 is cyclically left-shifted to the end. Formally, given a nonzero state $u_c = c_1, c_2, \ldots, c_{n-1}$, we choose $i$ with $1 \leq i < n$ to be the smallest index for which $c_i = 1$. We define

$$\Lambda u_c := c_{i+1}, \ldots, c_{n-1}, c_1, \ldots, c_i$$

and use the notation $\Lambda^j u_c$ to mean $\Lambda^{j-1}(\Lambda u_c)$, with $\Lambda^0 u_c := u_c$. In a PCR cycle, $\Lambda$ transforms $u_c$ to the necklace after at most $j$ steps, where $j$ is the weight of the cycle. In (01011), for example, if $u_c = 01101$, then $\Lambda u_c = 10101$ and $\Lambda^2 u_c = 01101$ is already the necklace.

All ingredients to explicitly construct successor rules in the class $\Psi$ are now in place and distinct ways to determine the desired state in any PRR cycle can be explicitly written.

**Theorem 7.** Let positive integers $n$, $t$, and $k_1, \ldots, k_t$ be such that

$$n \geq 3, \quad 2 \leq t \leq n-1, \quad 1 = k_1 < k_2 < \ldots < k_t = n, \quad \text{and } k_{t-1} < n-1.$$

For each state $c = c_0, c_1, \ldots, c_{n-1}$, let $u_c := c_1, c_2, \ldots, c_{n-1}$. The successor rule

$$\Psi_1(c) = \begin{cases} 
\bar{c}_0 + c_1 + c_{n-1} & \text{if } c_1 = 0 \text{ and } u_c \text{ is a co-necklace}, \\
c_0 + c_1 + c_{n-1} & \text{if } c_1 = 1 \text{ and there is an } i \text{ such that } k_i \leq wt(u_c) < k_{i+1} \text{ and } \Lambda^i u_c \text{ is a necklace}, \\
c_0 + c_1 + c_{n-1} & \text{otherwise,}
\end{cases}$$

(10)

generates de Bruijn sequences of order $n$. 


Proof. We prove correctness by showing that the conditions required by Theorem 6 are satisfied. Let \( v_c := u_c, 1 = c_1, c_2, \ldots, c_{n-1}, 1 \) and \( x_c := 0, u_c = 0, c_1, c_2, \ldots, c_{n-1} \). If \( c_1 = 0 \), then \( v_c \) is the cycle representative of a CCR cycle \( \mathcal{C} \) of order \( n \) if and only if \( u_c = 0, c_2, \ldots, c_{n-1} \) is the co-necklace of \( \mathcal{C} \). If \( c_1 = 1 \), then \( v_c \) is a uniquely determined state in a PCR cycle \( \mathcal{P} \) of order \( n \) if and only if \( \Lambda u_c = Lu_c = c_2, \ldots, c_{n-1}, 1 \) is uniquely determined in \( \mathcal{P} \). Since \( k_l \leq \text{wt}(u_c) < k_{l+1} \) and \( \Lambda^k u_c \) is the necklace of \( \mathcal{P} \), we confirm that \( u_c \) is uniquely determined.

Because the weight of the PCR cycle \((1^{n-1})\) is \( n - 1 \), we always take \( k_l = n \) for consistency in formulating the successor rule. Letting \( k_{l-1} = n - 1 \) does not add any value since \((1^{n-1})\) has just one state. The number of inequivalent de Bruijn sequences that Theorem 7 generates is established as the next result.

**Proposition 8.** Taking all valid parameters \( t \) and \( \{k_1, k_2, \ldots, k_l\} \) in the statement of Theorem 7, the number of de Bruijn sequences generated by \( \Psi_1 \) in Equation (10) is \( 2^{n-3} \).

**Proof.** For each choice of \( t \) and \( \{k_1, k_2, \ldots, k_l\} \), with \( k_1 = 1, k_{l-1} < n - 1 \), and \( k_l = n \), we obtain a critical set of conjugate pairs to join all of the cycles. Distinct choices of \( t \) and \( \{k_1, k_2, \ldots, k_l\} \) yield inequivalent de Bruijn sequences. The total number of choices is

\[
\binom{n-3}{0} + \binom{n-3}{1} + \cdots + \binom{n-3}{n-3} = \sum_{j=0}^{n-3} \binom{n-3}{j} = 2^{n-3}.
\]

The next theorem presents another way to uniquely identify a state in each cycle and obtain the corresponding successor rule.

**Theorem 9.** Let \( \Delta := \text{lcm}(1, 2, \ldots, n - 2) \) and \( 1 \leq k \leq \Delta \). For each state \( c = c_0, c_1, \ldots, c_{n-1}, 1 \), let \( u_c := c_1, c_2, \ldots, c_{n-1} \). The successor rule

\[
\Psi_2(c) = \begin{cases} 
c_0 + c_1 + c_{n-1} & \text{if } c_1 = 0 \text{ and } u_c \text{ is a co-necklace}, 
c_0 + c_1 + c_{n-1} & \text{if } c_1 = 1 \text{ and } \Lambda^k u_c \text{ is a necklace}, 
c_0 + c_1 + c_{n-1} & \text{otherwise}, 
\end{cases}
\]

generates a total of

\[
\text{lcm}(1, 2, \ldots, n - 2) \geq 2^{n-3}
\]
de Bruijn sequences of order \( n \) by varying \( k \).

**Proof.** The proof of the first part is similar to the proof of Theorem 7 and is, therefore, omitted here. To enumerate these de Bruijn sequences, we note that the weights of the PCR cycles vary from 1 to \( n - 2 \), except for \((1^{n-1})\) which has only one state. The weight of each \( \mathcal{P} \neq (1^{n-1}) \) gives the number of states that matters. For any given \( k > 0 \), let \( k' = k - 1 \) and \( w_c = \Lambda^k u_c = Lu_c \). We consider the system of congruences

\[
\{k' \equiv a_i \pmod{i} \mid 1 \leq i \leq (n-2)\}.
\]

Applying \( \Psi_2 \), if \( \Lambda^k u_c = \Lambda^{k'} w_c \) is a necklace, then \( \Lambda^k w_c \) is the necklace in a cycle of weight \( i \). Hence, \( a_i \) uniquely identifies a state in the cycle. So each possible choice of \( \{a_1, \ldots, a_{n-2}\} \) uniquely identifies a collection of states, each belonging to a cycle. By the Chinese Remainder
Theorem [5], the number of distinct choices of \( \{a_1, \ldots, a_{n-2}\} \) is \( \text{lcm}(1, 2, \ldots, n-2) \), which is obtained as \( k \) runs from 1 to \( \Delta \). Hence, there are \( \text{lcm}(1, 2, \ldots, n-2) \) de Bruijn sequences. By [9, Section 2], we obtain

\[
\text{lcm}(1, 2, \ldots, n-2) \geq (n-2) \left( \frac{n-3}{n-4} \right) \geq 2^{n-3}.
\]

(15)

Example 2. We continue from Example 2 to consider \( \Psi \) for \( n = 6 \). Theorem 7 provides 8 distinct successor rules. The resulting 8 distinct de Bruijn sequences are listed in the first part of Table 1. Applying Theorem 9, again for \( n = 6 \), yields the 12 distinct de Bruijn sequences listed in the second part of Table 1. For ease of comparison, the initial state is fixed to be 000000.

We note that Theorems 7 and 9 may produce equivalent sequences. In Table 1, the sequences in entries 1, 2, 3, and 8 based on Theorem 7 are the same as the sequences in entries 2, 3, 4, and 1 based on Theorem 9, respectively. Table 1 contains 16 distinct de Bruijn sequences in total.

| Entry \( \{k_1, k_2, \ldots, k_t\} \) | The resulting sequence based on Theorem 7 | The resulting sequence based on Theorem 9 |
|------------------------------------|------------------------------------------|------------------------------------------|
| 1 \{1, 6\}                        | 0000001111110001011101100011001101010101001 | 0000001111110001011101100011001101010101001 |
| 2 \{1, 2, 6\}                     | 0000001111110001011101100011001101010101001 | 0000001111110001011101100011001101010101001 |
| 3 \{1, 3, 6\}                     | 0000001111110001011101100011001101010101001 | 0000001111110001011101100011001101010101001 |
| 4 \{1, 4, 6\}                     | 0000001111110001011101100011001101010101001 | 0000001111110001011101100011001101010101001 |
| 5 \{1, 2, 3, 6\}                  | 0000001111110001011101100011001101010101001 | 0000001111110001011101100011001101010101001 |
| 6 \{1, 2, 4, 6\}                  | 0000001111110001011101100011001101010101001 | 0000001111110001011101100011001101010101001 |
| 7 \{1, 3, 4, 6\}                  | 0000001111110001011101100011001101010101001 | 0000001111110001011101100011001101010101001 |
| 8 \{1, 2, 3, 4, 6\}               | 0000001111110001011101100011001101010101001 | 0000001111110001011101100011001101010101001 |

There might be other ways to find more critical sets by uniquely identifying a state whose last bit is 1 in each of the PCR cycles, leading to more successor rules. Interested readers are invited to invent their favourites.

4. The second class of successor rules from the PRR

In this section we discuss another class of successor rules.
Theorem 10. Let $n \geq 3$ be given. For each state $c = c_0, c_1, \ldots, c_{n-1}$ produced by $f_{PRR}$, we define $x_c$ based on the value of $c_{n-1}$ as

$$x_c := \begin{cases} 
0, c_1, c_2, \ldots, c_{n-1} & \text{if } c_{n-1} = 0, \\
\hat{c}_1, \hat{c}_2, \ldots, \hat{c}_{n-2}, 0, c_1 & \text{if } c_{n-1} = 1.
\end{cases}$$

The states forming the conjugate pair $(c, \hat{c})$ belong to a critical set $A_T$ if exactly one of the following conditions is satisfied.

(i) The state $x_c$ is a uniquely determined state whose first bit is 0 in a PCR cycle $\mathcal{P}$ if $c_{n-1} = 0$.

(ii) The state $x_c$ is the cycle representative of a CCR cycle $\mathcal{C}$ if $c_{n-1} = 1$.

The successor rule

$$Y(c) := \begin{cases} 
\overline{c_0 + c_1 + c_{n-1}} & \text{if } c \in A_T, \\
\overline{c_0 + c_1 + c_{n-1}} & \text{otherwise.}
\end{cases}$$

(16)

generates a de Bruijn sequence of order $n$.

Proof. We prove that we can define an order $\prec$ on all cycles of the PRR and $A_T$ is a critical set. For any two distinct CCR cycles $\mathcal{C}_i$ and $\mathcal{C}_j$, we define $\mathcal{C}_i \prec \mathcal{C}_j$ if $\mathcal{C}_i \prec_{\text{lex}} \mathcal{C}_j$. This allows us to order all of the CCR cycles lexicographically as

$$(0^{n-1}1^{n-1}) = \mathcal{C}_1 \prec \mathcal{C}_2 \prec \cdots \prec \mathcal{C}_i \prec_{\text{lex}} \mathcal{C}_j \prec \cdots \prec \mathcal{C}_t := Z_{n-1}^t.$$

Our next task is to include the other cycles. It is clear that each cycle, except for $(1^{n-1})$, has a unique state $x_c$ satisfying exactly one of the two conditions (i) and (ii).

If $x_c$ satisfies condition (ii), then $x_c$ belongs to a CCR cycle $\mathcal{C}$. Hence, either $c$ or $\hat{c}$ belongs to the same CCR cycle $\mathcal{C}$. Without loss of generality, we assume $c \in \mathcal{C}$. If $\mathcal{C} = (0^{n-1}1^{n-1})$, which is the lexicographically least among all of the CCR cycles, then the cycle representative is $x_c = 0^{n-1}1$. In this case, we have $c = 01^{n-1}$ and, thus, $\hat{c}$ is in the PCR cycle $(1^{n-1})$. We order $(1^{n-1}) \prec (0^{n-1}1^{n-1})$. If $c \in \mathcal{C} \neq (0^{n-1}1^{n-1})$, then $c_{n-1} = 1$ and $c_0 = 0$. Since $x_c = \hat{c}_1, \hat{c}_2, \ldots, \hat{c}_{n-2}, 0, c_1$ is the cycle representative of a CCR cycle $\mathcal{C}$, we must have $\hat{c}_1 = 0$ and, hence, $c_1 = c_{n-1} = 1$. Thus, $\hat{c}$ must be in some PCR cycle $\mathcal{P} = (1, c_1, \ldots, c_{n-1})$ and we say that $\mathcal{C}$ is adjacent to $\mathcal{P}$.

On the other hand, if $x_c = c$ satisfies condition (i), then $c$ belongs to a PCR cycle $\mathcal{P}$. Hence, $\hat{c}$ is in a CCR cycle $\mathcal{C}'$. When this is the case, we say that $\mathcal{P}$ is adjacent to $\mathcal{C}'$.

If a CCR cycle $\mathcal{C} \neq (0^{n-1}1^{n-1})$ is adjacent to a PCR cycle $\mathcal{P}$ and $\mathcal{P}$ is adjacent to another CCR cycle $\mathcal{C}'$, then $\mathcal{C}' \prec_{\text{lex}} \mathcal{C}$. From the above discussion, if $c = c_0, c_1, \ldots, c_{n-1} \in \mathcal{C}$, then $\hat{c} = 1, c_1, \ldots, c_{n-1}$ is a state in $\mathcal{P}$. A shift of $\hat{c}$, say $u := c_j, c_{j+1}, \ldots, c_{n-2}, 1, \ldots, c_j$, with $c_j = 0$ for some $2 \leq j < n - 1$, can then be uniquely determined in $\mathcal{P}$. Its conjugate state $u := c_j, c_{j+1}, \ldots, c_{n-2}, 1, \ldots, c_j$ is therefore in a CCR cycle $\mathcal{C}'$. We next consider a shift $u$ of $\hat{u}$, say $w := c_1, c_2, \ldots, c_{j+1}, \ldots, c_{n-2}, 1$, which must also be in $\mathcal{C}'$. Because $c_j = 1$, we know that $w$ is lexicographically less than the cycle representative $x_c = \hat{c}_1, \hat{c}_2, \ldots, \hat{c}_{n-2}, 0, c_1$ of $\mathcal{C}$. Hence $\mathcal{C}' \prec_{\text{lex}} \mathcal{C}$. In this case, we order $\mathcal{C}' \prec \mathcal{P} \prec \mathcal{C}$.

We can now define the order among all cycles of the PRR. We note that each PCR cycle $\mathcal{P} \neq (1^{n-1})$ contains a uniquely determined state $c = x_c$. Hence, there exists a unique CCR cycle $\mathcal{C}$ containing $\hat{c}$ such that $\mathcal{P}$ is adjacent to $\mathcal{C}$ and $\mathcal{C} < \mathcal{P}$. [12]
Figure 1: Above: A typical rooted tree based on the successor rule $\Upsilon$, with $\mathscr{E}_1 = (0^{n-1}1^n)$ and $\mathcal{P}_{1,0} = (0^{n-1})$. Letting $t := Z_{n-1}$, the CCR cycles are arranged in decreasing lexicographic order $c_{i-1} \succ c_{i-2} \succ \cdots \succ c_1$ from left to right. It may be the case that there are more than one CCR cycles, say $C_i \in 011101$ itself. Based on the respective cycle representatives of the CCR cycles, we use as our edge to a common PCR cycle.

Below: A rooted tree when $n = 6$, using the cycles specified in Example 1. In each PCR cycle $\mathcal{P} \neq (111111)$, its cycle representative is chosen as the uniquely determined state $\mathbf{x}_c$, which happens to be the state $\mathbf{e}$ itself. Based on the respective cycle representatives of the CCR cycles, we use as our $c \in A_T$ the state $011111 \in \mathcal{E}_1$, $011101 \in \mathcal{E}_2$, $011011 \in \mathcal{E}_3$, and $010101 \in \mathcal{E}_4$.

For $i$ from 1 to $t$, we collect all PCR cycles $\mathcal{P}_{i,1}, \ldots, \mathcal{P}_{i,s_i}$ that are adjacent to $\mathcal{E}_i$. Obviously, $\mathcal{E}_i \prec \mathcal{P}_{i,j}$. Moreover, when $\mathcal{E}_{i+1}$ exists, we fix $\mathcal{P}_{i,j} \prec \mathcal{E}_{i+1}$. The order on $\mathcal{P}_{i,1}, \ldots, \mathcal{P}_{i,s_i}$ can be chosen arbitrarily. The cycles produced by the PRR are finally ordered as

$$\langle 1^{n-1} \rangle \prec \mathcal{E}_1 \prec \langle \text{all PCR cycles adjacent with } \mathcal{E}_1 \text{ in any order} \rangle \prec \mathcal{E}_2 \prec \langle \text{all PCR cycles adjacent with } \mathcal{E}_2 \text{ in any order} \rangle \prec \mathcal{E}_3 \prec \cdots \prec \mathcal{E}_i \prec \langle \text{all PCR cycles adjacent with } \mathcal{E}_i \text{ in any order} \rangle.$$  (17)

The above procedure ensures that $A_T$ is indeed a critical set. Each conjugate pair $(\mathbf{c}, \hat{\mathbf{c}})$ contributes $\mathbf{c}$ and $\hat{\mathbf{c}}$ to $A_T$. Again, either $\mathbf{c}$ or $\hat{\mathbf{c}}$ belongs to the same cycle as $\mathbf{x}_c$ does. Without loss of generality, we assume $\mathbf{c}$ and $\mathbf{x}_c$ belong to the same cycle. Then the cycle containing both $\mathbf{x}_c$ and $\mathbf{c}$ is greater than the cycle containing $\hat{\mathbf{c}}$ in our newly defined order. By Theorem 3 the successor rule generates a de Bruijn sequence.

We observe from the proof of Theorem 10 that ordering all cycles in a line is not strictly necessary. We can in fact still be able to generate a de Bruijn sequence as long as all pairs of cycles containing the conjugate pairs in the set are comparable so that we can find a critical set of states that form the conjugate pairs. This means that we can slightly generalize Theorem 3 by relaxing the condition that all cycles must be ordered. A typical rooted tree can be constructed following the successor rule $\Upsilon$ in the proof of Theorem 10. We demonstrate it in Figure 1.

By Theorem 10 distinct ways of determining the unique state whose first bit is 0 in any PCR cycle $\mathcal{P}$ lead to different successor rules, generating inequivalent de Bruijn sequences. Similar with the operator $\Lambda$, we define an operator $\Theta$ that fixes $1^{n-1}$ and $01^{n-2}$. In all other cases, if $\mathbf{u} := c_1, c_2, \ldots, c_{n-1}$ and $i$ is the least index for which $c_i = 0$, with $2 \leq i < n$, then

$$\Theta \mathbf{u} = c_1, \ldots, c_{i-1}, c_{i+1}, \ldots, c_{n-1},$$

where $\Theta^0 \mathbf{u} := \mathbf{u}$ and $\Theta^i \mathbf{u} := \Theta^{i-1}(\Theta \mathbf{u})$.

The next two theorems construct numerous explicit successor rules based on Theorem 10. Their respective proofs follow the same line of argument as those of Theorems 7 and 9 and are omitted here for brevity.
Theorem 11. Let positive integer \( n, t, \) and \( k_1, \ldots, k_t \) be such that
\[
\begin{align*}
&n \geq 3, \quad 2 \leq t \leq n - 1, \quad 1 = k_1 < k_2 < \ldots < k_t = n, \quad \text{and} \quad k_{t-1} < n - 1. \\
&\text{For each state } \mathbf{c} = c_0, c_1, \ldots, c_{n-1}, \text{ let } \vec{y}_c := \vec{c}_1, \vec{c}_2, \ldots, \vec{c}_{n-2}, 0 \text{ and } \vec{w}_c := 0, c_1, \ldots, c_{n-2}. \text{ The successor rule }
\end{align*}
\]
\[
Y_1(\mathbf{c}) = \begin{cases} 
\frac{c_0 + c_1 + c_{n-1}}{\bar{c}_0 + c_1 + c_{n-1}} & \text{if } c_{n-1} = 1 \text{ and } \vec{y}_c \text{ is a co-necklace,} \\
\frac{c_0 + c_1 + c_{n-1}}{c_0 + c_1 + c_{n-1}} & \text{if } c_{n-1} = 0 \text{ and there is an } i \text{ such that } \\
k_i \leq \omega(\vec{w}_c) < k_{i+1} \text{ and } \Theta^k \vec{w}_c \text{ is a necklace,} \\
\frac{c_0 + c_1 + c_{n-1}}{c_0 + c_1 + c_{n-1}} & \text{otherwise,} 
\end{cases}
\tag{18}
\]
generates \( 2^{n-3} \) inequivalent de Bruijn sequences of order \( n \) by taking all possible parameters.

Theorem 12. For each \( \mathbf{c} = c_0, c_1, \ldots, c_{n-1} \), let \( \vec{y}_c := \vec{c}_1, \vec{c}_2, \ldots, \vec{c}_{n-2}, 0 \) and \( \vec{w}_c := 0, c_1, \ldots, c_{n-2} \). Let \( k \) be a nonnegative integer. The successor rule
\[
Y_2(\mathbf{c}) = \begin{cases} 
\frac{c_0 + c_1 + c_{n-1}}{\bar{c}_0 + c_1 + c_{n-1}} & \text{if } c_{n-1} = 1 \text{ and } \vec{y}_c \text{ is a co-necklace,} \\
\frac{c_0 + c_1 + c_{n-1}}{c_0 + c_1 + c_{n-1}} & \text{if } c_{n-1} = 0 \text{ and } \Theta^k \vec{w}_c \text{ is a necklace,} \\
\frac{c_0 + c_1 + c_{n-1}}{c_0 + c_1 + c_{n-1}} & \text{otherwise,} 
\end{cases}
\tag{19}
\]
generates \( \text{lcm}(1, 2, \ldots, n - 2) \) inequivalent de Bruijn sequences of order \( n \).

Example 3. Let us consider successor rules in the class \( \Upsilon \) for \( n = 6 \). Using Theorem \[17\] we obtain 8 distinct successor rules, resulting in 8 distinct de Bruijn sequences. Theorem \[12\] gives us 12 distinct de Bruijn sequences. In Table \[2\] the sequences in entries 1, 2, 3, and 8 on Theorem \[17\] are the same as the sequences in entries 2, 3, 4, and 1 on Theorem \[12\] respectively. Table \[2\] contains 16 distinct de Bruijn sequences in total.

There are a number of alternatives to determine a unique state in a PCR cycle \( \mathcal{P} \) whose first bit is 0 that will result in valid new successor rules for de Bruijn sequences. We omit the details here since Theorems \[11\] and \[12\] have already highlighted some of these possibilities.

5. Complexity

We end by considering the complexity of the successor rules constructed in this paper. It is clear that the space complexity is \( O(n) \). In the cycle structure of the PRR of order \( n \), checking the cycle representative of a CCR cycle \( \mathcal{C} \) is equivalent to checking the co-necklace in a cycle generated by the CCR of order \( n - 1 \). To pinpoint a unique state in a PCR cycle \( \mathcal{P} \) is equivalent to checking whether a state is a necklace in a cycle generated by the PCR of order \( n - 1 \) after simple left shifts. The latter can be done in \( O(n) \) time as was established in \[14\]. Thus, each successor rule in the two classes \( \Lambda \) and \( \Upsilon \) requires time and space complexities \( O(n) \) to generate the next bit of a de Bruijn sequence of order \( n \) from a given \( n \)-stage state.

Based on the special LFSR whose characteristic polynomial is \( f_{\text{PRR}}(x) = x^n + x^{n-1} + x + 1 = (x^n + 1)(x + 1) \) for \( n \geq 3 \), we have come up with two generic classes of successor rules. Each class contains numerous distinct successor rules, yielding mostly pairwise inequivalent de Bruijn sequences. The resulting family is of size \( O(2^{n-3}) \). The time and space complexities to generate the next bit in each of the instances are both \( O(n) \). In the appendix we supply a basic
Table 2: Inequivalent de Bruijn sequences constructed based on Theorems 11 and 12 with $n = 6$.

| Entry | $\{k_1, k_2, \ldots, k_t\}$ | The resulting de Bruijn sequence based on Theorem 11 |
|-------|-----------------|--------------------------------------------------|
| 1     | $\{1, 6\}$     | (000000111111010001000101100110101001001100110011111) |
| 2     | $\{1, 2, 6\}$  | (0000001111110101000101110101101010010011001100110011001) |
| 3     | $\{1, 3, 6\}$  | (0000001111110101000101011101101010010011001100110011001) |
| 4     | $\{1, 4, 6\}$  | (0000001111110100111010111010110010001011001100110011001) |
| 5     | $\{1, 2, 3, 6\}$ | (0000001111110101011101010100010110011001100110011001) |
| 6     | $\{1, 2, 4, 6\}$ | (000000111111010010100100101100110011001100110011001) |
| 7     | $\{1, 3, 4, 6\}$ | (00000011111101000101000101100110011001100110011001) |
| 8     | $\{1, 2, 3, 4, 6\}$ | (0000001111110101011001010100010110011001100110011001) |

| Entry | $k$ | The resulting de Bruijn sequence based on Theorem 12 |
|-------|-----|--------------------------------------------------|
| 1     | 0   | (0000001111110101010010011001101000001011001100110011001) |
| 2     | 1   | (000000111111010010000101100110011001100110011001) |
| 3     | 2   | (000000111111010001000101100110011001100110011001) |
| 4     | 3   | (000000111111010011101101110101010001011001100110011001) |
| 5     | 4   | (000000111111010010100100010110011001100110011001) |
| 6     | 5   | (000000111111011001000101100110011001100110011001) |
| 7     | 6   | (0000001111110101110100100010110011001100110011001) |
| 8     | 7   | (00000011111101000010110111011010101000101100110011001) |
| 9     | 8   | (0000001111110101110100100010110011001100110011001) |
| 10    | 9   | (000000111111010101100101010001011001100110011001) |
| 11    | 10  | (000000111111010101100101010001011001100110011001) |
| 12    | 11  | (0000001111110100010100010110011001100110011001) |

implementation in C that produces all of the de Bruijn sequences based on Theorems 7, 9, 11, and 12.

The route that we propose here can be particularly useful to analyse the suitability of an arbitrary FSR whose cycles have small periods. Identifying more classes of suitable FSRs that efficiently produce larger families of de Bruijn sequences via successor rules is an interesting direction to investigate. Adding specific desirable properties for the resulting sequences could be an intriguing challenge to explore.

Acknowledgements

We thank the reviewers for making us aware of an MSc thesis [22], a recent preprint [23], and a webpage [24]. We also thank their helpful suggestions to clarify our contribution and improve the presentation of this paper.

The work of Z. Chang is supported by the National Natural Science Foundation of China under Grant 61772476. Nanyang Technological University Grant Number M4080456 supports the work of M. F. Ezerman. The research of Q. Wang is partially support by the Natural Sciences and Engineering Research Council of Canada (RGPIN-2017-06410).

References

[1] T. van Aardenne-Ehrenfest and N. G. de Bruijn, “Circuits and trees in oriented linear graphs,” Simon Stevin, vol. 28, pp. 203–217, 1951.
Appendix: Source Code

A basic implementation in C is supplied for the interested readers.
This is a basic implementation of the manuscript
An Efficiently Generated Family of Binary de Bruijn Sequences
in submission to Discrete Mathematics
This v1.1 is dated October 1, 2020
*/

#include <stdio.h>
#include <stdlib.h>
#include <math.h>

int *ki_list,t,**all_ki , ki_case =0, lcmnum =0;

void lcm (int ar[], int size){
    int i =0;
    lcmnum = ar[0];
    while (1) {
        for (i = 0; i < size ; i++) {
            if(lcmnum % ar[i]) break;
        }
        if( i == size ) break;
        lcmnum ++;
    }
}

int hamming_weight(int sequence[], int n) {
    int weight = 0;
    for (int j=0; j<n; j++){
        if(sequence[j] == 1) weight ++;
    }
    return weight;
}

int if_necklace(int sequence[], int size){
    int p = 1;
    for (int j = 0; j<size; j++){
        if(sequence[j] < sequence[j-p]) return 0;
        if(sequence[j] > sequence[j-p]) p = j+1;
    }
    if (size % p != 0) return 0;
    return 1;
}

int shift(int sequence[], int size, int ki, int shift_case){
    int states[3*size], state[size];
    int shift_order = 0, position = size;
    for (int j=0; j<size; j++){
        states[j] = sequence[j];
        states[j+size] = sequence[j];
        states[j+2*size] = sequence[j];
    }
    while(shift_order < ki){
        if(states[++position] ==1) shift_order ++;
    }
    if (shift_case == 1){
        for (int j=0; j<size; j++){
            state[j] = states[j+position+1-size];
        }
    }
}
if (if_necklace(state, size)) return 1;
return 0;
}
if (shift_case == 2){
for (int j=0; j<size; j++){
state[j] = !states[j]*position;
}
if (if_necklace(state, size)) return 1;
return 0;
}
}

int alg_theoremSeven(int sequence[], int n, int kicase){
int i, j, u[n-1], u_co[2*(n-1)], number_list[n-1];
if (kicase == pow(2, n-3)-1){
number_list[0] = 1;
number_list[1] = n;
j = 1;
}
else{
for (j=0; j<n-2; j++){
if (all_ki[kicase][j]==0) break;
number_list[j] = all_ki[kicase][j];
}
number_list[j] = n;
}
for (i=0; i<n-1; i++){
u[i] = sequence[i+1];
u_co[i] = sequence[i+1];
u_co[i+n-1] = 1-sequence[i+1];
}
if (sequence[1]==0){
if (if_necklace(u_co, 2*(n-1))) return (sequence[0]+sequence[1]+sequence[n-1]+1) %2;
}
else{
int weight = hamming_weight(u, n-1);
for (i=0; i<n-1; i++){
if (number_list[i] <= weight && weight < number_list[i+1]){
if (shift(u, n-1, number_list[i]-1, 1)) return (sequence[0]+sequence[1]+sequence[n-1]+1) %2;
}
}
return (sequence[0]+sequence[1]+sequence[n-1]) %2;
}

int alg_theoremNine(int sequence[], int n, int k_number){
int i, u[n-1], u_co[2*(n-1)];
for (i=0; i<n-1; i++){
u[i] = sequence[i+1];
u_co[i] = sequence[i+1];
u_co[i+n-1] = 1-sequence[i+1];
}
if (sequence[1]==0){
if (if_necklace(u_co, 2*(n-1))) return (sequence[0]+sequence[1]+sequence[n-1]+1) %2;
}
else{
if (shift(u, n-1, k_number % hamming_weight(u, n-1), 1)) return (sequence[0]+sequence[1]+sequence[n-1]+1)%2;
} return (sequence[0]+sequence[1]+sequence[n-1])%2;

int alg_theoremEleven(int sequence[], int n, int kicase){
    int i, j, u[n-1], u_co[2*(n-1)], w_bar[n-1], number_list[n-1];
    if (kicase == pow(2, n-3)-1){
        number_list[0] = 1;
        number_list[1] = n;
        j = 1;
    } else{
        for(j=0;j<n-2;j++) {
            if (all_ki[kicase][j]==0) break;
            number_list[j] = all_ki[kicase][j];
        }
        number_list[j] = n;
    }
    w_bar[0] = 1;
    u[n-2] = 0, u_co[n-2] = 0, u_co[2*n-3] = 1;
    for (i=0;i<n-2;i++){
        u[i] = 1-sequence[i+1];
        u_co[i] = 1-sequence[i+1];
        u_co[i+n-1] = sequence[i+1];
        w_bar[i+1] = 1-sequence[i+1];
    }
    if (sequence[n-1]==1){
        if (if_necklace(u_co, 2*(n-1))) return (sequence[0]+sequence[1]+sequence[n-1]+1)%2;
    } else{
        int zero_number = hamming_weight(w_bar, n-1);
        for (i=0;i<n-1;i++){
            if (number_list[i] <= zero_number && zero_number < number_list[i+1]){
                if (shift(w_bar, n-1, number_list[i], 2)) return (sequence[0]+sequence[1]+sequence[n-1]+1)%2;
            }
        }
        return (sequence[0]+sequence[1]+sequence[n-1])%2;
    }

int alg_theoremTwelve(int sequence[], int n, int k_number){
    int i, u[n-1], u_co[2*(n-1)], w_bar[n-1];
    w_bar[0] = 1;
    u[n-2] = 0, u_co[n-2] = 0, u_co[2*n-3] = 1;
    for (i=0;i<n-2;i++){
        u[i] = 1-sequence[i+1];
        u_co[i] = 1-sequence[i+1];
        u_co[i+n-1] = sequence[i+1];
        w_bar[i+1] = 1-sequence[i+1];
    }
    if (sequence[n-1]==1){
        if (if_necklace(u_co, 2*(n-1))) return (sequence[0]+sequence[1]+sequence[n-1]+1)%2;
    } else{
if (shift(w_bar, n-1, k_number % hamming_weight(w_bar, n-1), 2)) return
(sequence[0]+sequence[1]+sequence[n-1]+1)%2;
}
return (sequence[0]+sequence[1]+sequence[n-1])%2;
}

void comb(int m, int k){
int i, j;
for(i=m; i>=k; i--){
ki_list[k-1] = i;
if(k>1) comb(i-1, k-1);
else{
all_ki[ki_case][0] = 1;
for(j=0; j<t; j++) all_ki[ki_case][j+1] = ki_list[j]+1;
ki_case++;
}
}
}

void DB(int alg_number, int n) {
int i, j, new_bit, kicase, k_number, state[n];
if (alg_number == 1 || alg_number == 3){
for (kicase=0; kicase<pow(2, n-3); kicase++){
for (i=0; i<n; i++) state[i] = 0;
for (j=0; j<n-2; j++){
if (all_ki[kicase][j]==0) break;
printf("%d", all_ki[kicase][j]);
}
printf(" ");
do{
printf("%d", state[0]);
switch(alg_number) {
case 1: new_bit = alg_theoremSeven(state, n, kicase); break;
case 3: new_bit = alg_theoremEleven(state, n, kicase); break;
default: break;
}
for (i=0; i<n; i++) state[i] = state[i+1];
state[n-1] = new_bit;
} while (hamming_weight(state, n)>0);
printf("\n");
}
else{
for (k_number=1; k_number<=lcmnum; k_number++){
for (i=0; i<n; i++) state[i] = 0;
printf("%d", k_number);
printf(" ");
do{
printf("%d", state[0]);
switch(alg_number) {
case 2: new_bit = alg_theoremNine(state, n, k_number); break;
case 4: new_bit = alg_theoremTwelve(state, n, k_number); break;
default: break;
}
for (i=0; i<n; i++) state[i] = state[i+1];
state[n-1] = new_bit;
} while (hamming_weight(state, n)>0);
printf("\n");
}
void main() {
    int n, k, alg_number, i;
    printf("Enter n: "); scanf("%d", &n);
    int max_number = pow(2, n-3);
    ki_list = (int*) calloc((n-3), sizeof(int));
    all_ki = (int**) calloc(max_number, sizeof(int*));
    for(i=0; i<max_number; i++) {
        all_ki[i] = (int*) calloc(n-2, sizeof(int));
    }
    for(i=n-3; i>0; i--){
        t = 1;
        comb(n-3,i);
        all_ki[ki_case][0] = 1;
        int klist[n-2];
        for (i=0; i<n-2; i++){
            klist[i] = i+1;
        }
        lcm(klist, n-2);
        for (alg_number=1; alg_number<5; alg_number++){
            DB(alg_number, n);
            printf("\n");
        }
        for (i=0; i<max_number; i++) free(all_ki[i]);
        free(all_ki), free(ki_list);
    }
}