**Theoretical Informatics and Applications** Will be set by the publisher
Informatique Théorique et Applications

# LAS VEGAS ALGORITHMS TO GENERATE UNIVERSAL CYCLES AND DE BRUIJN SEQUENCES UNIFORMLY AT RANDOM

JOE SAWADA AND DANIEL GABRIĆ

**Abstract**. We present practical algorithms for generating universal cycles uniformly at random. In particular, we consider universal cycles for shorthand permutations, subsets and multiset permutations, weak orders, and orientable sequences. Additionally, we consider de Bruijn sequences, weight-range de Bruin sequences, and de Bruijn sequences, with forbidden $0^z$ substring. Each algorithm, seeded with a random element from the given set, applies a random walk of an underlying Eulerian de Bruijn graph to obtain a random arborescence (spanning in-tree). Given the random arborescence and the de Bruijn graph, a corresponding random universal cycle can be generated in constant time per symbol. We present experimental results on the average cover time needed to compute a random arborescence for each object using a Las Vegas algorithm.

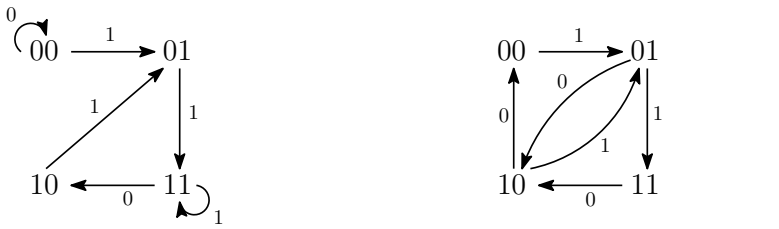
**AMS Subject Classification.** — Give AMS classification codes —.

## 1. INTRODUCTION

Let $\mathbf{\Sigma}_k(n)$ denote the set of all strings of length $n$ over the alphabet $\{0, 1, \ldots, k-1\}$. Let $\mathbf{S}$ denote a subset of $\mathbf{\Sigma}_k(n)$. The *de Bruijn graph* of $\mathbf{S}$, denoted $G(\mathbf{S})$, is the directed graph where each vertex corresponds to a length-$(n-1)$ prefix or a length-$(n-1)$ suffix of a string in $\mathbf{S}$; for each string $u_1u_2\cdots u_n$ in $\mathbf{S}$ there is a directed edge labeled $u_n$ from vertex $u = u_1u_2\cdots u_{n-1}$ to vertex $v = u_2u_2\cdots u_n$. For example, see Figure 1.

A *universal cycle* for $\mathbf{S}$, is a cyclic string of length $|\mathbf{S}|$ that contains each string in $\mathbf{S}$ as a substring exactly once (including the wraparound); they exist if and only if $G(\mathbf{S})$ is Eulerian, that is, $G(\mathbf{S})$ contains an Euler cycle. For example, consider $\mathbf{S} = \{001, 010, 101, 011, 110, 100\}$ and the de Bruijn graph $G(\mathbf{S})$ illustrated in Figure 1(b): the Euler cycle $00, 01, 11, 10, 01, 10, 00$ corresponds to the universal cycle $110100$ obtained by outputting the labels on the edges in the Euler cycle. In this paper we are

*Keywords and phrases:* Las Vegas algorithm, universal cycle, de Bruijn sequence, weak order, subsets, permutations, orientable sequence




(a) $G(\{001, 000, 011, 111, 110, 101\})$ (b) $G(\{001, 010, 101, 011, 110, 100\})$

FIGURE 1. Two de Bruijn graphs. The graph in (b) is Eulerian.

concerned with interesting subsets $\mathbf{S}$ whose underlying de Bruijn graph is Eulerian, i.e., $\mathbf{S}$ admits a universal cycle. In particular, we consider:

(1) (shorthand) permutations, subsets, and permutations of a multiset,
(2) weak orders,
(3) $k$-ary strings of length $n$ (which yield de Bruijn sequences),
(4) $k$-ary strings of length $n$ that do not contain $0^z$ (cyclically),
(5) $k$-ary strings of length $n$ with weight in the range $[a, b]$, and
(6) $k$-ary strings of length $n$ that produce asymptotically optimal orientable sequences,

where the *weight* of a string is the sum of its symbols, and $[a, b]$ denotes the set of integers $\{a, a+1, \ldots, b\}$.

The primary objective of this paper is to describe practical algorithms to generate universal cycles for these objects *uniformly at random*, while using exponential space to store an underlying de Bruijn graph. We recall a generic Las Vegas algorithm from [16] that generates a random Euler cycle in any directed Eulerian graph by first generating a random arborescence (spanning in-tree). For each set $\mathbf{S}$, we generate a random string in $\mathbf{S}$ in order to seed the algorithm by selecting a root for the arborescence. The generic algorithm is presented in Section 3. Then, in Section 4 we consider each of the aforementioned sets $\mathbf{S}$ and provide (i) a discussion of how to generate a random element from $\mathbf{S}$ and (ii) experimental evidence for the average cover time to compute a random arborescence in $G(\mathbf{S})$.

**Motivation.** This paper is motivated by a recent result from Lipták and Parmigiani [18] that generates random de Bruijn sequences, although not uniformly at random (see Section 4.3). In that paper, they compared their approach with an implementation of Fleury's algorithm [11] to generate Euler cycles, modified by adding randomization. That implementation could not generate all possible de Bruijn sequences, however, it served "as the closest available method for comparison" [18]. Indeed, despite the vast literature on universal cycle constructions, and in particular, de Bruijn sequences, we also found no detailed discussion or resource regarding the generation of these sequences uniformly at random, other than a passing comment by Propp and Wilson in [19, p.172]. However, it is well known that a random arborescence in a directed Eulerian graph can be used to generate a random Euler cycle [16].

## 2. PRELIMINARIES

Let $G = (V, E)$ denote a directed graph consisting of a non-empty set of vertices $V$ and an edge set $E$ consisting of ordered pairs of elements in $V$. A *walk* in $G$ is a sequence of vertices $v_1, v_2, \ldots, v_j$ such that $(v_i, v_{i+1}) \in E$ for all $i$ in $\{1, 2, \ldots, j-1\}$. A *reverse walk* is a sequence of vertices $v_1, v_2, \ldots, v_j$ such that $(v_{i+1}, v_i) \in E$ for all $i$ in $\{1, 2, \ldots, j-1\}$. Let $G$ be represented by a standard adjacency list representation. We define a *traversal* to be a walk starting from some vertex $v$ that follows the rule: at each vertex $u$ , the next vertex corresponds to the first unused edge on $u$'s adjacency list; the traversal terminates when it reaches a vertex whose adjacency list has been exhausted.

**Example 1** Consider the directed graph $G = (V, E)$ where $V = \{u, v, w\}$ and $E = \{(u, u), (u, v), (v, w), (w, u)\}$. Then $u, v, w$ is a walk in $G$, and $w, v, u$ is a reverse walk in $G$. Give the adjacency list representation $u \to u, v$, $v \to w$, and $w \to u$, the traversal starting at $u$ is the walk $u, u, v, w, u$ using all four edges. It corresponds to an Euler cycle in $G$.

A traversal that starts and ends with the same vertex, say $r$, and visits all the edges in $E$ generates an Euler cycle, i.e., the traversal does not "burn bridges" [11]. This means that the $|V| - 1$ edges corresponding to the last edges on each adjacency list, except for $r$'s, form an arborescence (spanning in-tree) rooted at $r$. Using this well-known fact, all Euler cycles can be generated as follows:

**Generate all Euler Cycles for a directed graph** $G$

(1) Generate all arborescences $\mathcal{T}$ for each possible root $r \in V$.
(2) For each $\mathcal{T}$ generated in step (1), take each edge $(u, v)$ in $\mathcal{T}$, and set the vertex $v$ to be the last on $u$'s adjacency list; then generate all possible orderings for the remaining vertices on each adjacency list.
(3) For each set of adjacency lists generated in step (2), generate a traversal of $G$ starting at the corresponding root $r$.

It is important to note that the above algorithm will generate all Euler cycles in $G$ exactly once, where the starting edge in each cycle is important. By fixing a single root $r$ at step (1), we generate all Euler cycles up to equivalence when the edges are considered to be a circular list of edges; the starting edge in the cycle is immaterial. However, note that the same equivalent cycle could be generated twice. For instance, consider the graph in Example 1 with root vertex $u$. The algorithm produces two sets of edge listings at step (2): one where $u \to u, v$, and one where $u \to v, u$. The two listings produce equivalent Euler cycles, namely $u, u, v, w, u$ and $u, v, w, u, u$.

## 3. RANDOM GENERATION

In this section the algorithm outlined in Section 2 to generate all possible Euler cycles in a directed graph is applied to generate a single universal cycle *uniformly at random* for a set $\mathbf{S}$ with an underlying Eulerian de Bruijn graph $G(\mathbf{S})$. The Las Vegas algorithm

describe by Algorithm R below summarizes the approach from Kandel et al. [16], which extends the work from [4]. We note one difference in our presentation. The algorithm in [16], takes as input a (cyclic) sequence $\mathcal{S}$ that may contain duplicate length $n$ substrings. From this sequence, it constructs a de Bruijn graph that allows for multiple edges between vertices. The algorithm is then initialized by selecting a random rotation of $\mathcal{S}$, which effectively generates a random edge in the underlying graph. For our purposes, we do not have an initial sequence $\mathcal{S}$. Instead, for each set $\mathbf{S}$ considered in Section 4, we present an efficient algorithm to compute a random element in $\mathbf{S}$, which corresponds to a random edge in $G(\mathbf{S})$.

**Algorithm R**

Generate a universal cycle for set $\mathbf{S}$ uniformly at random given the underlying (Eulerian) de Bruijn graph $G(\mathbf{S})$:

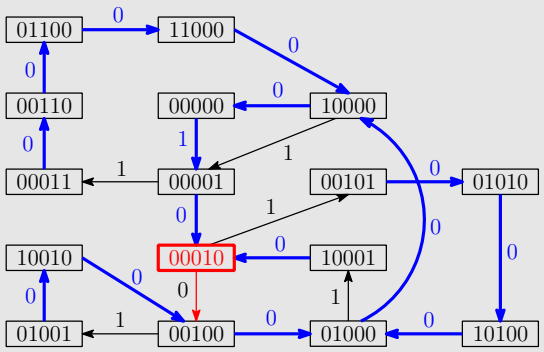
(1) Generate a random edge $(r, v)$ in $G(\mathbf{S})$, i.e., a string in $\mathbf{S}$, to obtain a random root vertex $r$
(2) Generate a random arborescence $T$ directed to root $r$
(3) Make each edge of $T$ (the bridges) the last edge on the adjacency list of the corresponding outgoing vertex (the root does not have such a bridge), then randomly assign the order of the remaining outgoing edges
(4) Starting from $r$, perform a traversal of $G(\mathbf{S})$, outputting the label on each edge as it is visited.

Selecting a random vertex instead of an edge at step (1) of Algorithm R will not lead to Euler cycle generated *uniformly* at random if $G(\mathbf{S})$ is not regular, as illustrated in the following example.

**Example 2** Consider $\mathbf{S} = \{001, 010, 101, 011, 110, 100\}$ and its corresponding de Bruijn graph $G(\mathbf{S})$ illustrated in Figure 1(b). Every vertex in $G(\mathbf{S})$ roots two unique spanning arborescences. For each tree rooted at $r = 000$ there is one corresponding edge labeling; however, each tree rooted at $r = 010$ has two edge labelings since the there are two possible adjacency lists for the root. Thus, when $r = 000$, there are two possible universal cycles, while if $r = 010$ there are 4 possible universal cycles that could get generated. Thus, if a random vertex instead of a random edge is chosen at step (1) in Algorithm R, a universal cycle generated from $r = 000$ will occur twice as frequently as a universal cycle rooted at $r = 010$.

The following example highlights the steps from Algorithm R.

**Example 3** Consider the set $\mathbf{S}$ consisting of all binary strings of length $n = 6$ with weight in the range [1,2]. The de Bruijn graph $G(\mathbf{S})$ is shown below with the randomly selected edge 000100 from step (1) of Algorithm R.




The graph illustrates a random arborescence $T$ rooted at $r = 00010$. Based on $T$, the ordering of the adjacency lists for the vertices following step (3) in Algorithm R is as follows, where the adjacency list of the root 00010 is randomly selected as 00100,00101:

| | | | | | |
|---|---|---|---|---|---|
| 00000 | $\rightarrow$ | 00001 | 00110 | $\rightarrow$ | 01100 |
| 00001 | $\rightarrow$ | 00011, 00010 | 01001 | $\rightarrow$ | 10010 |
| 00010 | $\rightarrow$ | 00100, 00101 | 01010 | $\rightarrow$ | 10100 |
| 00100 | $\rightarrow$ | 01001, 01000 | 01100 | $\rightarrow$ | 11000 |
| 01000 | $\rightarrow$ | 10001, 10000 | 10001 | $\rightarrow$ | 00010 |
| 10000 | $\rightarrow$ | 00001, 00000 | 10010 | $\rightarrow$ | 00100 |
| 00011 | $\rightarrow$ | 00110 | 10100 | $\rightarrow$ | 01000 |
| 00101 | $\rightarrow$ | 01010 | 11000 | $\rightarrow$ | 10000. |

Step (4) from Algorithm R produces the following random universal cycle for $\mathbf{S}$:

01000101000011000010.

For Eulerian graphs, an arborescence can be generated uniformly at random by performing a random backwards walk until every vertex is visited. The first time a vertex is visited, the edge is recorded as a *tree edge* in the random arborescence [16]. The running time of this step depends on the *cover time* of the random walk, which is the number of steps it takes to visit every vertex. The algorithm requires $\Theta(|\mathbf{S}|)$ space to store the graph. The final step (4) of Algorithm R generates a random universal cycle for $\mathbf{S}$ in constant time per symbol.

Theorem 3 in [16] shows that the expected cover time of a directed Eulerian graph is at most $|V|^2|E|$. In the next section, however, our experiments seem to indicate that some de Bruijn graphs appear to have a significantly faster expected cover time. We do not have a proof of this observation, nor have we found a proof in the literature.

There are algorithms to generate random arborescences that are not dependent on the expected cover time. Wilson [26] shows how to produce a random arborescence in time proportional to the maximum hitting time. The maximum hitting time is the maximum over all pairs of vertices $u$ and $v$ of the expected number of steps for a random walk to travel from $u$ to $v$. Propp and Wilson [19] show how to produce a random arborescence in time proportional to the smaller of the maximum hitting time and the mean hitting time.

The mean hitting time is the average over all pairs of vertices $u$ and $v$ of the expected number of steps for a random walk to travel from $u$ to $v$.

## 4. APPLICATIONS AND EXPERIMENTAL RESULTS

In this section we apply Algorithm R to generate universal cycles uniformly at random for shorthand permutations, subsets and multiset permutations, $k$-ary strings (de Bruijn sequences), generalizations of de Bruijn sequences including those with no $0^z$ substring and those with bounded weight, and orientable sequences.

For each object, we discuss how to generate a random edge in the underlying de Bruijn graph to seed the algorithm, and present experimental evidence for the cover time required to compute a random arborescence. Implementations of our algorithms are available at `http://debruijnsequence.org/db/random`. In our implementations to compute the random arborescences, we did not pre-compute the shift-graphs, but instead used a mapping of each vertex to an integer (using a ranking algorithm or similar) to store whether a vertex had been visited. We applied a similar strategy to generate the random universal cycle in step (4) of Algorithm R.

The results presented in this section with respect to the cover times of certain de Bruijn graphs are experimental. We leave it as in interesting open problem to determine a tight upper bound on the expected cover time for the de Bruijn graphs being considered.

### 4.1. PERMUTATIONS, SUBSETS, AND MULTISET PERMUTATIONS

Universal cycles do not exist, in general, for permutations and subsets. For permutations, however, observe that the final symbol is redundant. If $p_1p_2\cdots p_n$ is a permutation, we say that $p_1p_2\cdots p_{n-1}$ is a *shorthand permutation* of order $n$, where the last symbol is implied. Let $\mathbf{SP}(n)$ be the set of all shorthand permutations of order $n$. Similarly, if $b_1b_2\cdots b_n$ is a binary string with $k$ ones (representing a $k$-subset of an $n$-set), we say that $b_1b_2\cdots b_{n-1}$ is a *shorthand $k$-subset* of order $n$, where the last bit is implied. Let $\mathbf{S}(n,k)$ be the set of all shorthand $k$-subsets of order $n$. Multiset permutations (strings with fixed content) generalize both permutations and subsets. If $m_1m_2\cdots m_n$ is a permutation of the multiset $\{s_1, s_2, \ldots, s_n\}$, then we say $m_1m_2\cdots m_{n-1}$ is a *shorthand multiset permutation*. When each $s_i = i$, a multiset permutation is simply a permutation, and when the multiset contains $k$ ones and $(n-k)$ zeros, it represents a binary string with weight $k$ representing a $k$-subset of an $n$-set.

For shorthand permutations, the underlying de Bruijn graph has $n!$ edges. Each vertex has in-degree = out-degree = 2 and thus there are $n!/2$ vertices. For shorthand $k$-subsets, where $k \geq 2$, the underlying de Bruijn graph has $\binom{n}{k}$ edges and each vertex is a binary string of length $n-2$ with weight $k-2$, $k-1$, or $k$; there are $\binom{n-2}{k-2} + \binom{n-2}{k-1} + \binom{n-2}{k}$ vertices. Note that each vertex has the same in-degree as out-degree; however, this value may be either 1 or 2. For example, if $n = 5$ and $k = 2$, the vertex 001 has two incoming edges from 000 and 100, and outgoing edges to 010 and 011, while the vertex 000 has one incoming edge from 100 and one outgoing edge to 001.

It is well known that a random permutation can be generated by applying the Fisher-Yates shuffle [10]; an $O(n)$ time implementation is provided by Knuth [17, Algorithm P] based on a presentation by Durstenfeld [9]. As noted by Arndt [5], it is straightforward to apply the shuffle to generate an unbiased random multiset permutation $m_1 m_2 \cdots m_n$ in $\mathcal{O}(n)$ time as illustrated in Algorithm 1. Thus, $m_1 m_2 \cdots m_{n-1}$ is a random shorthand multiset permutation that can be used to seed Algorithm R for shorthand permutations, shorthand subsets, and more generally, shorthand multiset permutations.

**Algorithm 1** Random generation of a permutation $m_1 m_2 \cdots m_n$ of the multiset $\{s_1, s_2, \ldots, s_n\}$ applying the Fisher-Yates shuffle

$m_1 m_2 \cdots m_n \leftarrow s_1 s_2 \cdots s_n$
**for** $i$ **from** $n$ **down to** $2$ **do**
  $j \leftarrow$ random integer in $[1, i]$
  SWAP$(m_i, m_j)$

Table 1 shows the minimum, maximum, and average ratios of the cover time to total edges in $G(\mathbf{SP}(n))$ and $G(\mathbf{S}(n, n/2))$ by running Algorithm R for 10,000 iterations.

| | **Ratio**: cover time / $n!$ | | |
|---|---|---|---|
| $n$ | **Min** | **Max** | **Avg** |
| 3 | 0.3 | 0.3 | 0.3 |
| 4 | 0.5 | 3.7 | 1.1 |
| 5 | 0.8 | 5.3 | 2.0 |
| 6 | 1.7 | 6.8 | 3.0 |
| 7 | 2.7 | 6.5 | 3.9 |
| 8 | 3.9 | 8.0 | 4.9 |
| 9 | 5.0 | 7.7 | 5.9 |
| 10 | 6.4 | 8.1 | 7.0 |

| | **Ratio**: cover time / $\binom{n}{k}$ | | |
|---|---|---|---|
| $n$ | **Min** | **Max** | **Avg** |
| 10 | 1.8 | 18.9 | 5.0 |
| 12 | 3.2 | 18.0 | 6.3 |
| 14 | 4.6 | 16.4 | 7.6 |
| 16 | 5.9 | 18.8 | 8.8 |
| 18 | 7.2 | 20.3 | 10.1 |
| 20 | 8.5 | 21.8 | 11.4 |
| 22 | 9.7 | 21.8 | 12.7 |
| 24 | 11.2 | 22.6 | 14.1 |
| 26 | 12.4 | 24.0 | 15.4 |

TABLE 1. The minimum, maximum, and average ratios of the cover time to total edges in the de Bruijn graphs $G(\mathbf{SP}(n))$ (left) and $G(\mathbf{S}(n, n/2))$ (right) by running Algorithm R for 10,000 iterations.

Universal cycles for shorthand permutations can be constructed in $\mathcal{O}(1)$ amortized time per symbol using $\mathcal{O}(n)$ space [15, 21]. An $\mathcal{O}(n)$-time successor rule is presented in [14], and an $\mathcal{O}(1)$ amortized time per symbol algorithm applying concatenation trees is presented in [22] that uses $\mathcal{O}(n^2)$ space. Universal cycles for shorthand subsets can be constructed in $\mathcal{O}(1)$ amortized time per symbol using $\mathcal{O}(n)$ space [20]. Universal cycles for shorthand multiset permutations (strings with fixed content) can be constructed in $\mathcal{O}(1)$ amortized time per symbol using $\mathcal{O}(n)$ space [23].

### 4.2. WEAK ORDERS

A *weak order* is the number of ways $n$ competitors can finish in a race if ties are allowed. Let $\mathbf{W}(n)$ denote the set of weak orders with $n$ competitors, and let $W_n$ denote

$|\mathbf{W}(n)|$. For example,

$$\mathbf{W}(3) = \{111, 113, 131, 311, 122, 212, 221, 123, 132, 213, 231, 312, 321\},$$

and $W_3 = 13$. The number of weak orders where there is a $k$-way tie for first is given by $\binom{n}{k}W_{n-k}$ for $k < n$; there is 1 weak order when $k = n$. Thus, $W_n = \sum_{k=1}^{n} \binom{n}{k} W_{n-k}$.

To generate a random weak order, we will apply this recurrence to group the strings of $\mathbf{W}(n)$ based on their content. Let $c_i$ denote the number of occurrences of the symbol $i$ in $\omega = w_1 w_2 \cdots w_n$. Since every weak order contains 1 as its smallest symbol and the largest symbol possible is $n$, we say that $(c_1, c_2, \ldots, c_n)$ is the *content* (also known as the Parikh vector) of $\omega$. For example, if $n = 3$ then each weak order has content $(1, 1, 1), (1, 2, 0), (2, 0, 1)$, or $(3, 0, 0)$. If we partition $\mathbf{W}(n)$ into subsets based on their content, we can order the subsets based on the lexicographic ordering of the corresponding content. The weak orders with exactly one 1 comes first, followed by those with exactly two 1s and so on. Thus, to generate a random weak ordering in $\mathbf{W}(n)$, we can (i) select a random integer $r$ in $[1, W_n]$, (ii) apply the recurrence to determine the content $(c_1, c_2, \ldots, c_n)$ based on the described ordering, and (iii) apply a Fisher-Yates shuffle to obtain a random multiset permutation (weak order) with content $(c_1, c_2, \ldots, c_n)$ (see Section 4.1). Details are provided in Algorithm 2. Note that the integer $r$ does not uniquely determine the weak order being generated, however, it is possible to obtain this property by applying an unranking algorithm for multiset permutations instead of applying the shuffle.

**Algorithm 2** Random generation of a weak order $\omega$ from $\mathbf{W}(n)$.

```
(c_1, c_2, ..., c_n) ← (0, 0, ... 0)
t ← n
v ← 1
r ← random integer in [1, W_n]
while t ≥ 1 do
    ▷ Determine c_v applying the recurrence for W_t
    for j from 1 to t do
        p_j ← C(t, j) W_{t-j}
        if r ≤ p_j then break
        r ← r − p_j
    c_v ← j
    v ← v + j
    t ← t − j

▷ Apply Fisher-Yates shuffle to generate a random multiset permutation
ω ← a random multiset permutation with content (c_1, c_2, ..., c_n)
```

**Lemma 4.1.** *Algorithm 2 can generate a weak order $w_1 w_2 \cdots w_n$ uniformly at random using $\mathcal{O}(n^2)$ simple operations on numbers up to $W_n$.*

*Proof.* Consider Algorithm 2. The values $W_j$, for $1 \leq j \leq n$ can be precomputed via dynamic programming using $\mathcal{O}(n^2)$ simple operations on numbers up to $W_n$. We

| | **Ratio**: cover time / $W_n$ | | |
|---|---|---|---|
| $n$ | **Min** | **Max** | **Avg** |
| 3 | 0.5 | 9.3 | 1.8 |
| 4 | 1.1 | 12.7 | 3.7 |
| 5 | 2.5 | 12.9 | 5.4 |
| 6 | 4.1 | 16.7 | 7.2 |
| 7 | 6.4 | 21.3 | 9.3 |
| 8 | 8.7 | 18.5 | 11.5 |
| 9 | 12.0 | 15.4 | 13.8 |

TABLE 2. The minimum, maximum, and average ratios of the cover time to total edges in $G(\mathbf{W}(n))$ by running Algorithm R for 10,000 iterations.

can compute the required binomial coefficients of the form $\binom{n}{k}$ in the same time using Pascal's identity. The **while** loop iterates at most $n$ times and each iteration requires at most $\mathcal{O}(n)$ simple operations on numbers up to $W_n$. □

Table 2 shows the minimum, maximum, and average ratios of the cover time to total edges in $G(\mathbf{W}(n))$ by running Algorithm R for 10,000 iterations.

Universal cycles for weak orders can be constructed via a successor rule that generates the sequence in $\mathcal{O}(n)$ time per symbol using $\mathcal{O}(n)$ space [25]. By applying concatenation trees, they can be generated in $\mathcal{O}(1)$ amortized time using $\mathcal{O}(n^2)$ space [22]. See the enumeration sequence A000670 for $W_n$ in the Online Encyclopedia of Integer Sequences [1].

### 4.3. DE BRUIJN SEQUENCES

For de Bruijn sequences, the underlying de Bruijn graph $G(\mathbf{\Sigma}_k(n))$ has $k^{n-1}$ vertices and $k^n$ edges. A random $k$-ary string (edge) can be computed in $\mathcal{O}(n)$ time by generating a random symbol in $[0, k-1]$ $n$ times. Table 3 and Table 4 show the minimum, maximum, and average ratios of the cover time to total edges in $G(\mathbf{\Sigma}_k(n))$, for $k = 2, 3, 4$ by running Algorithm R for 10,000 iterations.

If an application does not require a sequence generated uniformly at random, an algorithm which applies a Burrows-Wheeler transform can be applied; it outputs each de Bruijn sequence with positive probability [18]. The algorithm requires $\Theta(2^n)$ space and produces each symbol in $\mathcal{O}(\alpha(2^n))$ amortized time per symbol for $k = 2$, where $\alpha(n)$ is the inverse Ackerman function. It is important to note that $\alpha(n)$ grows extremely slowly, with $\alpha(n) \leq 5$ for any $n$ of practical value. The first estimate on the mean *discrepancy* of de Bruijn sequences is obtained using this algorithm [18].

In Table 5, we compare the running times of the (non-uniform) C++ algorithm implemented by the authors of [18] and our (uniform) algorithm implemented in C; both implementations are available online for download at [2]. We average the running time

| $n$ | **Ratio**: cover time / $2^n$ **Min** | **Max** | **Avg** |
|---|---|---|---|
| 4 | 0.4 | 8.4 | 1.3 |
| 5 | 0.5 | 7.1 | 1.7 |
| 6 | 0.6 | 8.3 | 2.1 |
| 7 | 0.9 | 8.7 | 2.4 |
| 8 | 1.1 | 8.6 | 2.7 |
| 9 | 1.4 | 9.0 | 3.1 |
| 10 | 1.9 | 9.0 | 3.4 |
| 11 | 2.2 | 9.9 | 3.8 |
| 12 | 2.5 | 10.7 | 4.1 |
| 13 | 2.9 | 9.0 | 4.5 |
| 14 | 3.4 | 10.2 | 4.8 |
| 15 | 3.8 | 10.1 | 5.1 |

| $n$ | **Ratio**: cover time / $2^n$ **Min** | **Max** | **Avg** |
|---|---|---|---|
| 16 | 4.1 | 10.4 | 5.5 |
| 17 | 4.5 | 9.8 | 5.9 |
| 18 | 4.9 | 9.1 | 6.2 |
| 19 | 5.3 | 9.8 | 6.5 |
| 20 | 5.8 | 9.8 | 6.9 |
| 21 | 6.1 | 9.4 | 7.3 |
| 22 | 6.6 | 10.4 | 7.8 |
| 23 | 7.0 | 9.2 | 8.0 |
| 24 | 7.7 | 11.6 | 9.1 |
| 25 | 7.6 | 10.5 | 8.5 |
| 26 | 8.1 | 10.7 | 8.9 |

TABLE 3. The minimum, maximum, and average ratios of the cover time to total edges in the de Bruijn graph $G(\mathbf{\Sigma}_2(n))$ by running Algorithm R for 10,000 iterations.

| $n$ | **Ratio**: cover time / $3^n$ **Min** | **Max** | **Avg** |
|---|---|---|---|
| 3 | 0.3 | 3.9 | 0.9 |
| 4 | 0.5 | 4.5 | 1.3 |
| 5 | 0.7 | 4.6 | 1.7 |
| 6 | 1.0 | 5.6 | 2.0 |
| 7 | 1.5 | 5.1 | 2.4 |
| 8 | 1.8 | 5.6 | 2.8 |
| 9 | 2.2 | 6.3 | 3.1 |
| 10 | 2.6 | 6.0 | 3.5 |
| 11 | 3.0 | 6.1 | 3.9 |
| 12 | 3.4 | 6.3 | 4.3 |
| 13 | 4.0 | 6.4 | 4.6 |
| 14 | 4.3 | 6.3 | 4.9 |
| 15 | 4.6 | 6.6 | 5.3 |
| 16 | 5.0 | 6.7 | 5.7 |

| $n$ | **Ratio**: cover time / $4^n$ **Min** | **Max** | **Avg** |
|---|---|---|---|
| 4 | 0.5 | 4.5 | 1.3 |
| 5 | 0.7 | 4.6 | 1.7 |
| 6 | 1.0 | 5.6 | 2.0 |
| 7 | 1.5 | 5.1 | 2.4 |
| 8 | 1.8 | 5.6 | 2.8 |
| 9 | 2.2 | 6.3 | 3.1 |
| 10 | 2.6 | 6.0 | 3.5 |
| 11 | 3.0 | 6.1 | 3.9 |
| 12 | 3.6 | 4.7 | 4.0 |
| 13 | 3.9 | 5.0 | 4.3 |

TABLE 4. The minimum, maximum, and average ratios of the cover time to total edges in the de Bruijn graph $G(\mathbf{\Sigma}_k(n))$ for $k = 3$ and $k = 4$ by running Algorithm R for 10,000 iterations.

over 10 iterations, not including the time it takes to output the sequence.[1] Interestingly,

[1]Our experiments were run on an iMac desktop with an Apple M4 processor. Our experimental times are faster than those reported in [18], which is likely due to a faster processor.

| | Average clock time in seconds | |
|---|---|---|
| $n$ | **Non-uniform [18]** | **Uniform** |
| 22 | $< 1$ | 3 |
| 23 | $< 1$ | 6 |
| 24 | $< 1$ | 14 |
| 25 | 2 | 31 |
| 26 | 6 | 92 |
| 27 | 13 | 211 |
| 28 | 28 | 509 |
| 29 | 56 | 1094 |
| 30 | 122 | 2287 |

TABLE 5. Comparing the average clock time in seconds over 10 iterations between the algorithm from [18] which generates a de Bruijn sequence non-uniformly at random, and Algorithm R which generates a de Bruijn sequence uniformly at random.

using the standard `rand()` function in the latter implementation resulted in a large cycle of generated bits, which did not allow all vertices to be visited during the "random" walk when $n \geq 28$. Thus, an alternate method for generating random bits had to be deployed. [2] If running time is a concern, the non-uniform result from [18] may be preferred for such applications. However, if a sequence is desired to be generated *uniformly at random*, the one presented in this paper should be considered. It is also important to note that the algorithm described in this paper is a "Las Vegas" algorithm which means that no upper bound on the running time can be given, as it is not guaranteed to terminate.

### 4.4. WEIGHT-RANGE DE BRUIJN SEQUENCES

Let $\mathbf{WR}_k(n, [\ell, u])$ denote the subset of strings in $\mathbf{\Sigma}_k(n)$ with weight in the range $[\ell, u]$. Let $WR_k(n, [\ell, u])$ denote $|\mathbf{WR}_k(n, [\ell, u])|$. It is straightforward to observe that $WR_k(n, [\ell, u]) = 0$ if $u < 0$ or $\ell > n(k-1)$; otherwise, if $n = 1$ then $WR_k(n, [\ell, u]) = min(u, k-1) - max(\ell, 0) + 1$, and if $n > 1$:

$$WR_k(n, [\ell, u]) = \sum_{j=0}^{k-1} WR_k(n-1, [\ell-j, u-j]).$$

A *weight-range de Bruijn sequence* is a universal cycle for the set $\mathbf{WR}_k(n, [\ell, u])$. The de Bruijn graph $G(\mathbf{WR}_k(n, [\ell, u]))$ is generally not regular. A random edge $s_1 s_2 \cdots s_n$ can be generated using values for $WR_k(n, [\ell, u])$ as outlined in Algorithm 3. The algorithm essentially unranks a string in $\mathbf{WR}_k(n, [\ell, u])$ as it appears in lexicographic order.

[2] We added a value corresponding to the number of times the current vertex had been visited to the output of `rand()`.

**Lemma 4.2.** *Algorithm 3 can generate a string $s_1 s_2 \cdots s_n$ from $\mathbf{WR}_k(n, [\ell, u])$ uniformly at random using $\mathcal{O}(kn^3)$ simple operations on numbers up to $k^n$.*

*Proof.* Consider Algorithm 3. The required values $WR_k(n, [\ell, u])$, for $1 \leq j \leq n$ can be precomputed via dynamic programming using $\mathcal{O}(kn^3)$ simple operations on numbers up to $k^n$. The outer **for** loop iterates at most $n-1$ times and each iteration requires $\mathcal{O}(k+n)$ simple operations on numbers up to $k^n$. □

**Algorithm 3** Random generation of a string $s_1 s_2 \cdots s_n$ in $\mathbf{WR}_k(n, [\ell, u])$

$r \leftarrow$ random integer in $[1,\ WR_k(n, [\ell, u])]$
**for** $j$ **from** $1$ **to** $n-1$ **do**
  **for** $i$ **from** $0$ **to** $k-1$ **do** $n_i \leftarrow WR_k(n-j, [\ell-i, u-i])$
  $i \leftarrow 0$
  **while** $r > n_i$ **do** $r \leftarrow r - n_i;\ i \leftarrow i+1$
  $s_j \leftarrow i$
  $\ell \leftarrow \ell - i;\ u \leftarrow u - i$
**if** $\ell < 0$ **then** $\ell \leftarrow 0$
$s_n \leftarrow \ell + r - 1$

Table 6 shows the minimum, maximum, and average ratios of the cover time to total edges in $G(\mathbf{WR}_2(n, [5, 10]))$ by running Algorithm R for 10,000 iterations.

| | **Ratio**: cover time / $WR_2(n, [5, 10])$ | | |
|---|---|---|---|
| $n$ | **Min** | **Max** | **Avg** |
| 10 | 2.3 | 19.4 | 5.1 |
| 11 | 2.5 | 15.9 | 5.5 |
| 12 | 3.2 | 13.0 | 6.0 |
| 13 | 3.9 | 15.6 | 6.5 |
| 14 | 4.2 | 15.8 | 7.2 |
| 15 | 5.2 | 14.7 | 7.9 |
| 16 | 6.0 | 16.0 | 8.6 |
| 17 | 6.7 | 15.2 | 9.3 |
| 18 | 7.4 | 17.0 | 10.2 |
| 19 | 8.1 | 17.5 | 10.8 |
| 20 | 9.1 | 17.2 | 11.5 |

TABLE 6. The minimum, maximum, and average ratios of the cover time to total edges in the de Bruijn graph $G(\mathbf{WR}_2(n, [5, 10]))$ by running Algorithm R for 10,000 iterations.

Weight-range de Bruijn sequences can be constructed via an $\mathcal{O}(n)$ time per symbol successor rule when the minimum weight is 0, or the maximum weight is $(k-1)^n$ [GSWW20]. In the binary case, they can be constructed for any weight range in $\mathcal{O}(1)$ amortized time [SWW13]. When $k = 2$ and $\ell + 1 = u$, weight-range de Bruijn sequences correspond to the universal cycles for (shorthand) subsets discussed in Section 4.1.

### 4.5. DE BRUIJN SEQUENCES WITH FORBIDDEN $0^z$

A *necklace class* is an equivalence class of strings under rotation; we call the lexicographically smallest string in the class a *necklace*. The necklace class containing $\alpha$ is denoted $[\alpha]$. For example, if $\alpha = 0001$ then $[\alpha] = \{0001, 0010, 0100, 1000\}$. Let $\mathbf{N}_k(n, z)$ denote the set of all necklaces in $\mathbf{\Sigma}_k(n)$ with no $0^z$ substring for $z > 1$. All such necklaces end with 1 when $z \leq n$. Let $\mathbf{Z}_k(n, z) = \bigcup_{\alpha \in \mathbf{N}_k(n,z)} [\alpha]$. It is known that $\mathbf{Z}_k(n, z)$ admits a maximal length universal cycle that does not contain the substring $0^z$ [7]. We call a maximum length universal cycle that does not contain $0^z$ as a substring, a *de Bruijn sequence with forbidden* $0^z$.

The de Bruijn graph $G(\mathbf{Z}_k(n, z))$ is not necessarily regular. A random edge can be generated by applying the following recurrences. Let $F_k(n, z)$ denote the number of $k$-ary strings of length $n$ with no $0^z$ substring. It satisfies the following recurrence for $z < n$:

$$F_k(n, z) = (k-1) \sum_{j=1}^{z} F_k(n - j, z),$$

where $F_k(n, z) = k^n$ for $z > n$ and $F_k(n, n) = k^n - 1$.

Let $Z_k(n, z)$ denote the number of $k$-ary strings of length $n$ with no $0^z$ substring, including the wraparound. It satisfies the following recurrence for $z < n$ obtained by partitioning the strings into those beginning with a non-zero, and those with $j$ zeros in the wraparound, in which case there are $k-1$ possibilities for each of the first non-zero and last non-zero:

$$Z_k(n, z) = (k-1)F_k(n - 1, z) + (k-1)^2 \sum_{j=1}^{z-1} j \cdot F_k(n - j - 2, z),$$

where $Z_k(n, z) = k^n$ for $z > n$ and $Z_k(n, n) = k^n - 1$.

Given these recurrences, we can compute a random string in $\mathbf{F}_k(n, z)$ following a similar unranking strategy using lexicographic order as done with $\mathbf{WR}_k(n, [\ell, u])$ in the previous subsection. We omit the details in this case. Table 7 shows the minimum, maximum, and average ratios of the cover time to total edges in $G(\mathbf{F}_2(n, 2))$ and $G(\mathbf{F}_2(n, 3))$ by running Algorithm R for 10,000 iterations.

The lexicographically smallest de Bruijn sequences with forbidden $0^z$ can be generated via a simple greedy algorithm [24]; it can also be generated efficiently by concatenating the aperiodic prefixes of necklaces with no $0^z$ substring as they appear in lexicographic order [12, 24]. An exponential number of such sequences can be efficiently generated by applying cycle-joining as described in [7].

### 4.6. ORIENTABLE SEQUENCES

Recall the definitions of a necklace class and necklace from the previous subsection. A *bracelet class* is an equivalence class of strings under rotation and reversal; we call the

| $n$ | **Ratio**: cover time / $F_2(n,2)$ **Min** | **Max** | **Avg** |
|---|---|---|---|
| 8 | 1.0 | 20.9 | 3.6 |
| 9 | 1.2 | 15.9 | 3.9 |
| 10 | 1.7 | 21.2 | 4.6 |
| 11 | 2.0 | 13.5 | 4.8 |
| 12 | 2.4 | 17.6 | 5.7 |
| 13 | 2.9 | 15.7 | 5.7 |
| 14 | 3.3 | 19.0 | 6.4 |
| 15 | 3.8 | 18.1 | 6.9 |
| 16 | 4.3 | 21.9 | 7.4 |
| 17 | 4.3 | 15.9 | 7.8 |
| 18 | 5.2 | 21.9 | 8.4 |
| 19 | 5.9 | 18.7 | 8.8 |
| 20 | 6.4 | 17.2 | 9.3 |
| 21 | 6.9 | 21.2 | 9.7 |
| 22 | 7.6 | 20.1 | 10.2 |
| 23 | 8.0 | 18.7 | 10.7 |
| 24 | 8.3 | 24.8 | 11.2 |

| $n$ | **Ratio**: cover time / $F_2(n,3)$ **Min** | **Max** | **Avg** |
|---|---|---|---|
| 8 | 1.4 | 21.5 | 4.3 |
| 9 | 1.8 | 13.3 | 4.6 |
| 10 | 2.5 | 18.7 | 5.3 |
| 11 | 3.1 | 14.8 | 5.8 |
| 12 | 3.8 | 17.5 | 6.5 |
| 13 | 4.1 | 16.5 | 7.0 |
| 14 | 4.5 | 17.0 | 7.7 |
| 15 | 5.5 | 16.7 | 8.2 |
| 16 | 6.1 | 20.0 | 8.9 |
| 17 | 6.7 | 19.8 | 9.5 |
| 18 | 7.4 | 17.7 | 10.1 |
| 19 | 8.2 | 16.3 | 10.6 |
| 20 | 8.8 | 16.9 | 11.3 |
| 21 | 9.6 | 15.8 | 11.8 |
| 22 | 10.2 | 17.7 | 12.5 |
| 23 | 10.8 | 18.0 | 13.1 |
| 24 | 11.0 | 22.8 | 13.9 |

TABLE 7. The minimum, maximum, and average ratios of the cover time to total edges in the de Bruijn graphs $G(\mathbf{F}_2(n,2))$ (left) and $G(\mathbf{F}_2(n,3))$ (right) by running Algorithm R for 10,000 iterations.

lexicographically smallest string in the class a *bracelet*. A *bracelet* is said to be *asymmetric* if it is not in the same necklace class as its reversal. For example, 001011 is an asymmetric bracelet, but 001001 is not. Let $\mathbf{AB}_k(n)$ denote the set of all $k$-ary asymmetric bracelets of length $n$, and let $\mathbf{OS}_k(n) = \bigcup_{\alpha \in \mathbf{AB}_k(n)} [\alpha]$. Let $OS_k(n)$ denote $|\mathbf{OS}_k(n)|$.

For example, $\mathbf{AB}_2(7) = \{0001011, 0010111\}$, and

$$\begin{aligned}\mathbf{OS}_2(7) \quad = \quad & \{0001011, 0010110, 0101100, 1011000, 0110001, 1100010, 1000101\} \cup \\ & \{0010111, 0101110, 1011100, 0111001, 1110010, 1100101, 1001011\},\end{aligned}$$

where $OS_2(7) = 14$.

An *orientable sequence* is cyclic sequence such that each length-$n$ substring occurs at most once in *either direction*. For example, a maximum-length orientable sequence for $n = 5$ and $k = 2$ is 001101. A universal cycle for $\mathbf{OS}_k(n)$ is known to be an orientable sequence with asymptotically optimal length [3, 6]. A formula for $OS_k(n)$, is given in [8, 13]. Generating a random string from $\mathbf{OS}_k(n)$ does not appear to be a trivial matter. However, by randomly generating $k$-ary strings with rejection, on average only two random strings need to be generated to obtain a string in $\mathbf{OS}_k(n)$ as $n$ gets large. Thus, the expected time to generate a random edge in $G(\mathbf{OS}_k(n))$ is $\Theta(n)$.

| | **Ratio**: cover time / $OS_2(n)$ | | |
|---|---|---|---|
| $n$ | **Min** | **Max** | **Avg** |
| 6 | 0.8 | 0.8 | 0.8 |
| 7 | 0.9 | 8.5 | 1.5 |
| 8 | 0.9 | 18.8 | 3.2 |
| 9 | 1.3 | 13.2 | 4.3 |
| 10 | 2.1 | 14.7 | 5.0 |
| 11 | 2.7 | 18.4 | 5.7 |
| 12 | 3.6 | 16.3 | 6.5 |
| 13 | 4.2 | 17.0 | 7.2 |
| 14 | 5.1 | 17.6 | 7.9 |
| 15 | 5.3 | 18.0 | 8.5 |
| 16 | 6.2 | 18.9 | 9.2 |
| 17 | 7.0 | 18.4 | 9.9 |
| 18 | 7.8 | 18.6 | 10.6 |
| 19 | 8.6 | 17.7 | 11.2 |
| 20 | 9.5 | 17.7 | 11.8 |

TABLE 8. The minimum, maximum, and average ratios of the cover time to total edges in the de Bruijn graph $G(\mathbf{OS}_2(n))$ by running Algorithm R for 10,000 iterations.

Table 8 illustrates the minimum, maximum, and average ratios of the cover time to total edges in $G(\mathbf{OS}_2(n))$ by running Algorithm R for 10,000 iterations.

Orientable sequences with asymptotically optimal length can be constructed in $\mathcal{O}(n)$ time per symbol using $\mathcal{O}(n)$ space [13]; in the binary case, they can be constructed in $\mathcal{O}(1)$ amortized time per bit using $\mathcal{O}(n^2)$ space.

## REFERENCES


[1] OEIS Foundation Inc. (2025), Entry A000670 in The On-Line Encyclopedia of Integer Sequences, https://oeis.org/A000670.

[2] De Bruijn sequence and universal cycle constructions (2025). http://debruijnsequence.org., 2025.

[3] ALHAKIM, A., MITCHELL, C. J., SZMIDT, J., AND WILD, P. R. Orientable sequences over non-binary alphabets. In *Cryptography and Communications (to appear)* (2024).

[4] ALTSCHUL, S. F., AND ERICKSON, B. W. Significance of nucleotide sequence alignments: a method for random sequence permutation that preserves dinucleotide and codon usage. *Molecular Biology and Evolution 2*, 6 (11 1985), 526–538.

[5] ARNDT, J. *Generating Random Permutations*. Phd thesis, Australian National University, 2010.

[6] BURNS, J., AND MITCHELL, C. Position sensing coding schemes. In *Cryptography and Coding III (M.J.Ganley, ed.)* (1993), Oxford University Press, pp. 31–66.

[7] CHEE, Y. M., ETZION, T., NGUYEN, T. L., TA, D. H., TRAN, V. D., AND VU, V. K. Maximum length RLL sequences in de Bruijn graph, arXiv preprint arXiv:2403.01454, 2024.

[8] DAI, Z.-D., MARTIN, K., ROBSHAW, B., AND WILD, P. Orientable sequences. In *Cryptography and Coding III (M.J.Ganley, ed.)* (1993), Oxford University Press, pp. 97–115.

[9] DURSTENFELD, R. Algorithm 235: Random permutation. *Commun. ACM 7*, 7 (July 1964), 420.

[10] FISHER, R. A., AND YATES, F. *Statistical Tables for Biological, Agricultural and Medical Research*. Oliver and Boyd, London, 1938.
[11] FLEURY, P.-H. Deux problèmes de géométrie de situation. *Journal de mathématiques élémentaires 42* (1883), 257–261.
[12] GABRIĆ, D., AND SAWADA, J. Constructing de Bruijn sequences by concatenating smaller universal cycles. *Theoret. Comput. Sci. 743* (2018), 12–22.
[13] GABRIĆ, D., AND SAWADA, J. Constructing $k$-ary orientable sequences with asymptotically optimal length. *Designs, Codes and Cryptography* (Feb 2025).
[14] GABRIĆ, D., SAWADA, J., WILLIAMS, A., AND WONG, D. A successor rule framework for constructing $k$ -ary de Bruijn sequences and universal cycles. *IEEE Transactions on Information Theory 66*, 1 (2020), 679–687.
[15] HOLROYD, A. E., RUSKEY, F., AND WILLIAMS, A. Shorthand universal cycles for permutations. *Algorithmica 64*, 2 (2012), 215–245.
[16] KANDEL, D., MATIAS, Y., UNGER, R., AND WINKLER, P. Shuffling biological sequences. *Discrete Applied Mathematics 71*, 1 (1996), 171–185.
[17] KNUTH, D. E. *The Art of Computer Programming, Volume 2: Seminumerical Algorithms*, 3rd ed., vol. 2. Addison-Wesley, Reading MA, 1997.
[18] LIPTÁK, Z., AND PARMIGIANI, L. A BWT-based algorithm for random de Bruijn sequence construction. In *LATIN 2024, LNCS 14578* (2024), pp. 130–145.
[19] PROPP, J. G., AND WILSON, D. B. How to get a perfectly random sample from a generic Markov chain and generate a random spanning tree of a directed graph. *Journal of Algorithms 27*, 2 (1998), 170–217.
[20] RUSKEY, F., SAWADA, J., AND WILLIAMS, A. De Bruijn sequences for fixed-weight binary strings. *SIAM J. Discrete Math. 26*, 2 (2012), 605–617.
[21] RUSKEY, F., AND WILLIAMS, A. An explicit universal cycle for the (n-1)-permutations of an n-set. *ACM Trans. Algorithms 6*, 3 (July 2010), 1–12.
[22] SAWADA, J., SEARS, J., TRAUTRIM, A., AND WILLIAMS, A. Concatenation trees: A framework for efficient universal cycle and de Bruijn sequence constructions. *arXiv preprint arXiv:2308.12405* (2024).
[23] SAWADA, J., AND WILLIAMS, A. A universal cycle for strings with fixed-content. *Manuscript* (2021).
[24] SAWADA, J., WILLIAMS, A., AND WONG, D. Generalizing the classic greedy and necklace constructions of de Bruijn sequences and universal cycles. *Electron. J. Combin. 23*, 1 (2016), Paper 1.24, 20.
[25] SAWADA, J., AND WONG, D. Efficient universal cycle constructions for weak orders. *Discrete Mathematics 343*, 10 (2020), 112022.
[26] WILSON, D. B. Generating random spanning trees more quickly than the cover time. In *Proceedings of the Twenty-Eighth Annual ACM Symposium on Theory of Computing* (New York, NY, USA, 1996), STOC '96, Association for Computing Machinery, p. 296–303.

Communicated by (The editor will be set by the publisher).
(The dates will be set by the publisher).