Universal cycles for weight-range binary strings

Joe Sawada[1⋆], Aaron Williams[2⋆⋆], and Dennis Wong[1⋆⋆⋆]

[1] School of Computer Science, University of Guelph, Canada.
[2] Department of Mathematics and Statistics, McGill University, Canada.

Abstract. We present an efficient universal cycle construction for the set of binary strings of length n whose weight (number of 1s) are in the range $c, c+1, \ldots, d$ where $0 \leq c < d \leq n$. The construction can be implemented to generate each character in constant amortized time using $O(n)$ space which based on a simple lemma for gluing universal cycles together. The Gluing lemma can also be applied to construct universal cycles for other combinatorial objects including passwords and labeled graphs.

1 Introduction

Let $\mathbf{B}(n)$ denote the set of all binary strings of length n. A *universal cycle* for a set $\mathbf{S}$ is a cyclic sequence $u_1 u_2 \cdots u_{|\mathbf{S}|}$ where each substring of length n corresponds to a unique object in $\mathbf{S}$. When $\mathbf{S} = \mathbf{B}(n)$ these sequences are commonly known as *de Bruijn sequences* [6, 7, 13] and efficient constructions are well known [10, 9, 16]. For example, the cyclic sequence 0000100110101111 is a universal cycle (de Bruijn sequence) for $\mathbf{B}(4)$; the 16 unique substrings of length 4 when the sequence is considered cyclicly are:

$$0000, 0001, 0010, 0100, 1001, 0011, 0110, 1101, 1010, 0101, 1011, 0111, 1111, 1110, 1100, 1000.$$

Universal cycles have been studied for a variety of combinatorial objects including permutations, partitions, subsets, labeled graphs, various functions, and passwords [1, 3–5, 12, 14, 15, 17, 20]. In this paper we focus on the set $\mathbf{B}_c^d(n)$, which denotes the subset of $\mathbf{B}(n)$ containing strings with *weight* (number of 1s) in the range $c, c+1, \ldots, d$, or in other words, the subset of $\mathbf{B}(n)$ containing strings with *weight-range* from c to d. As an example, $\mathbf{B}_2^3(4) = \{0011, 0101, 0110, 1001, 1010, 1100, 0111, 1011, 1101, 1110\}$ and a universal cycle for this set is 0011101011. Using standard techniques, it can be shown that universal cycles exist for all $\mathbf{B}_c^d(n)$ where $0 \leq c < d \leq n$. When $c = d$, they exist only when $c \in \{0, 1, n-1, n\}$. However, finding efficient constructions remains a difficult problem. In this paper, a universal cycle has an *efficient construction* if each successive symbol of the sequence can be generated in constant amortized time (CAT) while using a polynomial amount of space with respect to the objects in the sequence. By *constant amortized time* we mean amortized $O(1)$-time, where the constant does not depend on the size of the objects in the sequence. Some special cases have been previously studied:

- if $c = d - 1$, then an efficient construction is known [19],
- if $c = 0$ or $d = n$, then an efficient construction is known [23],

⋆ Research supported by NSERC. email: jsawada@uoguelph.ca
⋆⋆ email: haron@uvic.ca
⋆⋆⋆ email: cwong@uoguelph.ca

– if $d - c + 1$ is even, then an inefficient construction is known [24, 25].

This paper provides the first efficient construction for universal cycles of $\mathbf{B}_c^d(n)$ for all $0 \leq c < d \leq n$. By applying the efficient construction for the case when $c = d - 1$, our construction yields an algorithm that can generate each character in constant amortized time using $O(n)$ space.

The rest of this paper is presented as follows. In Section 2 we provide a simple proof for the existence of universal cycle for $\mathbf{B}_c^d(n)$ where $0 \leq c < d \leq n$; a more complicated proof is given in [2]. In Section 3, we present a generic result that states when two universal cycles can be glued together to obtain a new universal cycle. We then apply this result in Section 4 and Section 5 to provide an efficient construction of universal cycles for $\mathbf{B}_c^d(n)$. We conclude with Section 6, where we consider other combinatorial objects including passwords and labeled graphs.

2 Universal cycle existence for $\mathbf{B}_d^c(n)$

The *de Bruijn graph* $G(\mathbf{S})$ for a set of length n strings $\mathbf{S}$ is a directed graph whose vertex set consists of the length $n-1$ prefixes and suffixes of the strings in $\mathbf{S}$. For each string $b_1 \cdots b_n \in \mathbf{S}$ there is an edge labeled b_n that is directed from the prefix $b_1 \cdots b_{n-1}$ to the suffix $b_2 \cdots b_n$. Thus, the graph has $|\mathbf{S}|$ edges. As an example, the de Bruijn graph $G(\mathbf{B}_2^4(4))$ in illustrated in Fig. 1.

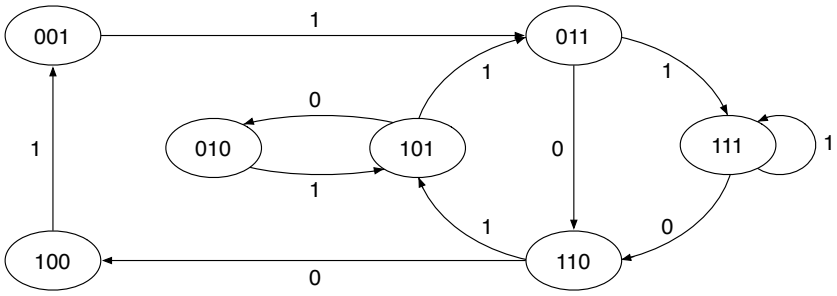

Fig. 1: The de Bruijn graph $G(\mathbf{B}_2^4(4))$.

A (directed) cycle in a directed graph $G = (V, E)$ is a sequence $v_1, \cdots, v_j, v_1$ where $v_i \in V$ and $\{v_i, v_{i+1}\} \in E$. A directed graph is said to be *Eulerian* if it contains a directed cycle that includes each edge exactly once. It is well known that $\mathbf{S}$ admits a universal cycle if and only if $G(\mathbf{S})$ is Eulerian. If $G(\mathbf{S})$ contains an Eulerian cycle, then a universal cycle is produced by traversing an Eulerian cycle and outputting the edge labels. However, in practice, such a method for producing a universal cycle is often impractical due to the size of the graph that must be stored in memory. For example, the memory required to store the de Bruijn graph for $\mathbf{B}(n)$ is $O(2^n)$.

A directed graph is said to be *balanced* if the in-degree of each vertex is the same as its out-degree. It is *strongly connected* if there is a directed path between every pair of vertices. The following result is well-known, and appears in many references such as [18]:

Lemma 1. *A directed graph is Eulerian if and only if it is balanced and strongly connected.*

Theorem 1. $G(\mathbf{B}_c^d(n))$ *is Eulerian for* $0 \leq c < d \leq n$.

Proof. We prove that $G(\mathbf{B}_c^d(n))$ is Eulerian by showing that it is balanced and strongly connected.

Balanced: The vertex set of $G(\mathbf{B}_c^d(n))$ contains all strings of length $n-1$ with weight in the range $c-1, c, \ldots, d$. Each vertex with weight $c-1$ has one incoming edge and one outgoing edge, each labeled 1. Each vertex with weight d has one incoming edge and one outgoing edge, each labeled 0. All other vertices have in-degree and out-degree equal to 2.

Strongly connected: We apply induction on the size of the weight-range $c, c+1, \ldots, d$. The base case when $c = d-1$ is proved in Theorem 2.4 of [19]. For the inductive step assume that $G(\mathbf{B}_c^{d-1}(n))$ is strongly connected for $0 \leq c < d-1$, and consider $G(\mathbf{B}_c^d(n))$. Observe:

- the vertex set of $G(\mathbf{B}_c^d(n))$ is equal to the union of the vertex sets of $G(\mathbf{B}_c^{d-1}(n))$ and $G(\mathbf{B}_{d-1}^d(n))$,
- the intersection of the vertex sets for $G(\mathbf{B}_c^{d-1}(n))$ and $G(\mathbf{B}_{d-1}^d(n))$ is non-empty,
- the edge sets of $G(\mathbf{B}_c^{d-1}(n))$ and $G(\mathbf{B}_{d-1}^d(n))$ are both subsets of the edge set for $G(\mathbf{B}_c^d(n))$.

Thus, since both $G(\mathbf{B}_c^{d-1}(n))$ and $G(\mathbf{B}_{d-1}^d(n))$ are strongly connected (inductive hypothesis and base case), there will be a directed path between any two vertices in $G(\mathbf{B}_c^d(n))$. □

3 Gluing universal cycles

In this section we consider concatenating two universal cycles together to obtain a new universal cycle. An Eulerian cycle of a directed graph can be obtained by Hierholzer's algorithm [18, 11]. Hierholzer's approach is to construct an Eulerian cycle by exhaustively concatenating edge-disjoint cycles that share a common vertex. The algorithm repeatedly applies the following lemma to produce an Eulerian cycle.

Lemma 2. *Let* $G = (V, E)$ *and* $H = (V', E')$ *be two Eulerian graphs such that* $V \cap V' \neq \emptyset$ *and* $E \cap E' = \emptyset$. *Let* $\boldsymbol{C}_G = u_1, u_2, \cdots, u_j, u_1$ *and* $\boldsymbol{C}_H = v_1, v_2, \cdots, v_k, v_1$ *denote Eulerian cycles in* G *and* H *respectively such that* $u_1 = v_1$. *Then the concatenation of the two cycles* $\boldsymbol{C}_{GH} = u_1, \cdots, u_j, v_1, \cdots, v_k, v_1$ *is an Eulerian cycle for* $G \cup H$.

As mentioned in Section 2, each universal cycle for a set $\mathbf{S}$ corresponds to an Eulerian cycle of its de Bruijn graph $G(\mathbf{S})$. Thus by Lemma 2, universal cycles for two sets $\mathbf{S}_1$ and $\mathbf{S}_2$ can be joined together to form a new universal cycle for $\mathbf{S}_1 \cup \mathbf{S}_2$ if $G(\mathbf{S}_1)$ and $G(\mathbf{S}_2)$ are edge-disjoint and share a common vertex, or in other words, $\mathbf{S}_1$ and $\mathbf{S}_2$ are disjoint and have elements that share a length $n-1$ prefix or a length $n-1$ suffix. As an example, consider the following two universal cycles:

- universal cycle for $\mathbf{B}_1^2(5)$: 00<u>00</u>1<u>1</u>000101001,
- universal cycle for $\mathbf{B}_3^4(5)$: <u>0011</u>11011010111.

The de Bruijn graphs $G(\mathbf{B}_1^2(5))$ and $G(\mathbf{B}_3^4(5))$ are edge-disjoint and share a common vertex $\alpha = 0011$. Since the universal cycles are cyclic they can be re-written as 0011000101001000 and 0011110110101111 respectively. By gluing these two strings together, observe that we obtain a universal cycle for $\mathbf{B}_1^4(5) = \mathbf{B}_1^2(5) \cup \mathbf{B}_3^4(5)$:

$$0011000101001000011110110101111.$$

This example illustrates the following lemma which considers gluing universal cycles for an alphabet of arbitrary size:

Lemma 3 (The Gluing lemma). *Let U_1 and U_2 be universal cycles for the sets of length n string $\mathbf{S_1}$ and $\mathbf{S_2}$, where $\mathbf{S_1} \cap \mathbf{S_2} = \emptyset$ and the length $n-1$ prefixes of U_1 and U_2 are the same. Then the concatenated string $U_1 \cdot U_2$ is a universel cycle for $\mathbf{S_1} \cup \mathbf{S_2}$.*

The phrasing of the Gluing lemma with respect to universal cycles provides a simple method for "constructing" new universal cycles from existing ones. We apply this lemma in the upcoming sections to construct weight-range universal cycles for $\mathbf{B}_c^d(n)$.

4 Universal cycle construction for $\mathbf{B}_c^d(n)$

As mentioned earlier, there exists an efficient universal cycle construction for $\mathbf{B}_{d-1}^d(n)$ [19]. By leveraging this result, we can use the Gluing lemma to create a universal cycle for binary strings with an *even weight-range* ($d - c + 1$ is even). To create universal cycles for binary strings with an *odd weight-range* ($d - c + 1$ is odd), we will glue in individual necklaces which are defined below.

4.1 Preliminary definitions and notations

A *necklace* is defined to be the lexicographically smallest string in an equivalence class of strings under rotation. Aperiodic necklaces are called *Lyndon words*. A string is called a *prenecklace* if it is the prefix of some necklace. The *aperiodic prefix* of a string α, denoted as $ap(\alpha)$, is its shortest prefix whose repeated concatenation yields α. That is, the aperiodic prefix of $\alpha = a_1 a_2 \cdots a_n$ is the shortest prefix $ap(\alpha) = a_1 a_2 \cdots a_k$ such that $(ap(\alpha))^{n/k} = \alpha$, where exponentiation denotes repeated concatenation. For example, when $\alpha = 001001001$, $ap(\alpha) = 001$. We denote $\mathbf{N}_k(n)$ to be the set of binary necklaces of length n and weight k.

To further illustrate these objects, the prenecklaces, necklaces and Lyndon words in $\mathbf{B}_4(6)$ are listed as follows:

Prenecklaces	Necklaces	Lyndon words
001111	001111	001111
010111	010111	010111
011011	011011	
011101		
011110		

4.2 Even weight-range

Suppose we want to construct a universal cycle for $\mathbf{B}^d_{d-3}(n)$ from the universal cycles for $\mathbf{B}^{d-2}_{d-3}(n)$ and $\mathbf{B}^d_{d-1}(n)$. Observe that the sets $\mathbf{B}^{d-2}_{d-3}(n)$ and $\mathbf{B}^d_{d-1}(n)$ are disjoint, and their universal cycles share many length $n-1$ substrings (those with weight $d-2$). Thus, we can apply the Gluing lemma to construct a universal cycle for $\mathbf{B}^d_{d-3}(n)$. We can then repeatedly apply the Gluing lemma on the resulting universal cycle and *dual-weight universal cycles* (weight-range universal cycles with $d-c+1=2$) of lower weight-ranges to obtain a universal cycle of an arbitrary even weight-range. However, the difficult task remains: How can we produce the glued universal cycle efficiently, that is, without scanning for common substrings of length $n-1$?

To find an efficient construction, we must revisit the efficient construction of a universal cycle for $\mathbf{B}^d_{d-1}(n)$. The universal cycle UC^d_{d-1} for $\mathbf{B}^d_{d-1}(n)$ can be efficiently constructed by the algorithm described in [19] as follows:

1. List the necklaces of length $n+1$ and weight d in reverse *cool-lex order* [21],
2. Append the aperiodic prefixes of the necklaces together to get the string UC^d_{d-1}.

As an example, Fig. 2 demonstrates a dual-weight universal cycle for $\mathbf{B}^4_3(7)$ constructed by concatenating the aperiodic prefixes of $\mathbf{N}_4(8)$ in reverse cool-lex order.

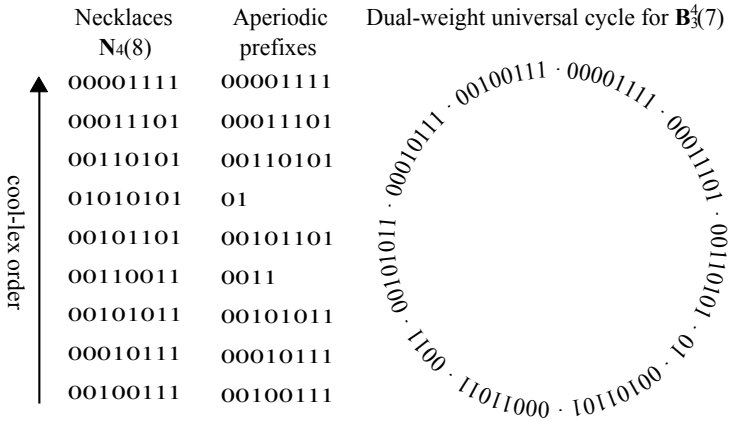

Fig. 2: Concatenating the aperiodic prefixes of $\mathbf{N}_4(8)$ in reverse cool-lex order to create a dual-weight universal cycle for $\mathbf{B}^4_3(7)$.

Using this construction, we can find the common length $n-1$ substring α of the universal cycles $UC^{d-2}_{d-3}(n)$ and $UC^d_{d-1}(n)$ by the following lemma:

Lemma 4. [19] *The first necklace in reverse cool-lex order for* $\mathbf{B}_d(n+1)$ *is* $0^{n-d+1}1^d$.

Applying this lemma, the first $n+1$ characters in $UC^{d-2}_{d-3}(n)$ and $UC^d_{d-1}(n)$ are $0^{n-d+3}1^{d-2}$ and $0^{n-d+1}1^d$ respectively. Thus, if we rotate $UC^{d-2}_{d-3}(n)$ to the left by 2 characters, then the first $n-1$ characters of both cycles will be $0^{n-d+1}1^{d-2}$.

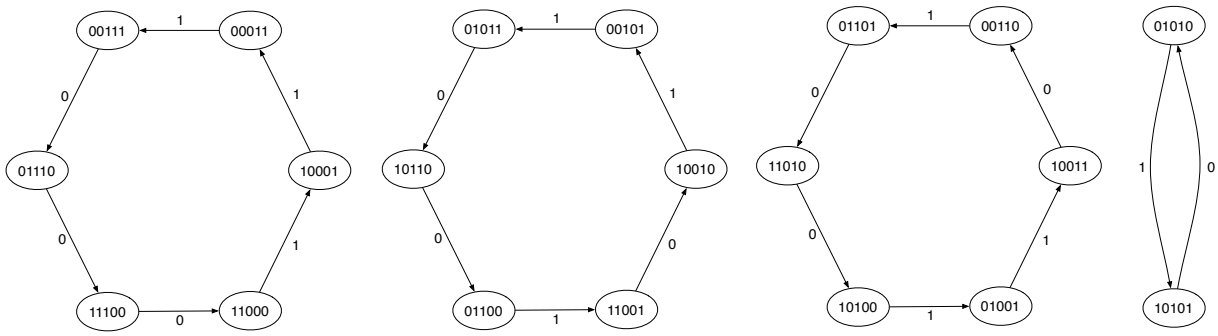

Fig. 3: Illustrating the de Bruijn graph corresponding to the universal cycles of the 4 necklace equivalence classes $\mathbf{Neck}(000111)$, $\mathbf{Neck}(001011)$, $\mathbf{Neck}(001101)$ and $\mathbf{Neck}(010101)$ that make up $\mathbf{B}_3(6)$.

Let $VC^d_{d-1}(n)$ denote the sequence $UC^d_{d-1}(n)$ with the first 2 characters removed. The following recursive formula provides a construction of the universal cycle $UE^d_c(n)$ for $\mathbf{B}^d_c(n)$, where the weight-range $d-c+1$ is even:

$$UE^d_c(n) = \begin{cases} UC^d_{d-1}(n) & \text{if } c = d-1; \\ UC^d_{c+2}(n) \cdot VC^{c+1}_c(n) \cdot 00 & \text{if } c < d-1. \end{cases}$$

By expanding the recursive function, we obtain the following formula:

$$UE^d_c(n) = UC^d_{d-1}(n) \cdot VC^{d-2}_{d-3}(n) \cdot VC^{d-4}_{d-5}(n) \cdots VC^{c+1}_c(n) \cdot 0^{d-c-1}.$$

Theorem 2. *$UE^d_c(n)$ is a universal cycle for $\mathbf{B}^d_c(n)$ when $0 \leq c < d \leq n$ and $d-c+1$ is even.*

Since the universal cycle UC^d_{d-1} can be constructed in constant amortized time per character using $O(n)$ space by the algorithm described in [19], and VC^d_{d-1} can easily be obtained by constructing UC^d_{d-1} and removing its first 2 characters, we therefore arrive the following theorem:

Theorem 3. *A universal cycle $UE^d_c(n)$ for $\mathbf{B}^d_c(n)$ can be constructed in constant amortized time per character using $O(n)$ space when $d-c+1$ is even and $0 \leq c < d \leq n$.*

4.3 Extending the weight-range (odd weight-range)

In this section, we consider modifying a universal cycle for $\mathbf{B}^{d-1}_c(n)$ into a universal cycle for $\mathbf{B}^d_c(n)$. We will refer to this process as *incrementing* the universal cycle's weight range since the weight-range is incremented by one value. The process of incrementing the universal cycle's weight range allows us to extend an arbitrary even weight-range universal cycle to an odd weight-range universal cycle.

Let $\mathbf{Neck}(\alpha)$ denote the set of strings rotationally equivalent to α. We partition the strings in $\mathbf{B}_d(n)$ into their necklace equivalence classes such that $\mathbf{B}_d(n) = \mathbf{Neck}(n_1) \cup \mathbf{Neck}(n_2) \cup \cdots \cup \mathbf{Neck}(n_{|\mathbf{N}_d(n)|})$ where $n_j \in \mathbf{N}_d(n)$. For example, $\mathbf{B}_3(6)$ can be partitioned into the 4 subsets $\mathbf{B}_3(6) = \mathbf{Neck}(000111) \cup \mathbf{Neck}(001011) \cup \mathbf{Neck}(001101) \cup \mathbf{Neck}(010101)$ with the elements of each set listed as follows:.

- ▷ $\mathbf{Neck}(000111) = \{000111, 001110, 011100, 111000, 110001, 100011\}$,
- ▷ $\mathbf{Neck}(001011) = \{001011, 010110, 101100, 011001, 110010, 100101\}$,
- ▷ $\mathbf{Neck}(001101) = \{001101, 011010, 110100, 101001, 010011, 100110\}$, and
- ▷ $\mathbf{Neck}(010101) = \{010101, 101010\}$.

Observe that the de Bruijn graph $G(\mathbf{Neck}(n_j))$ forms a simple cycle with its edge labels corresponds to $ap(n_j)$, which is a universal cycle for $\mathbf{Neck}(n_j)$. As an example, Fig. 3 illustrates the de Bruijn graphs for the 4 necklace equivalence classes that make up $\mathbf{B}_3(6)$. The concatenation of edge labels of the cycles are 000111, 001011, 001101, and 01 respectively, which correspond to the universal cycles for $\mathbf{Neck}(000111)$, $\mathbf{Neck}(001011)$, $\mathbf{Neck}(001101)$ and $\mathbf{Neck}(010101)$.

A special case arises when a universal cycle for $\mathbf{Neck}(n_j)$ has length less than n that happens when n_j is periodic. In such a universal cycle, each length n element of $\mathbf{Neck}(n_j)$ is obtained by repeatedly traversing the universal cycle $\frac{n}{|\mathbf{Neck}(n_j)|}$ times. As an example, a universal cycle for $\mathbf{Neck}(010101) = 01$ where the characters of the universal cycle correspond to the strings in $\mathbf{Neck}(010101)$, that is 010101 and 101010 when repeatedly traversing the universal cycle 3 times. We consider the length $n-1$ prefix of such a universal cycle to be the length $n-1$ prefix of the length n string that corresponds to the first character of the universal cycle. For example, the length 5 prefix of the universal cycle 01 for $\mathbf{Neck}(010101)$ is 01010.

Recall that by the Gluing lemma, two universal cycles can be joined together if the sets for the universal cycles are disjoint and the length $n-1$ prefixes are the same. Observe that $\mathbf{B}_c^{d-1}(n)$ and $\mathbf{Neck}(n_j)$ are disjoint for each $n_j \in \mathbf{N}_d(n)$, and we can rotate the universal cycles such that their first length $n-1$ prefixes are the same. We can then apply the Gluing lemma to repeatedly concatenate universal cycles for each necklace equivalence class $\mathbf{Neck}(n_j)$ with the universal cycle for $\mathbf{B}_c^{d-1}(n)$. For example, consider the universal cycles for $\mathbf{Neck}(000111)$, $\mathbf{Neck}(001011)$, $\mathbf{Neck}(001101)$, $\mathbf{Neck}(010101)$ and $\mathbf{B}_1^2(6)$: 000001100001010001001. By repeatedly applying the Gluing lemma, we obtain a universal cycle for $\mathbf{B}_1^3(n)$:

Step	U_1	U_2	$U_1 \cdot U_2$
1	000111	000110000101000100100	000111000110000101000100100
2	001101	000111000110000101000100100	000111000110100110000101000100100
3	001011	000111000110100110000101000100100	000111000110100110000101100101000100100
4	01	000111000110100110000101100101000100100	000111000110100110000101100101010000100100

Observe that the order of inserting the universal cycles for $\mathbf{Neck}(n_j)$ does not affect the final universal cycle.

A *linear universal string* for a universal cycle is a linear sequence obtained by appending the first $n-1$ characters of a universal cycle to its end. For example, the linear universal string for the universal cycle U for $\mathbf{B}(4)$: 0000100110101111 is 0000100110101111000. Let $u_1u_2\cdots u_m$ denote the linear universal string obtained from UD_c^{d-1}, where UD_c^{d-1} is a universal cycle for $\mathbf{B}_c^{d-1}(n)$ where $m = |UD_c^{d-1}| + n - 1$. The linear universal string $u_1u_2\cdots u_m$ contains each element for $\mathbf{B}_c^{d-1}$ exactly once, the string $\alpha\cdot 0$ therefore exists as a substring of the linear universal

string when $\alpha \cdot 1 \in \mathbf{N}_d(n)$. Thus, we can increment the weight-range of UD_c^{d-1} and output a universal cycle for $\mathbf{B}_c^d(n)$ as follows:

for t **from** $n-1$ **to** m **do**
  if $\alpha \cdot 1 \in \mathbf{N}_d(n)$ **then**
    Print$(ap(\alpha \cdot 1))$
  Print(u_s)

Let UD_c^d denote the output string that results from this construction.

Theorem 4. *UD_c^d is a universal cycle for $\mathbf{B}_c^d(n)$ when $0 \leq c < d \leq n$.*

Proof. The string $ap(\alpha \cdot 1)$ is a universal cycle for $\mathbf{Neck}(\alpha \cdot 1)$. Since $\mathbf{B}_c^{d-1}(n)$ and $\mathbf{Neck}(\alpha \cdot 1)$ are disjoint and have the same length $n-1$ prefix, that is α, by the Gluing lemma the construction exhaustively concatenate universal cycles for each necklace equivalence class $\mathbf{Neck}(n_j)$ and UD_c^{d-1}. The resulting string UD_c^d is a universal cycle for the set $\mathbf{B}_c^{d-1}(n) \cup \mathbf{Neck}(n_1) \cup \mathbf{Neck}(n_2) \cup \cdots \cup \mathbf{Neck}(n_{|\mathbf{N}_d(n)|})$, that is $\mathbf{B}_c^d(n)$. □

5 Implementation

In this section, we consider the problem of efficiently modifying a universal cycle for $\mathbf{B}_c^{d-1}(n)$ into a universal cycle for $\mathbf{B}_c^d(n)$. We assume that there is an efficient algorithm that outputs the universal cycle for $\mathbf{B}_c^{d-1}(n)$ one character at a time. We buffer this output into a sliding window, and examine it to determine if any additional characters need to be output. We first describe how this process works, and then we describe how to make the process efficient.

5.1 A simple algorithm: SimpleIncrement()

The construction SimpleIncrement() follows the approach in Section 4.3. The algorithm reads each character from a linear universal string $u_1 \cdots u_m$. It examines the sliding window $\alpha = u_s \cdots u_t$ of size $n-1$ and inserts $ap(\alpha \cdot 1)$ if $\alpha \cdot 1 \in \mathbf{N}_d(n)$. The weight of α is maintained by the variable w which can be easily updated with a constant amount of computation per character. To examine if $\alpha \cdot 1 \in \mathbf{N}_d(n)$, we apply Duval's algorithm [8] which returns 0 if α is not a necklace, or otherwise returns the length of $ap(\alpha)$ which runs in $O(n)$ time. The length of $ap(\alpha)$ is maintained by the variable p.

Pseudocode that produces the universal cycle $UD_c^d(n)$ is illustrated by the procedure SimpleIncrement() in Fig. 4. The initial call is SimpleIncrement(). The procedure calls the function AperiodicPrefix$(x_1 \cdots x_n)$ to examine if $\alpha \cdot 1 \in \mathbf{N}_d(n)$, which is the implementation of Duval's algorithm. Thus, each time we read a character from an input linear universal string we need $O(n)$ amount of computation to examine $\alpha \cdot 1$ for its aperiodic prefix. The sliding window of size $n-1$ can be implemented using circular array data structure that requires a constant amount of computation to update using $O(n)$ space.

```
function AperiodicPrefix(x_1 ··· x_n)
int p
 1: p ← 1
 2: for i from 2 to n do
 3:     if x_{i−p} < x_i then p ← i
 4:     else if x_{i−p} > x_i then return 0
 5: if n mod p = 0 then return p
 6: else return 0
```

```
procedure SimpleIncrement()
int p, w
 1: s ← 1
 2: for t from n − 1 to m do
 3:     w ← Weight(u_s ··· u_t)
 4:     p ← AperiodicPrefix(u_s ··· u_t 1)
 5:     if p > 0 and w = d − 1 then
 6:         Print(u_s ··· u_{s+p−2} 1) // Insertion of ap(α · 1)
 7:     Print(u_s)
 8:     s ← s + 1
```

Fig. 4: Pseudocode of SimpleIncrement().

Thus, in addition to the time and space required to produce an input linear universal string for $\mathbf{B}_c^{d-1}(n)$, the algorithm SimpleIncrement() uses an additional $O(n)$ amount of computation per character and $O(n)$ space to constructs a universal cycle $UD_c^d(n)$ for $\mathbf{B}_c^d(n)$. From Theorem 3, we can construct even weight-range universal cycle in constant amortized time per character using $O(n)$ space, we therefore arrive the following theorem:

Theorem 5. *A universal cycle $UD_c^d(n)$ for $\mathbf{B}_c^d(n)$ can be constructed in $O(n)$ amortized time per character using $O(n)$ space for any weight-range where $0 \leq c < d \leq n$.*

5.2 Extending SimpleIncrement() to CAT

The major overhead of SimpleIncrement() in runtime comes from the function AperiodicPrefix($x_1 \cdots x_n$) that examines the sliding window for its aperiodic prefix using $O(n)$ amount of computation per character. To efficiently locate the position to insert the aperiodic prefixes, we instead maintain a sliding window $\beta = u_s \cdots u_t$ of variable size which stores the longest prenecklace of length at most n that ends at position t. Each time we read a character from an input linear universal string, we increment t and update the length of the aperiodic prefix p if β is a prenecklace. If β is not a prenecklace, then we update s to $s + \lfloor \frac{t-s}{p} \rfloor \cdot p$ and update p to be the length of $ap(u_{s+\lfloor \frac{t-s}{p} \rfloor \cdot p} \cdots u_t)$. The computation required to maintain the variables s and p is constant per character when the size of the sliding window has not reached n.

The content of the sliding window β is a length n prenecklace when its size reaches n. If $u_{t-p} < 1$ and the weight of β is $d-1$, then $u_s \cdots u_{t-1} \cdot 1 \in \mathbf{N}_d(n)$ and we insert $ap(u_s \cdots u_{t-1} \cdot 1)$. The insertion requires a constant amount of computation per character. We scan β for the next starting position s' such that $u_{s'} \cdots u_t$ is a prenecklace and update s to s' which requires $O(n)$ amount of computation per character. However, the scan takes place only when the size of the sliding window reaches n, which is bounded by the number of prenecklaces in $\mathbf{B}_c^d(n)$.

The number of length n prenecklaces of the weight-range c to d divided by the number of elements in $\mathbf{B}_c^d(n)$ is bounded by a constant factor of $\frac{1}{n}$, thus the total amount of computation required to update all prenecklaces is proportional to the length of universal cycle generated. The sliding window of size at most n can be implemented using circular array data structure that

requires a constant amount of computation per character to update while using $O(n)$ space. The implementation detail and runtime analysis of the algorithm are discussed in the Appendix.

Thus, in addition to the time and space required to produce an input linear universal string for $\mathbf{B}_c^{d-1}(n)$, the algorithm uses an additional constant amount of computation per character and O(n) space to constructs a universal cycle $UD_c^d(n)$ for $\mathbf{B}_c^d(n)$. From Theorem 3, we can construct even weight-range universal cycle in constant amortized time per character using $O(n)$ space, we therefore arrive the following theorem:

Theorem 6. *A universal cycle $UD_c^d(n)$ for $\mathbf{B}_c^d(n)$ can be constructed in constant amortized time per character using $O(n)$ space for any weight-range where $0 \leq c < d \leq n$.*

6 Other applications of the Gluing lemma

In this section we consider other sets of strings and their associated universal cycles and apply the Gluing lemma to produce new universal cycles.

6.1 Passwords

In [17], a *passwords* is defined to be all strings of length n over an alphabet of size k partitioned into $q < k$ classes where each string contains at least one character from each class. For instance, 4 natural classes would be: lower case letters, upper case letters, digits, and special characters. A very secure password would contain one symbol from each class. They prove the following result:

Theorem 7. [17] *A universal cycle exists for all n-letter passwords over an alphabet of size k partitioned into $q < k$ classes, provided that $n \geq 2q$.*

We relax the definition of a *password* to be a string that contains at least one symbol from $q' \leq q$ classes. In fact, this is a common requirement of passwords where they must either contain a number or a special character. As an example, consider all passwords of length n containing characters in *at least* two classes. Such strings can be partitioned into $\binom{4}{2}$ sets of words containing exactly 2 classes, plus 4 sets of words containing exactly 3 classes, plus one set containing characters from all 4 classes. Observe that all sets are disjoint, and the sets containing strings from exactly 2 classes have many strings that have $n-1$ characters in common. For instance 'aAAAAAAA' and '1AAAAAAA' and '#AAAAAAA'. Similarly, there exist common strings of length $n-1$ between a set of exactly 2 classes and a set with one additional class. For instance 'aAAAAAAA' and 'aAAAAAA3'. Thus, the following theorem follows from Lemma 3:

Theorem 8. *Let an alphabet of size k be partitioned into $q < k$ classes. There exists a universal cycle for all strings of length n containing letters from at least $q' \leq q$ classes, provided that $n \geq 2q$.*

Observe that if $q' = 1$, then the universal cycle is a traditional de Bruijn sequence over an alphabet of size k.

6.2 Labeled graphs

In [3], a number of universal cycle existence questions are given for various labeled graphs. Instead of strings, they construct graph with a sliding window of size k that represents the labeled graph. In particular, they give the following result:

Theorem 9. [3] *Universal cycle exists for labeled graphs with precisely m edges (and k vertices).*

Since graphs with m edges and graphs with $m+1$ edges are disjoint and their universal cycles have many graphs with identical $k-1$ *windows*, we can apply Lemma 3 to obtain the following result:

Theorem 10. *Universal cycle exists for labeled graphs with between m_1 and m_2 edges (and k vertices).*

It remains an open problem to find efficient constructions for such universal cycles.

References

1. A. Bechel, B. LaBounty-Lay, and A. Godbole. Universal cycles of discrete functions. In *Proceedings of the Thirty-Ninth Southeastern International Conference on Combinatorics, Graph Theory and Computing. Congressus Numerantium 189*, pages 121–128, 2008.
2. A. Blanca and A. Godbole. On universal cycles for new classes of combinatorial structures. *SIAM J. Discret. Math.*, 25(4):1832–1842, December 2011.
3. G. Brockman, B. Kay, and E. Snively. On universal cycles of labeled graphs. *Electronic Journal of Combinatorics*, 17(1):9 pp, 2010.
4. K. Casteels and B. Stevens. Universal cycles for $(n-1)$-partitions of an n-set. *Discrete Mathematics*, 309:5341–5340, 2009.
5. F. Chung, P. Diaconis, and R. Graham. Universal cycles for combinatorial structures. *Discrete Mathematics*, 110:43–59, December 1992.
6. N. G. de Bruijn. A combinatorial problem. *Koninklijke Nederlandse Akademie v. Wetenschappen*, 49:758–764, 1946.
7. N. G. de Bruijn. Acknowledgement of priority to C. Flye Sainte-Marie on the counting of circular arrangements of $2n$ zeros and ones that show each n-letter word exactly once. *T.H. Report 75-WSK-06*, page 13, 1975.
8. J. P. Duval. Factorizing words over an ordered alphabet. *J. Algorithms*, 4(4):363–381, 1983.
9. H. Fredericksen and I. J. Kessler. An algorithm for generating necklaces of beads in two colors. *Discrete Mathematics*, 61:181–188, 1986.
10. H. Fredericksen and J. Maiorana. Necklaces of beads in k colors and k-ary de Bruijn sequences. *Discrete Mathematics*, 23:207–210, 1978.
11. C. Hierholzer and C. Wiener. Ueber die Möglichkeit, einen Linienzug ohne Wiederholung und ohne Unterbrechung zu umfahren. *Mathematische Annalen*, 6(1):30–32, March 1873.
12. G. Hurlbert. On universal cycles for k-subets of an n-element set. *Siam Journal on Discrete Mathematics*, 7:598–604, 1994.
13. G. Hurlbert, B. Jackson, and B. Stevens (Editors). Generalisations of de Bruijn sequences and Gray codes. *Discrete Mathematics*, 309:5255–5348, 2009.
14. B. Jackson. Universal cycles of k-subsets and k-permutations. *Discrete Mathematics*, 117:114–150, 1993.
15. R. Johnson. Universal cycles for permutations. *Discrete Mathematics*, 309:5264–5270, 2009.
16. D. E. Knuth. Generating all tuples and permutations, fascicle 2. *The Art of Computer Programming*, 4, 2005.
17. A. Leitner and A. Godbole. Universal cycles of classes of restricted words. *Discrete Mathematics*, 310:3303–3309, 2010.
18. K. H. Rosen. *Discrete Mathematics and Its Applications*. McGraw-Hill Higher Education, 5th edition, 2002.
19. F. Ruskey, J. Sawada, and A. Williams. De Bruijn sequences for fixed-weight binary strings. *SIAM Journal on Discrete Mathematics*, 26(2):605–617, 2012.
20. F. Ruskey and A. Williams. An explicit universal cycle for the $(n-1)$-permutations of an n-set. *ACM Transactions on Algorithms*, 6(3):12 pp, 2010.
21. F. Ruskey, A. Williams, and J. Sawada. Binary bubble languages and cool-lex order. *J. Comb. Theory, Ser. A*, 119(1):155–169, 2012.
22. J. Sawada and F. Ruskey. An efficient algorithm for generating necklaces with fixed density. In Robert Endre Tarjan and Tandy Warnow, editors, *SODA*, pages 752–758. ACM/SIAM, 1999.
23. J. Sawada, B. Stevens, and A. Williams. De Bruijn sequences for the binary strings with maximum specified density. In *Proceeding of 5^{th} International Workshop on Algorithms and Computation (WALCOM 2011), New Dehli, India, LNCS*, 2011.
24. B. Stevens and A. Williams. The coolest order of binary strings. In Evangelos Kranakis, Danny Krizanc, and Flaminia L. Luccio, editors, *FUN*, volume 7288 of *Lecture Notes in Computer Science*, pages 322–333. Springer, 2012.
25. B. Stevens and A. Williams. The coolest way to generate binary strings. *Theory of Computer Systems*, 2013. To appear.

7 Appendix

In this section we provide the implementation detail and analysis of the efficient algorithm discussed in Section 5.2. The algorithm runs in constant amortized time per character using $O(n)$ space.

7.1 A CAT algorithm: FastIncrement()

The algorithm to efficiently increment the weight-range of a weight-range universal cycle can be organized into three stages as follows:

Glue universal cycles: We insert $ap(u_s \cdots u_{t-1} \cdot 1)$ before the position s if $u_s \cdots u_{t-1} \cdot 1 \in \mathbf{N}_d(n)$. The string $u_s \cdots u_{t-1} \cdot \ 1 \in \mathbf{N}_d(n)$ if $t - s + 1 = n$, $w = d - 1$ and $u_t = 0$.

Maintain prenecklace: We maintain the variables s and p such that β is a prenecklace and p is the length of $ap(\beta)$. There are a few possible cases here:

- if $u_{t-p} < u_t$, then β is a prenecklace and $ap(\beta) = u_s \cdots u_t$, thus we update $p = |\beta| = t-s+1$;
- if $u_{t-p} = u_t$, then β is a prenecklace and the aperiodic prefix remains unchanged, that is $ap(\beta) = ap(u_s \cdots u_{t-1}) = u_s \cdots u_{s+p-1}$, we keep the variables s and p unchanged;
- if $u_{t-p} > u_t$, then β is not a prenecklace; we update s to $s + \lfloor \frac{t-s}{p} \rfloor \cdot p$; we update p to be the length of $ap(u_{s+\lfloor \frac{t-s}{p} \rfloor \cdot p} \cdots u_t)$;
  - ▷ for example, consider $n = 13$, $\beta = u_1 \cdots u_{12} = 001001001000$ and $p = 3$; β is not a necklace because $u_{12-3} > u_{12}$; the variable s is thus updated to $1 + \lfloor \frac{12-1}{3} \rfloor \cdot 3 = 10$ such that the sliding window starts with $u_{10} \cdots u_{12} = 000$ which is lexicographically smaller than $u_1 \cdots u_3 = 001$; p is updated to the length of $ap(u_{10} \cdots u_{12}) = |ap(000)| = 1$.

Maintain window-size: When the size of the sliding window β reaches n, we increment the variable s to s' such that $u_{s'} \cdots u_t$ is a prenecklace; we update p to be the length of $ap(u_{s'} \cdots u_t)$.

Pseudocode of such construction is illustrated by the procedure FastIncrement() in Fig. 5. The initial call is FastIncrement(). The function Update(k, w) scans the string $u_k \cdots a_t$ to update s to s' such that $u_{s'} \cdots u_t$ is a prenecklace and p to be the length of $ap(u_{s'} \cdots u_t)$. Despite the extra *for* loop in line 7 - 8 of Update(k, w), each time we reach the *for* loop, s' is updated to $s' + \lfloor \frac{i-s'}{p} \rfloor \cdot p$ and we output the string $u_{s'} \cdots u_{s'+\lfloor \frac{i-s'}{p} \rfloor \cdot p-1}$. Since $|u_{s'+\lfloor \frac{i-s'}{p} \rfloor \cdot p} \cdots u_i| \leq |u_{s'} \cdots u_{s'+\lfloor \frac{i-s'}{p} \rfloor \cdot p-1}|$, line 4 - 8 of Update($k$, w) requires constant amount of computation per character.

Analysis: We analyze the amount of computation divided by the number of characters output, and show that the total amount of computation of FastIncrement() divided by the characters output is bounded by a constant. The computation required for each of the above stage is as follows:

Glue universal cycles: The string $u_s \cdots u_{t-1} \cdot 1 \in \mathbf{N}_d(n)$ if $w = d-1$, $u_t = 0$ and $t-s+1 = n$, this can be verified using only a constant amount of computation per character.

Maintain prenecklace: If $u_{t-p} \leq u_t$, then β is a prenecklace and we update p which requires a constant amount of computation; if $a_{t-p} > a_t$, we call the function Update($s + \lfloor \frac{t-s}{p} \rfloor \cdot p$, w) which requires $O(t - s - \lfloor \frac{t-s}{p} \rfloor \cdot p)$ amount of computation; however, we output at least $\lfloor \frac{t-s}{p} \rfloor \cdot p$

```
function Update(int k, int w)
int p, s', w'
 1: s' ← k; p ← 1; w' ← w
 2: for i from k + 1 to t do
 3:     if u_{i−p} < u_i then p ← i − s' + 1
 4:     else if u_{i−p} > u_i then
 5:         s' ← s' + ⌊(i−s')/p⌋ · p
 6:         p ← 1
 7:         for j from s' + 1 to i do
 8:             if u_{j−p} < u_j then p ← j − s' + 1
 9: // Update weight w
10: for i from s to s' − 1 do
11:     if u_i = 1 then w' ← w − 1
12: return (s', p, w')
```

```
procedure FastIncrement()
int p, s', w
 1: p ← 1; w ← 0; s ← 1
 2: for t from 1 to m do
 3:     w ← w + u_t
 4:     // Glue universal cycles
 5:     if t − s + 1 = n and w = d − 1 and u_t = 0 then
 6:         if u_{t−p} < 1 then Print(u_s ··· u_{t−1} 1)
 7:         else Print(u_s ··· u_{s+p−1})
 8:     // Maintain prenecklace
 9:     if u_{t−p} < u_t then p ← t − s + 1
10:     else if u_{t−p} > u_t then
11:         (s', p, w) ← Update(s + ⌊(t−s)/p⌋ · p, w)
12:         Print(u_s ··· u_{s'−1})
13:         s ← s'
14:     // Maintain window-size
15:     if t − s + 1 = n then
16:         if p > n/2 then (s', p, w) ← Update(s + 1, w)
17:         else (s', p, w) ← Update(s + p, w)
18:         Print(u_s ··· u_{s'−1})
19:         s ← s'
```

Fig. 5: Pseudocode of FastIncrement().

characters. Since $t - s - \lfloor \frac{t-s}{p} \rfloor \cdot p < \lfloor \frac{t-s}{p} \rfloor \cdot p$, the amount of computation is proportional to the number of characters output.

Maintain window-size: The size of the sliding window reaches n. It calls the function Update($s + 1$, w) to update the variables s and p but outputs only one character, thus it requires $O(n)$ amount of computation per character.

From the analysis of each stage, only stage 3 of the algorithm requires more than a constant amount of computation per character. Observe that the algorithm will only go through stage 3 if β is a length n prenecklace. We show that the number of length n prenecklaces is bounded by the number of elements in the universal cycle over n, and thus the total amount of computation required to update all prenecklaces is proportional to the length of universal cycle generated.

We denote $\mathbf{L}_k(n)$ and $\mathbf{P}_k(n)$ be the set of binary Lyndon words and prenecklaces of length n and weight k respectively. Let $N(n,k)$, $L(n,k)$ and $P(n,k)$ denote the cardinality of $\mathbf{N}_k(n)$, $\mathbf{L}_k(n)$ and $\mathbf{P}_k(n)$, and let $P_0(n,k)$ and $P_1(n,k)$ denote the cardinality of the set of binary prenecklaces of length n and weight k that end with the character 0 and 1 respectively. The following lemma provides an upper bound of $P_1(n,k)$ in terms of $N(n,k)$ and $L(n,k)$:

Lemma 5. [22] $P_1(n,k) \leq N(n,k) + L(n,k)$.

Consider the upper bound of $P_0(n,k)$, replacing the last character of a prenecklace of weight k that ends with a 0 with the character 1 will always yield a unique necklace of weight $k+1$, $P_0(n,k)$ is therefore bounded by the number of necklaces of weight $k+1$:

Lemma 6. $P_0(n,k) \leq N(n,k+1)$.

The upper bound of $N(n,k)$ and $L(n,k)$ in terms of $\binom{n}{k}$ has been discussed in [22] and are given as follows:

$$L(n,k) \;\leq\; \frac{1}{n}\binom{n}{k} \quad \text{and} \quad N(n,k) \;\leq\; 2L(n,k) \;\leq\; \frac{2}{n}\binom{n}{k}.$$

The upper bound of $P(n,k)$ in terms of $\binom{n}{k}$ is therefore as follows:

$$\begin{aligned} P(n,k) &= P_0(n,k) + P_1(n,k) \\ &\leq N(n,k+1) + N(n,k) + L(n,k) \\ &\leq \frac{2}{n}\binom{n}{k+1} + \frac{2}{n}\binom{n}{k} + \frac{1}{n}\binom{n}{k} \\ &\leq \frac{2}{n}\binom{n}{k+1} + \frac{3}{n}\binom{n}{k}. \end{aligned}$$

Theorem 11. *Algorithm FastIncrement() is a CAT algorithm.*

Proof. Let hn be the amount of computation required in stage 3 of FastIncrement() to update the prenecklace, where h is a constant. The ratio between the total amount of computation required in stage 3 of FastIncrement() to the number of elements in the universal cycle for $\mathbf{B}_c^d(n)$ is as follows:

$$\begin{aligned} \frac{\text{Total computation in stage 3}}{|B_c^d|} &= \frac{(P(n,d-1) + P(n,d-2) + \cdots + P(n,c)) \times hn}{\binom{n}{d} + \binom{n}{d-1} + \cdots + \binom{n}{c}} \\ &\leq \frac{(\frac{2}{n}\binom{n}{d} + \frac{5}{n}\binom{n}{d-1} + \frac{5}{n}\binom{n}{d-2} + \cdots + \frac{5}{n}\binom{n}{c+1} + \frac{3}{n}\binom{n}{c}) \times hn}{\binom{n}{d} + \binom{n}{d-1} + \cdots + \binom{n}{c}} \\ &\leq \frac{(2\binom{n}{d} + 5\binom{n}{d-1} + 5\binom{n}{d-2} + \cdots + 5\binom{n}{c+1} + 3\binom{n}{c}) \times h}{\binom{n}{d} + \binom{n}{d-1} + \cdots + \binom{n}{c}} \\ &< 5h. \end{aligned}$$

Since stage 1 and 2 of FastIncrement() requires only constant amount of computation per character, the algorithm FastIncrement() is a CAT algorithm. □

As discussed in Section 4.2, there exists a CAT construction for even weight-range universal cycles using $O(n)$ space. The algorithm FastIncrement() can efficiently increment the weight-range of a weight-range universal cycle using an $O(n)$ space circular array, therefore a universal cycle $UD_c^d(n)$ for $\mathbf{B}_c^d(n)$ can be constructed in constant amortized time per character using $O(n)$ space.