Properties of de Bruijn Sequences and Graphs, and a Generating Algorithm

Reed Nelson

May 2021

1 Introduction

De Bruijn sequences are named after Nicolaas Govert de Bruijn, a Dutch mathematician who wrote about them in his 1946 paper *A Combinatorial Problem* [1]. The idea, however, dates back to at least the 19th century.

In this paper we will define what a de Bruijn sequence is, and seek to better the reader's understanding by exploring some of their interesting combinatorial and graph theoretical properties. We will proceed to apply some of these properties in the process of developing an algorithm to generate all de Bruijn sequences for a given alphabet and window size. This algorithm will be discussed both in terms of the the broad mathematical process the algorithm uses, and in more specific terms of its practical implementation. Following that will be a brief discussion of applications of these sequences in genomics and in card magic.

2 Properties

2.1 Definition and Notation

Let A be an alphabet of k distinct characters. A *de Bruijn sequence* is a cyclic string of characters from A[1], where each permutation of n characters appears exactly once. $B(k, n)$ will be used to denote the set of de Bruijn sequences of window length n and alphabet length k.

[1] A binary alphabet is often implied with de Bruijn sequences, but what this paper discusses applies for an arbitrary alphabet.

2.2 Basic Properties

Property 2.1. The length of a $B(k, n)$ sequence is k^n.

Proof. The number of n-permutations of k characters is k^n, and given by the definition, each of those permutations must appear exactly once. Take s to be a minimum-length sequence of length l consisting of all unique permutations in question. In each window length n sits one permutation. If the window is shifted over one character, this must give another unique permutation. Thus the smallest possible value of l would be k^n. Suppose $l > k^n$, then the number of windows in s would be greater than k^n, and thus s would contain non-unique permutations, and would not be a de Bruijn sequence. Therefore, the length of a $B(k, n)$ sequence is k^n. ■

Property 2.2. The number of distinct[2] $B(k, n)$ sequences equals $\frac{(k!)^{k^{n-1}}}{k^n}$.

Example 1. B(2,3) [3] has 2 distinct sequences, 8 characters in length:
00010111 11101000

2.3 De Bruijn Graphs

Each $B(k, n)$ has a corresponding de Bruijn graph. The example in Figure 1 shows the graph for $B(2, 3)$.

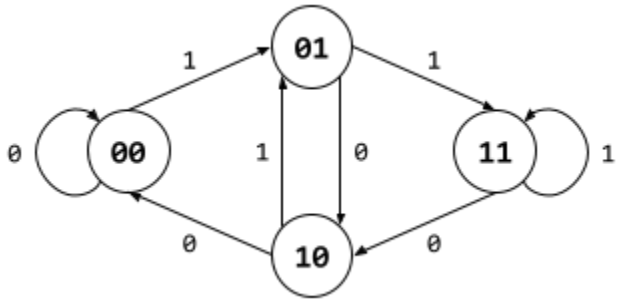

Figure 1: $B(2, 3)$ Graph

[2]Because de Bruijn sequences are cyclic, we consider mere rotations of one sequence to be the same (non-distinct) sequence.

[3]The nominal values of the characters used in the alphabet are unimportant, so for this paper and generally, when $k < 10$, we take $A = \{0, 1, \ldots, k\}$.

De Bruijn graphs are directed Eulerian graphs, which is to say that there exists a circuit within the graph which visits each edge exactly once. These graphs have a number of interrelated properties.

Property 2.3. For all vertices of $B(k, n)$, the indegree equals the outdegree which is equal to the size of the alphabet (k). This implies that for all k, each vertex has an even degree, which is consistent with Euler's Theorem[4].

Property 2.4. Each vertex represents a unique permutation of $n-1$ characters from A. From this, it follows that $B(k, n)$ is of order k^{n-1}.

Property 2.5. Each outgoing edge from each vertex is labeled with a different character $\alpha \in A$. Then any two of the following can be used to determine the label of the third: 2 adjacent vertices and their single adjoining edge. For example: the label of the vertex a particular edge points to can be determined by removing the leftmost character from the outgoing vertex, and appending to it the edge's label α. This is equivalent to shifting the window of a de Bruijn sequence right one character, and that new character being α.

Property 2.6. Because each vertex has exactly k incoming and outgoing edges, the number of edges of $B(k, n)$ are $k \cdot k^{n-1}$, or k^n. Notice that $B(k, n+1)$ has a number of vertices equal to the number of edges of $B(k, n)$. We can draw the graph for $B(k, n+1)$ in the following way: put a vertex where each edge of $B(k, n)$ would be, whose label is the character on that edge, concatenated to the source vertex. In this way, each vertex remains unique with respect to all other vertices in the $B(k, n+1)$ graph. Applying this method to the $B(2, 3)$ graph pictured above yields the $B(2, 4)$ graph pictured in Figure 2.

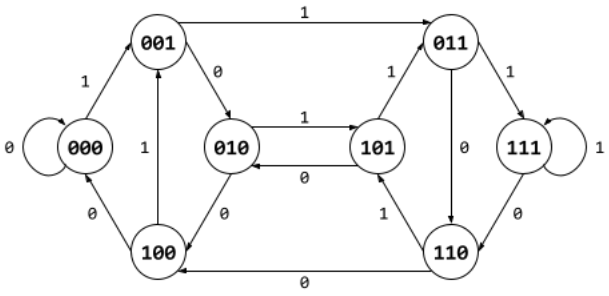

Figure 2: $B(2, 4)$ Graph

[4]Euler's Theorem states that a connected graph has an Eulerian circuit if and only if each vertex has an even degree.

Property 2.7. By following an Eulerian circuit through a $B(k, n-1)$ graph, and recording the label of each edge as it is visited, one gets a $B(k, n)$ sequence. Furthermore, the set of sequences constructed by taking all Eulerian circuits in the $B(k, n-1)$ graph is identical to the set of $B(k, n)$ sequences. This fact is key in creating a generating algorithm.

Property 2.8. An Eulerian traversal of a de Bruijn graph is a circuit.

Proof. The indegree of each edge of a de Bruijn graph equals its outdegree (Property 2.3). Let v be the start vertex of the Eulerian traversal. For v to have been traversed to, on each occasion, it must have first been traversed away from. Thus, once all incoming edges to v have been traversed, the outgoing edges necessarily have also been traversed. Therefore, if it is not the case that an incoming edge to v is the final edge to be traversed (making the traversal a circuit), then the traversal cannot be Eulerian. ■

3 A Generating Algorithm

3.1 Overview

The following method will generate all de Bruijn sequences of $B(k, n)$ for given values of k and n in two parts: first, generate the corresponding de Bruijn graph; second, traverse through it in all ways which visit each edge exactly once.

3.2 Constructing the Graph

The $B(k, n)$ graph will be represented by an adjacency matrix[5]. The data structure of choice will be a 2-dimensional array A_{ij}, for $0 \leq i, j \leq k^{n-1}$. This choice gives a space complexity of $O(k^{2n-2})$, which is generally unideal for sparse graphs, however we should expect time complexity to be a vastly larger bottleneck for most values of n and k. This choice gives a time complexity for accessing any element of the matrix of O(1).

Each vertex (a value in base-k) corresponds to the index of that value in decimal. Then whether one vertex points to another (or the same) depends on if the rightmost $n-1$ characters of the row index are the same as the leftmost $n-1$ characters of the column index (in the spirit of the example described

[5]An *adjacency matrix* is a boolean-valued n by n matrix where the indices of the rows and columns correspond to the vertices $1, \ldots, n$ in the graph, and the truth-value at a coordinate (i, j) depends on the existence of an edge from vertex i to vertex j.

under Property 2.5). The adjacency matrix corresponding to the $B(2,3)$ graph is pictured in Figure 3, and can be verified by examining Figure 1.

	00	01	10	11
00	1	1	0	0
01	0	0	1	1
10	1	1	0	0
11	0	0	1	1

Figure 3: $B(2,4)$ Adjacency Matrix

3.3 Traversing the Graph

From properties 2.7 and 2.8, we conclude that finding all B(k,n) sequences is equivalent to finding all distinct traversals of the B(k,n) graph which visit each edge once. This will be implemented by recursively traversing through the graph and identifying all traversals which meet that criterion, similar to a depth-first search. Each time a vertex is reached, there are two cases to consider:
Case: The vertex has non-traversed outgoing edges.
Then the edge with the label of lowest numerical value is followed, that edge is noted as visited, and it's label is appended to a list.
Case: The vertex has no non-traversed outgoing edges.
Then, if the list is length k^n, an Eulerian circuit has been completed, and the list is a de Bruijn sequence, so it is stored. If the list is less than length k^n, that traversal did not complete an Eulerian circuit and the list is not stored. In either case, the rightmost character is truncated from the list, and the vertex in question returns to the previously visited one.

This algorithm terminates when all paths in the graph have been traversed. Implementation involves the use of some additional data structures. A $k^{n-1} \times k^{n-1}$ boolean matrix can be used to keep track of whether an edge has yet been traversed, allowing for $O(1)$ access. Something in the form of a stack data structure ought to be used to store the values of the edges which have been traversed, as this allows for $O(1)$ pushing to and popping from the list.

3.4 Additional Notes

A Java implementation of an algorithm of this form is provided by myself, along with a suite of other tools pertaining to de Bruijn sequences and graphs. This code and select output files from the generating function can be found at github.com/reednel/DeBruijn.

In practice, it's biggest bottleneck is neither time nor space complexity (in runtime). The problem is one of volume of data output. These sequences grow in number and length extraordinarily fast (recall Properties 2.1 and 2.2). At 1 byte/character, a file containing all $B(2, 6)$ sequences would be 4 gigabytes in size. A file containing all $B(2, 7)$ sequences would be 65 thousand petabytes. Changing the alphabet makes for even more radical growth. A single $B(10, 10)$ sequence would be 10 gigabytes.

4 Applications

De Bruijn sequences have applications in a diverse set of areas. Most notable among them may be gene sequencing in the field of genomics. The basic idea is that codons can be thought of as characters in the alphabet A, and a de Bruijn sequence constructed from them provides an efficient, redundancy-free series of coding sequences. [2]

Another interesting application is in (especially self working)[6] card magic. For example, one might leverage the binary color options to order the deck in a $B(2, n)$ de Bruijn sequence. The spectator then draws n consecutive cards and recites their colors, on order. With this information and a bit of memorization ahead of time, the performer is able to identify exactly where in the sequence the spectator drew from, and thus what the values of their cards are. A strength of this trick is that one can cut the deck *ad infinitum* without ever breaking the sequence. Out of personal curiosity, the I have explored the possibility of strengthening this type of trick by using binary de Bruijn sequences which can be shuffled while retaining their order. The curious reader can find this work after the conclusion of the main paper.

For an idea of some of the other applications, de Bruijn sequences have been used in fMRI research, bit operations in computer science, and even in cracking PIN locks which accept rolling input.

[6] A *self working* trick refers to one which follows a fixed procedure that does not depend on circumstance or skill.

5 Summary

De Bruijn sequences and graphs have many interesting and not-too-complicated properties. This fact allowed me to extend my research beyond the existing literature and into a more exploratory domain. This resulted in the identification of several properties unstated in the literature, and an apparently novel generating algorithm.

While applications for de Bruijn sequences extend beyond what was discussed in section 4, none appear to be too grand or significant. The diversity of applications is perhaps more notable than the applications themselves.

References

[1] N. G. de Bruijn. A combinatorial problem. *Koninklijke Nederlandse Akademie van Wetenschappen te Amsterdam*, pages 758–764, 1946.

[2] Gongxin Peng, Peifeng Ji, and Fangqing Zhao. A novel codon-based de bruijn graph algorithm for gene construction from unassembled transcriptomes. *Genome Biology*, 17(1), 2016.

6 Shuffling of Binary de Bruijn Sequences

6.1 Shuffling

This extra work explores special binary de Bruijn sequences which have the property that after undergoing a perfect shuffle, they retain their original order. For brevity, we will say a binary de Bruijn sequence *shuffles* if and only if it has this property. Furthermore, in this section, perfect shuffle[7] refers to a perfect *in-shuffle* or perfect *out-shuffle*.

Let n be the length of a sequence, and i denote the index of each element in the sequence, for $0 \leq i \leq n-1$. An in-shuffle reorders the elements of the sequence according to the following rule:

$$\begin{array}{ll} i \to 2i+1, & \text{for } 0 \leq i < \frac{n}{2} \\ i \to 2i-n, & \text{for } \frac{n}{2} \leq i < n \end{array} \tag{1}$$

Example 1. Take a sequence to be of length 10. An in-shuffle reorders it's indices as follows (underlining for visual aid):

$$\underline{0123}456789 \to 5\underline{0}6\underline{1}7\underline{2}8\underline{3}9\underline{4}$$

Let n be the length of a sequence, and i denote the index of each element in the sequence, for $0 \leq i \leq n-1$. An out-shuffle reorders the elements of the sequence according to the following rule:

$$\begin{array}{ll} i \to 2i, & \text{for } 0 < i < \frac{n}{2} \\ i \to 2i-n+1, & \text{for } \frac{n}{2} \leq i < n \end{array} \tag{2}$$

Example 2. Take a sequence to be of length 10. An out-shuffle reorders it's indices as follows (underlining for visual aid):

$$\underline{0123}456789 \to \underline{0}5\underline{1}6\underline{2}7\underline{3}8\underline{4}9$$

6.2 Properties

Property 6.1. For a given n, it's not true that all $B(2,n)$ sequences shuffle.[8]

Property 6.2. Of sequences that shuffle, only certain rotations do.

[7] Concerning playing cards, a perfect shuffle (in or out) corresponds to a perfect *Faro shuffle*. In practice, it would take much skill to reliably execute the Faro shuffle perfectly.

[8] It would be of interest to prove that for all n, there exists a $B(2,n)$ sequence that shuffles.

6.3 Algorithms

I have written a suite of tools to identify $B(2, n)$ sequence that shuffle. For the reason that these algorithms are simpler and less interesting than the generating algorithm provided in the paper, they are left out here. Similar to the generating algorithm, the Java code for these tools can be found on github.com/reednel/DeBruijn. Below are some results from a function which checks each rotation of each sequence provided by the generator, and returns those which out-shuffle.

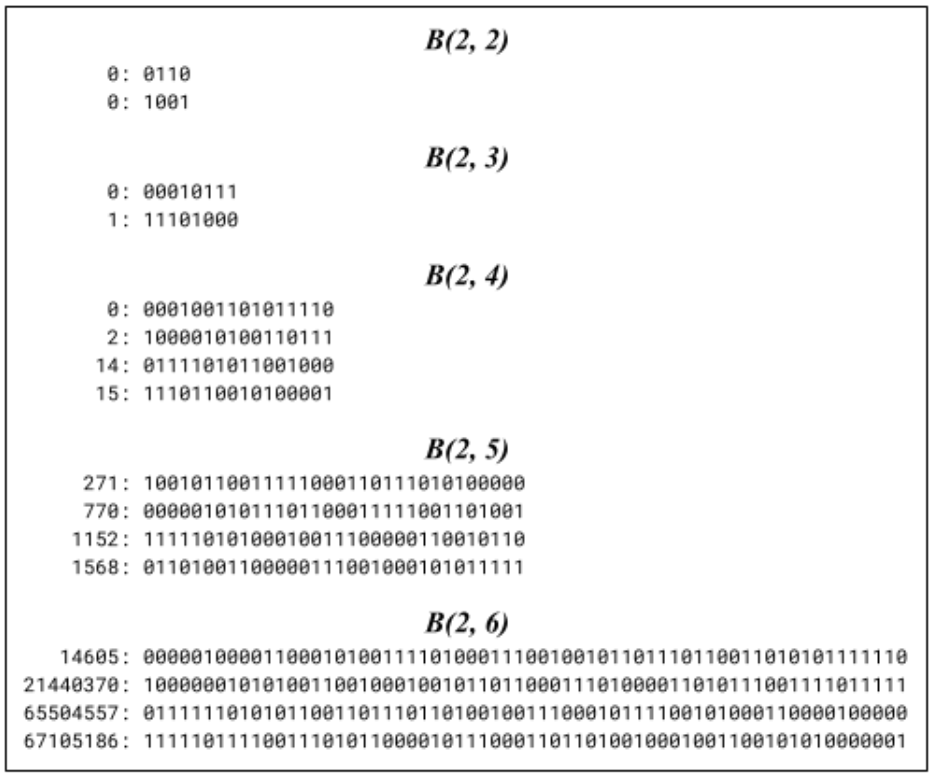

Figure 4: Sample out-shuffling de Bruijn sequences