Universal Cycles

CS7083 Final Project Report

Jayanth Dungavath

April 27, 2016

Contents

1 Abstract

The aim of this project is to implement and better understand the construction of de Bruijn Sequences and Universal Cycles using various algorithms like FKM algorithm by Fredricksen, Kessler and Maiorana (FKM) proposed in Fredricksen and Kessler [1] and in Fredricksen and Maiorana [2] and J. P. Duval [3] proposed an algorithm to produce Lyndon words and subsequently producing Universal Cycles. The study of these cycles has arisen in a variety of contexts like digital fault testing, pseudo-random number generations etc., there by triggering interest among researchers producing extensive results in these fields.

2 Introduction

2.1 de Bruijn Sequence

Let $\mathbf{T}(n,\ k)$ be the set of k-ary strings of length n. A de Bruijn sequence for $\mathbf{T}(n,\ k)$ is a sequence of length k^n that contains each string in $\mathbf{T}(n,\ k)$ exactly once as a substring when the sequence is viewed circularly [4]. For Example,

$$\mathbf{T}(2,3) = \{11, 12, 13, 21, 22, 23, 31, 32, 33\}$$

M. H. Martin's greedy algorithm [5] can construct a de Bruijn sequence for $\mathbf{T}(n,\ k)$ by starting a sequence with k^{n-1} (exponent denotes repetition of k by $n-1$ times) and repeatedly applying the following rule:

Append the smallest symbol in {1,2,...,k} *so that substrings of length n in the resulting linear sequence are distinct.*

When $n = 2$ and $k = 3$, the algorithm generates 3112132233 and a de Bruijn sequence is obtained by removing the initial k^{n-1} prefix. So, 112132233 is a de Bruijn Sequence for $T(2,$ 3). Though Martin's algorithm is easy to implement, it is a space complex algorithm taking $\Omega(k^n)$ space. An alternative, FKM algorithm is both space and time efficient algorithm and is discussed in a separate section.

2.2 Universal Cycles

Chung, Diaconis, and Graham in 1992 [6] have introduced the natural generalization of de Bruijn sequences for $\mathbf{T}(n,\ k)$ to subsets of $\mathbf{T}(n,\ k)$. Given a set of strings $\mathbf{S} \subseteq \mathbf{T}(n,\ k)$, a universal cycle for $\mathbf{S}$ is a sequence of length $|\mathbf{S}|$ that contains each string in $\mathbf{S}$ exactly once as a substring when the sequence is viewed circularly. For example, the subset $\mathbf{S_1} \subseteq \mathbf{T}(4,$ 3) of strings that have sum at least 10 is

$\mathbf{S_1}$ = {1333,2233,2323,2332,2333,3133,3223,3232,3233,3313,3322,3323,3331,3332,3333}

The universal cycle for $\mathbf{S_1}$ is

133322332323333

3 FKM Algorithm

FKM algorithm can be summarized as follows:

Concatenate the aperiodic prefixes of the necklaces in $\mathbf{T}(n, k)$ *in lexicographic order*

For example,

$\mathbf{T}(4,2)$ = {0000, 0001, 0010, 0011, 0100, 0101, 0110, 0111, 1000, 1001, 1010, 1011, 1100, 1101, 1110, 1111}

As the strings are already in lexicographic order, we need to find the aperiodic prefixes of the necklaces in $\mathbf{T}(n, k)$. A necklace or a Lyndon word can be defined as the lexicographically smallest string in an equivalence class strings under rotation. And, its aperiodic prefix is the shortest prefix that can be repeated to create the string itself. Following are the necklaces and their aperiodic prefixes of $\mathbf{T}(n, k)$:

Necklace	**Aperiodic Prefix**
0000	0
0001	0001
0011	0011
0101	01
0111	0111
1111	1

Concatenation of these aperiodic prefixes gives

$$0 \cdot 0001 \cdot 0011 \cdot 01 \cdot 0111 \cdot 1$$

Universal cycle 0000100110101111 contains each strings in $\mathbf{T}(4, 2)$ exactly once as a substring when it is viewed circularly. FKM algorithm always generates the same sequence for the given n, k.

4 J. P. Duval Algorithm

Duval in 1988 has given an algorithm that produces Lyndon words of length less than or equal to an integer n [3]. By concatenating the generated Lyndon words, we can get a Universal Cycle. Though the actual paper is not in English, I found the pseudo code of the algorithm [7] and implemented the same.

Algorithm is as follows:

1: w[1] ← a
2: i ← 1
3: **repeat**
4: **for** $j = 1$ to $n - i$
5: **do** $w[i + j] \leftarrow w[j]$
6: $i \leftarrow n$
7: **while** $i > 0$ **and** $w[i] = M$
8: **do** $i \leftarrow (i - 1)$
9: **if** $i > 0$
10: **then** $w[i] \leftarrow s(w[i])$
11: **until** $i = 0$

Letters a and M are respectively the first and last letters of the alphabet, and for every letter x excepted M, s(x) is the letter that follows x in the alphabet. We denote by w[1...n] an array of letters of dimension n.

5 Implementation

After completing the research and review of necessary material required for this project, I started implemeting these algorithms in Python. One entire module had been dedicated to implement FKM and another one to implement Duval's algorithm. Code samples are documented in the following section.

5.1 FKM algorithm implementation:

Implemented this algorithm based on the algorithm's description provided in [4]. Initially, all the words of length n are generated from the set of symbols s. Then necklaces are identified to later produce aperiodic prefixes which are then joined to produce a universal cycle.

Following universal cycle has been generated for $s = \{1, 2, 3\}$ and $n = 4$ with strings that have sum at least 10.

-bash-4.1$ python FKM.py
String: 123
Length: 4
1. De Bruijn Sequence
2. Universal Cycle
1/2? 2
Enter val such that each string has sum at least val: 10
Universal cycle: 133322332323333
-bash-4.1$

Code:

```
def toString(List):
    return ''.join(List)

strings = []
def stringPermutations (strng, dat, lst, idx):
    length = len(strng)
    for i in range(length):
        dat[idx] = string[i]
        if idx==lst:
            strings.append(toString(dat))
        else:
            stringPermutations (strng, dat, lst, idx+1)

def generateStrings (strng, length):
    global strings
    strings = []
    dat = [""] * (length+1)
    strng = sorted(strng)
    stringPermutations (strng, dat, length-1, 0)

def necklaces(p):
    for i in p:
        for j in range(1,len(i)):
            if (not(i[1:]+i[:1]==i)):
                x = i[j:]+i[:j]
                if x in p:
```

```
                    if (not(i==x)):
                        p.remove(x)
    return p

def atleast(x,val):
    p = []
    for i in x:
        sum = 0
        for j in i:
            sum += int(j)
        if (sum >= val):
            p.append(i)
    return p

def aperiodic(p):
    prefix = []
    for i in p:
        if(len(i)%2 == 0):
            if(i[1:]+i[:1]==i):
                prefix.append(i[:1])
            elif(i[:2]*(len(i)/2)==i):
                prefix.append(i[:2])
            elif(i[:len(i)/2]*2==i):
                prefix.append(i[:len(i)/2])
            else:
                prefix.append(i)
        elif(len(i)==9):
            if(i[1:]+i[:1]==i):
                prefix.append(i[:1])
            elif(i[:3]*3==i):
                prefix.append(i[:3])
            else:
                prefix.append(i)
        else:
            if(i[1:]+i[:1]==i):
                prefix.append(i[:1])
            else:
                prefix.append(i)
    return prefix

strng = raw_input("String: ")
length = input("Length: ")
```

```
generateStrings(strng, length)
strings = necklaces(strings)
strings = aperiodic(strings)
strings = "".join(strings)
print strings
```

5.2 Duval algorithm implementation:

Implemented two variations of this algorithm. One version, given $s = 3$ and $n = 2$ produces the following universal cycle.

$$[1, 1, 2, 1, 3, 2, 2, 3, 3] \tag{1}$$

And the other implementation produces following universal cycle.

$$[0, 0, 1, 0, 2, 1, 1, 2, 2] \tag{2}$$

Notice the change in the range of symbols. Former implementation considers s = {1, 2, 3} and latter implementation considers s = {0, 1, 2}.

Code

```
def lyndonWords(s,n):
    strings = [0] #strings = [-1]
    while strings:
        strings[-1] += 1
        yield strings
        los = len(strings)
        while len(strings) < n:
            strings.append(strings[-los])
        while strings and strings[-1] == s:
        #while strings and strings[-1] == s - 1:
            strings.pop()

def uCycle(s,n):
    cycle = []
    for c in lyndonWords(s,n):
        if n % len(c) == 0:
            cycle += c
    return cycle
```

Universal cycle in list 1 is generated using this particular implementation. Replacing the commented code with the respective lines produces the universal cycle list 2.

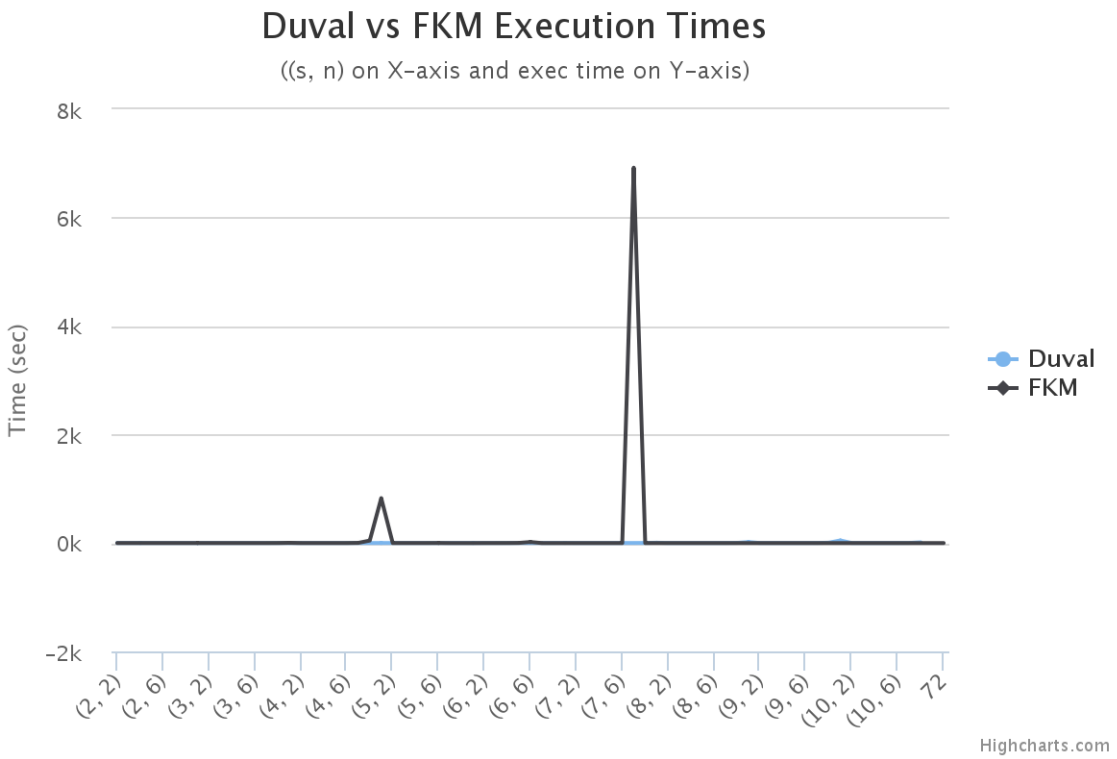

Figure 1: Duval vs FKM

6 Results

The implemented code works and they do generate the universal cycles as intended with drawbacks in each algorithm. Duval's algorithm is supposed to yield an overall running time of $\mathcal{O}(nN(n))$ [8], where N(n) stands for the number of necklaces over n. Time complexity of my implementation seems to complement the theory but the space complexity is really high.

To generate a universal cycle for given $s = 9, n = 9$, the program used a total of 6.382 GB space. But, it only took ~46 seconds. On the other hand, FKM algorithm consumes negligible space when it is compared to the Duval algorithm, but the algorithm is extremely slow. Given $s = 7, n = 7$, it took my FKM program over 6922 secs and Duval only 0.138 secs. After doing further research, I came to conclude that my FKM implementation isn't efficient. I found pseudo code of FKM algorithm [9] which when implemented might be even more efficient. Based on my sample runs on OSC Supercomputing Oakley Cluster, I generated graphs in figure 1, 2 and 3.

Interactive graphs of figure 1, 2 and 3 can be viewed by clicking here.

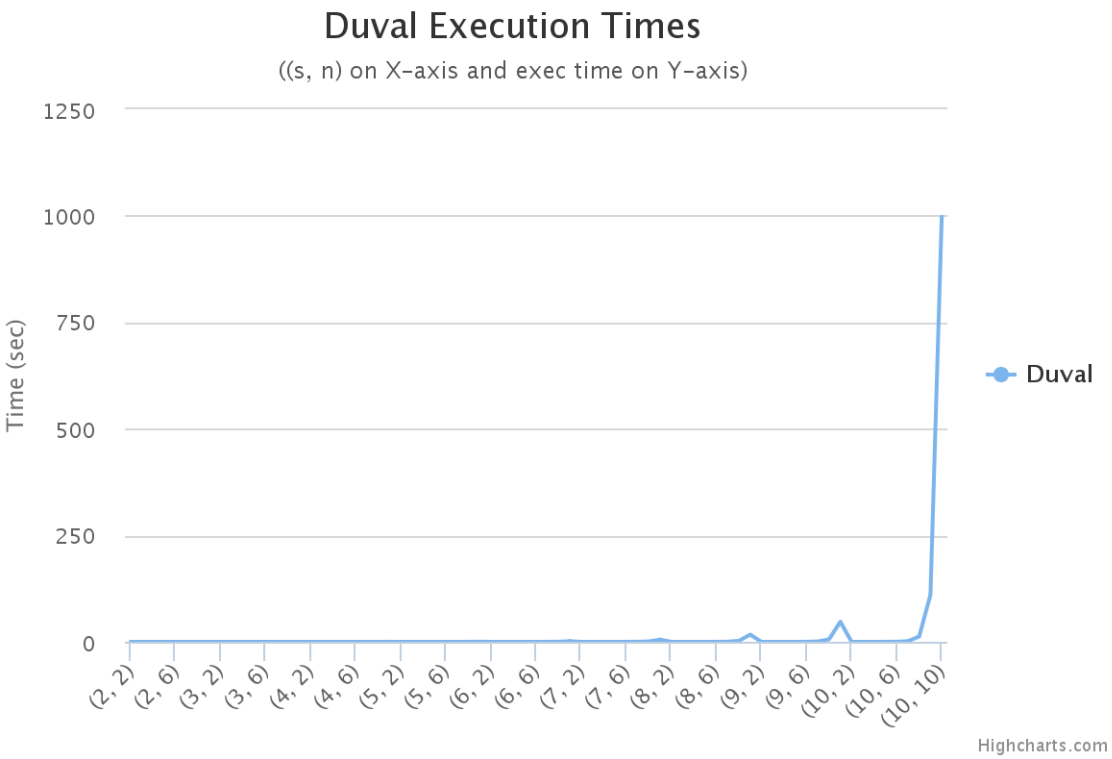

Figure 2: Duval Algorithm

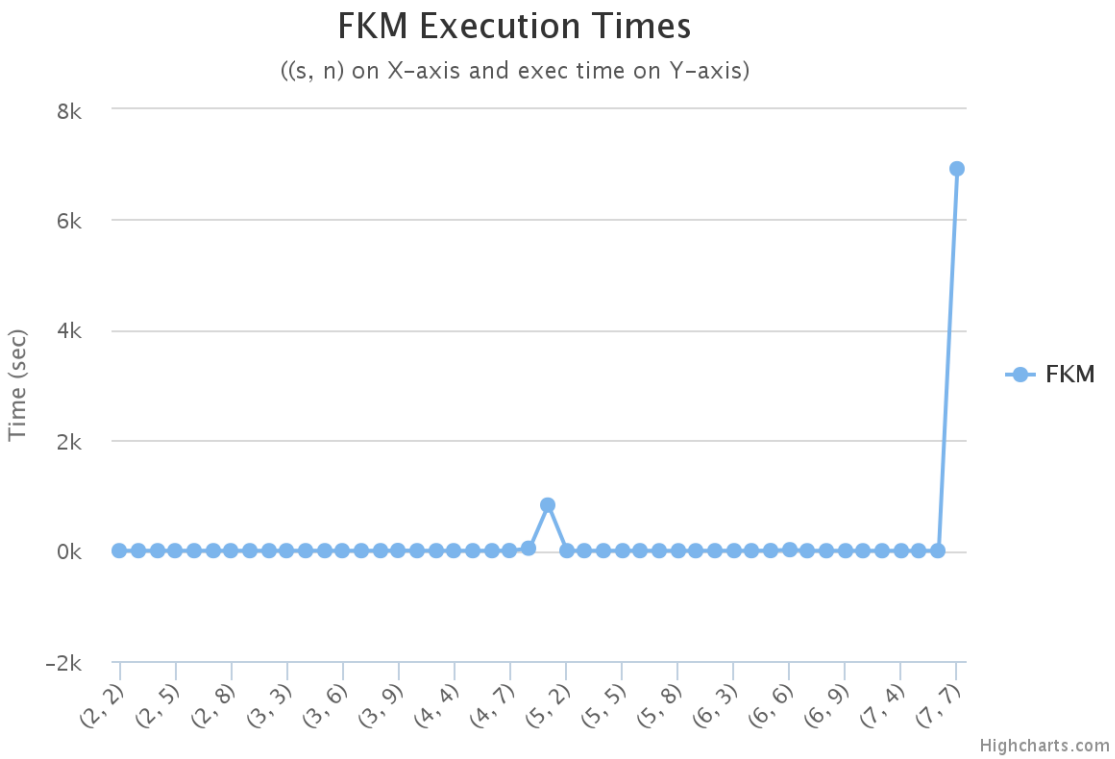

Figure 3: FKM Algorithm

7 Conclusion and Future work

Working on this project has helped in broadening my horizon and also gave me a research direction for my thesis work. Looking forward to further modify the implementations such that they are both time and space efficient. Also came across some interesting papers. Dong and Pei [10] have presented an algorithm for constructing de Bruijn sequences with large orders (such as $n = 128$). Also, there is another interesting paper by Diaconis and Graham [11] where in they show how to construct a **product cycle** of two universal cycles.

References

[1] H. Fredricksen and I. J. Kessler, *An algorithm for generating necklaces of beads in two colors*, Discrete Mathematics 61 (1986), 181-188.

[2] H. Fredricksen and J. Maiorana, *Necklaces of beads in k colors and k-ary de Bruijn sequences*, Discrete Mathematics 23 (1978), 207-210.

[3] J. P. Duval, *Generation d'une section des classes de conjugaison et arbre des mots de Lyndon de longueur bornee*, Theoretical Computer Science archive, Volume 60 Issue 3, September 1988 Pages 255 - 283.

[4] Joe Sawada, Aaron Williams and Dennis Wong, *Generalizing the Classic Greedy and Necklace Constructions of de Bruijn Sequences and Universal Cycles*, The Electronic Journal of Combinatorics 23(1) (2016), #P1.24.

[5] M. H. Martin, *A problem in arrangements*, Bulletin of the American Mathematical Society, 40:859?864, 1934.

[6] Chung, F., P. Diaconis and R. Graham, *Universal cycles for combinatorial structures*, Discrete Mathematics 110 (1992) 43-59.

[7] Marc Chemillier, *Periodic musical sequences and Lyndon words*, Soft Computing, 8:611?616, 2004.

[8] Berstel, J. and M. Pocchiola, *Average cost of Duval?s algorithm for generating Lyndon words*, Theoretical Computer Science 132 (1994) 415-425.

[9] F. Ruskey, C. D. Savage, and T. Wang, *Generating necklaces*, J. Algorithms, 13 (1992), pp. 414?430

[10] J. Dong and D. Pei, *Construction for de Bruijn sequences with large orders*, Cryptology ePrint Archive: Report 2015/1091.

[11] P. Diaconis and R. L. Graham, *Products of universal cycles*, A Lifetime of Puzzles, E. Demaine, M. Demaine and Tom Rodgers, eds., A K Peters, 2008, Boston, MA pp. 35?55.