A Binary de Bruijn Sequence Generator from Zech's Logarithms

Martianus Frederic Ezerman and Adamas Aqsa Fahreza

Abstract

This is a basic software implementation to generate binary de Bruijn sequences based on Zech's Logarithms.

I. Introduction

Technical Reference

We provide basic implementation of the procedures proposed in the preprint at https://arxiv.org/pdf/1705.03150.pdf

Z. Chang, M. F. Ezerman, A. A. Fahreza, S. Ling, J. Szmidt, and H. Wang,
"Binary de Bruijn Sequences via Zech's Logarithms."

We retain the definitions and notations in the paper and recommend that users cite it and provide a link to the source code.

License and Proper Attribution

II. Ingredients and Procedures for Cycle Joining

The implementation of the cycle joining method comes in two flavours. The first is for those who care more about speed of generation and need only a relatively small numbers of de Bruijn sequences. Here the input polynomial is limited to *trinomials*. This limitation can be easily removed by those well-versed in `python`. The second strives towards completeness. It can produce **all** de Bruijn sequences that can be generated from the given input polynomial and stipulated number t of cycles. It comes with two basic options. Users can specify the primitive polynomial $p(x)$ and a valid t. Otherwise, users simply input the desired order n. Here the software needs to build a Zech's Logarithm table for elements in the complete set of coset representatives. As n grows larger, say beyond 20, the waiting time can be impractical.

The source code files can be downloaded directly from Github at https://github.com/adamasstokhorst/ZechdB.

Ingredients common to both flavours are

1) A working `Python 2.7` equipped with `sympy`.
2) MAGMA computer algebra, either locally or through internet connection, to compute the Zech's logarithms.
   This implementation assumes no subscription to any proprietory computer algebra tools. It requests Zech's logarithmic values from the online MAGMA calculator http://magma.maths.usyd.edu.au/calc/. If one has access to MAGMA locally, simple modifications on how the required table is produced and stored can greatly enhance the speed at the expense of some storage space.

All examples below are run in the interactive environment `ipython`.

A. Flavour One

Additional ingredients to have are

1) The main file `zech_one.py`.
2) Two auxiliary files, namely `debruijn.py` and `mathhelper.py`.
3) If graph visualization is preferred, then `matplotlib` and `networkx` packages must be installed. This is optional.

Example 1. *The command*

```
import zech_one as z
z.get_db_seq([15,3,0],31,5,print_matrix=True)
```

outputs the compressed subadjacency graph $\widetilde{G}$ (where edges connecting any pair of vertices are compressed into one edge) on input primitive polynomial $x^{15}+x^3+1$ and $t=31$. Letting `print_matrix=False` *turns off the routine that outputs*

Research Grant TL-9014101684-01 support the authors' work.

M. F. Ezerman and A. A. Fahreza are with the School of Physical and Mathematical Sciences, Nanyang Technological University, 21 Nanyang Link, Singapore 637371, e-mails: {fredezerman,adamas}@ntu.edu.sg.

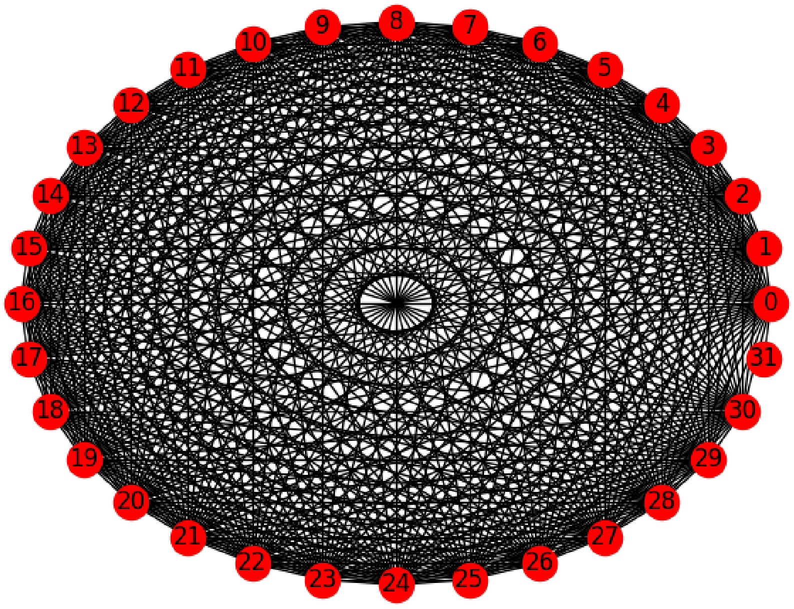

Fig. 1. Compressed subadjacency graph $\widetilde{G}$ for $f(x) = x^{15} + x^3 + 1$ with $t = 31$.

$\widetilde{G}$. The program generates the requested number, here it is 5, of sequences. The initial state of the sequences is $\mathbf{0}$, which can be tweaked easily. The program randomly selects one spanning tree Υ in $\widetilde{G}$ and outputs its edge list. It then randomly selects conjugate pairs, one each per edge in Υ, and use them to join the corresponding cycles, forming a de Bruijn sequence. Using the same spanning tree, it repeats the process for different choices of conjugate pairs for the respective edges until the required number of sequences are built. If this number is larger than the total number of sequences that can be built from Υ, the program chooses another spanning tree and repeat the procedure. In total, the number of sequences that can be produced based on $\widetilde{G}$ above is $\approx 2^{289.69}$.

B. Flavour Two

The main file is `zech_two`. If the number of sequences to generate is not specified, then all of them are explicitly built. Users may supply a specific primitive polynomial, a valid t, and an initial state (otherwise, the default is $\mathbf{0}$).

Example 2. *Supply the command*

```
import zech_two as z
import sympy
from sympy import Poly
from sympy.abc import x
for s in z.get_db_seq(p=Poly(x**10+x**3+1, x), t=31, num_seq=2, i_state='1'*10):
print s
```

to print out 2 de Bruijn sequences with initial states the all-one string $\mathbf{1}$ of length 10 based on primitive polynomial $x^{10}+x^3+1$ using $t = 31$. If an invalid value, say $t = 33$, is given, then the program will flag down the error and quit.

Users may supply only the order or the span n, the number of sequences to generate, and an initial state. This state is otherwise a random n-string. The program asks MAGMA for a random primitive polynomial $p(x)$ and then selects $t \mid (2^n - 1)$. This t is tested for suitability. Repeat until one such t is obtained. Now that $p(x)$ and t are available, the program continues as in the above case.

Example 3. *In interactive environment, run the command*

```
import zech_two as z
for s in z.get_db_seq(n=10, num_seq=4, i_state='1001001001'):
print s
```

to generate 4 de Bruijn sequences of order 10, each with initial state 1001001001.

III. Ingredients and Procedures for Cross Joining

The routine `zech_crossjoin_sample.py` executes the following steps for any order n.

1) Choose a primitive polynomial $p(x)$ of degree n and generate the m-sequence $\mathbf{m}$ from the initial state $1, \mathbf{0}^{n-1}$.
2) Build the Zech's logarithm table based on $p(x)$. Note that in this simple implementation we use the approach suggested in Remark 3 of the technical paper. For larger values of n it is more efficient to obtain the logarithms from MAGMA or its alternatives.
3) Pick random distinct integers a and b satisfying $a < b < \tau(a) < \tau(b)$.
4) Construct the matrix A_p from Equation 3 of the technical paper and use the square-and-multiply technique based on the binary representations of a and b to quickly find
$$\boldsymbol{\alpha} = (1,0,0,\ldots,0)\ A_p^a \text{ and } \boldsymbol{\beta} = (1,0,0,\ldots,0)\ A_p^b.$$
5) The resulting feedback function of the modified de Bruijn sequence follows from applying
$$\widehat{h}(x_0,\ldots,x_{n-1}) = h(x_0,x_1,\ldots,x_{n-1}) + \prod_{i=1}^{n-1}(x_i + a_i + 1) + \prod_{i=1}^{n-1}(x_i + b_i + 1) \tag{1}$$
using $\boldsymbol{\alpha}$ and $\boldsymbol{\beta}$, with h derived from $p(x)$.

Here is a small instance to illustrate the process.

Example 4. *On input $n = 5$, the routine selected $p(x) = x^5 + x^2 + 1$. The chosen pair was $(a,b) = (3,17)$ with $(\tau(a),\tau(b)) = (22,25)$. Hence, $\boldsymbol{\alpha} = (0,1,0,1,1)$ and $\boldsymbol{\beta} = (0,1,1,0,1)$. The feedback function of the resulting modified de Bruijn sequence $(10000\ 10010\ 11101\ 10011\ 11100\ 01101\ 0)$ is $\widehat{h} = x_0 + x_1 \cdot x_2 \cdot x_4 + x_1 \cdot x_3 \cdot x_4 + x_2$.* □