SOFTWARE FOR "MULTI DE BRUIJN SEQUENCES"

GLENN TESLER
gptesler@math.ucsd.edu
http://math.ucsd.edu/∼gptesler/multidebruijn

Last updated: July 16, 2017

This describes Maple software implementing formulas and algorithms in

Glenn Tesler (2017), *Multi de Bruijn Sequences*, Journal of Combinatorics, 8(3):439-474.

The software has been tested in Maple version 18. Load it with

```
read ‘multidebruijn.maple‘;
```

1. Multi de Bruijn sequences of various types

Definition	Notation
Alphabet	Ω
Alphabet size	$q = \|\Omega\|$
Word size	k
Multiplicity of each k-mer	m
Rotational order of a sequence	d; must divide into m
Specific k-mer that sequences start with	y

Number of multi de Bruijn sequences of each type. In these functions that give set sizes, "`_y`" is in some function names but the functions do not take a parameter y because they're the same value for all valid y.

Multi de Bruijn Sequence Type	Set	Function giving size of set
Linear	$\mathcal{L}(m,q,k)$	`Blinear(m,q,k)`
Cyclic	$\mathcal{C}(m,q,k)$	`Bcyclic(m,q,k)`
Linearized cyclic	$\mathcal{LC}(m,q,k)$	`Blinearized_all(m,q,k)`
Multicyclic	$\mathcal{MC}(m,q,k)$	`Bmulticyclic(m,q,k)`
Linear, starts with k-mer y	$\mathcal{L}_y(m,q,k)$	`Blinear_y(m,q,k)`
Linearized cyclic, starts with y	$\mathcal{LC}_y(m,q,k)$	`Blinearized_y(m,q,k)`
Cyclic, order 1	$\mathcal{C}^{(1)}(m,q,k)$	`Bcyclic_d1(m,q,k)`
Cyclic, order d	$\mathcal{C}^{(d)}(m,q,k)$	`Bcyclic_d(m,q,k,d)`
Linearized cyclic, starts with y, order d	$\mathcal{LC}_y^{(d)}(m,q,k)$	`Blinearized_y_d(m,q,k,d)`

Generate random multi de Bruijn sequences of each type.

`alphabet_type=false`: $\Omega = \{0, 1, \ldots, q-1\}$ (requires $1 \leq q \leq 10$)
`alpahbet_type=true`: Ω is the first q letters of $a, b, c, \ldots, z$ (requires $1 \leq q \leq 26$)

Multi de Bruijn Sequence Type	Set	Function giving uniform random element
Linear	$\mathcal{L}(m,q,k)$	`random_linear_mdb(m,q,k,alphabet_type)`
Cyclic	$\mathcal{C}(m,q,k)$	`random_cyclic_mdb(m,q,k,alphabet_type)`
Linearized cyclic, starts with 0^k	$\mathcal{LC}_{0^k}(m,q,k)$	`random_linearized_mdb(m,q,k,alphabet_type)`
Multicyclic	$\mathcal{MC}(m,q,k)$	`random_multicyclic_mdb(m,q,k,alphabet_type)`

`random_linearized_mdb` takes additional optional parameters:

- `root=r`: Start with k-mer $r \in [0, q^k - 1]$ (r is an integer, representing its k digit base q expansion)
- `linear=false` (default): Linearized sequence; the final $(k-1)$-mer is deleted.
  `linear=true`: Linear sequence. The final $(k-1)$-mer is retained. It should match the initial $(k-1)$-mer.

Generate all multi de Bruijn sequences of certain types, for small m, q, k.

- `list_mdb_brute_force(m,q,k)`
  Brute force search to find and count elements of $\mathcal{LC}_{0^k}(m,q,k)$ and $\mathcal{C}(m,q,k)$.
  Prints a list of all elements of $\mathcal{LC}_{0^k}(m,q,k)$.
  The elements that are lexicographically least among their rotations are marked with "`*`". Such elements can be used to represent $\mathcal{C}(m,q,k)$.

  Return value: $[n_1, n_2]$, where
  $n_1 = |\mathcal{LC}_{0^k}(m,q,k)|$ = number of linearized multi de Bruijn sequences starting with 0^k
  $n_2 = |\mathcal{C}(m,q,k)|$ = number of cyclic multi de Bruijn sequences

- `list_mdb_brute_force(m,q,k,return_seqs=true)`
  Brute force search to find elements of $\mathcal{C}(m,q,k)$.
  Does not print a report.

  Return value: list of strings that generate $\mathcal{C}(m,q,k)$. For each element of $\mathcal{C}(m,q,k)$, its lexicographically least rotation is output.

- `list_mdb_brute_force(...,prune_level=...)`
  Various methods to prune the search space. See the paper for details.
  `prune_level=0`: Do not prune search space
  `prune_level=1`: Prune based on k-mer counts (default)
  `prune_level=2`: Prune based on k-mer counts and also $1, 2, \ldots, (k-1)$-mer counts

- `list_all_multicyclic_mdb(m,q,k)` generates $\mathcal{MC}(m,q,k)$.
  Returns a two-part list: $\big[\ [\sigma_1, \sigma_2, \ldots],\ [w_1, w_2, \ldots]\ \big]$
  σ_i represent elements of $\mathcal{MC}(m,q,k)$ and $w_i = \mathrm{EBWT}(\sigma_i)$
  σ_i is represented as a list of strings, $[s_1, s_2, \ldots, s_r]$, corresponding to $(s_1)(s_2)\ldots(s_r)$

Functions on strings.

Definition	Notation	Function
Rotate string s by i characters to the right	$\rho^i(s)$	`rotate_string(s,i)`
Primitive root of string s	$\textsc{Root}(s)$	`word_root(s)`
Power of string	$s^i = s \cdots s$ (i times)	`word_power(s,i)`
Lexicographically least rotation of s		`word_canon(s)`

Functions on numeric representations of strings.

x is a number in base q with k digits, representing a string s:

$$x = \sum_{i=0}^{k-1} a_i q^i \qquad \text{represents string } s = \text{“}a_{k-1} \dots a_1 a_0\text{”} \text{ with } 0 \le a_i \le q-1.$$

`alphabet`=“$c_0 c_1 \dots c_{q-1}$” is a string giving the alphabet:

Digit $d \in \{0, \dots, q-1\}$ in x is represented by character c_d in s.

`alphabet_type=false` corresponds to `alphabet`=“012 … (q-1)”.

`alphabet_type=true` corresponds `alphabet` equalling the first q letters of “abc … z”.

Definition	Notation	Function
Convert x to k-digits base q, as a string	$a_{k-1} \dots a_0$	`int2kmer(x,q,k,alphabet)`
Rotate one position to the right	$\rho(s)$	`cycle_right1(q,k,x)`
Rotate i positions to the left	$\rho^{-i}(s)$	`cycle_left(q,k,i,x)`
Lexicographically least rotation of x using base q with mq^k digits		`calc_min_shift(m,q,k,x)`

Subroutines used in computing random or brute force multi de Bruijn sequences of each type.

- `random_db_tree(q,k,root)`: Use the Kandel et al algorithm to compute a random spanning tree T of $G(1,q,k)$ with specified root vertex `root`.
  There are $n = q^{k-1}$ vertices, numbered $x \in [0, n-1]$.
  Write x in base q as $a_{k-2} \dots a_0$. Then $x = \sum_{i=0}^{k-2} a_i q^i$ represents $(k-1)$-mer $a_{k-2} \dots a_0$.
  `root` $\in [0, n-1]$ is a vertex.

  Output: $[b_0, b_1, \dots, b_{n-1}]$, representing spanning tree T as a list.
  $b_x \in \{0, \dots, q-1\}$ indicates that vertex x has an outgoing edge in T:

  $$x = a_{k-2} \dots a_0 \xrightarrow{a_{k-2} \dots a_0 b_x} a_{k-3} \dots a_0 b_x$$

  Values of b_x for $x \neq$ `root` are chosen by the Kandel et al algorithm, while $b_{\text{root}} = 0$.

- `random_multiperm(m,alpha)`: Uniform random element of $\mathcal{W}_{m,q}$ over the alphabet specified by `alpha`.

  Recall $\mathcal{W}_{m,q}$ is the set of permutations of $0^m 1^m \dots (q-1)^m$, that is, strings of length mq with m copies of each symbol $0, 1, \dots, q-1$. This extends to other alphabets.

  The alphabet can be:
  `alpha` = string: Ω is the set of characters of the string. Set q to the length of the string.
  `alpha` = integer q or `[q,false]`: $\Omega = \{0, 1, \dots, q-1\}$. Requires $1 \leq q \leq 10$.
  `alpha` = `[q,true]`: Ω is the first q letters of abcd…z. Requires $1 \leq q \leq 26$.

- `random_multiperm_constrained(m,alpha,a_last)`: Uniform random element of the subset of $\mathcal{W}_{m,q}$ that ends in the `a_last`$^{\text{th}}$ character of the alphabet `alpha`.
  Note that `a_last` is a number $0, \dots, q-1$ regardless of how `alpha` is specified.

- `generate_mq_multiperms(m,q)`: List all elements of $\mathcal{W}_{m,q}$, using alphabet $0, \dots, q-1$.

- `exittable2string(m, q, k, exit_table, root_v)`
  Generate a string by a walk through the graph $G(m,q,k)$.
  The walk starts at vertex `root_v` $\in [0, q^{k-1} - 1]$.
  Table `exit_table` encodes the function $g(x)$ in the paper, which lists the order to select outgoing edges of x along the walk.

- `string2exittable(m,q,k,s)`
  $s \in \mathcal{LC}(m,q,k)$ is a string representing a linearization of a multi de Bruijn sequence.
  Form a table `exit_table`, representing the function $g(x)$ in the paper.
  The initial vertex is the first $(k-1)$-mer of s, and the initial edge is the first k-mer of s.
  The example in the paper was generated as follows. Since `random_linearized_mdb` is random, results will change each time. In this example, we want a random linearized multi de Bruijn sequence starting with 021, which is the base 3 representation of 7, so set `root` = 7. Note that `>` before the command is Maple's prompt and should not be entered.

```
> random_linearized_mdb(2,3,3,false,root=7);
       "021202210112012122202010211211011100222001000122102 00"

> exit_table := string2exittable(2,3,3,
          "021202210112012122202010211211011100222001000122102 00");
```

```
exit_table := table(
      ["12" = "001212", "21" = "202110", "01" = "120102",
       "02" = "120120", "11" = "221010", "22" = "120201",
       "00" = "210102", "20" = "212100", "10" = "121002"
    ])
```

```
> exittable2string(2,3,3,exit_table,"02");
      "021202210112012122202010211211011100222001000122102000 2"
```

Note that this is a *linear* string rather than a *linearized* string, so the initial $(k-1)$-mer 02 is repeated at the end. Delete it from the end to get the linearized string.

- `test_mdb(m,q,k,n)`: Test if $n \in [0, q^\ell - 1]$ (in base q with $\ell = mq^k$ digits, $n = a_{\ell-1} \dots a_0$) represents a cyclic multi de Bruijn sequence.
  Return value: `[success,i]`
  `success` is `true` if n does represent a cyclic multi de Bruijn sequence, `false` otherwise.
  `i` allows pruning as described in the paper. $a_{i+k-1} \cdots a_i$ is the first interval from the left where a k-mer count exceeds m.

- `test_uniformity(m,q,k,alphabet_type,seq_type,ntrials)`
  `test_uniformity(m,q,k,alphabet_type,seq_type,ntrials,verbose=boolean)`
  Generate `ntrials` random multi de Bruijn sequences of the type specified by `seq_type` and compute some statistics.

`seq_type`	Chooses uniform random element of set
"cyclic"	$\mathcal{C}(m,q,k)$
"linearized"	$\mathcal{LC}_{0^k}(m,q,k)$
"linear"	$\mathcal{L}(m,q,k)$
"multicyclic"	$\mathcal{MC}(m,q,k)$

  If `verbose` is true (default), it prints a report showing how many sequences were generated out of the total number N in the set determined by `seq_type`. Also how many times each sequence was generated, the z scores of these counts, and a χ^2 goodness-of-fit test for whether the distribution is uniform (null hypothesis: all N counts equal `ntrials`$/N$).
  Return value: P-value for the χ^2 goodness-of-fit test.

2. Burrows-Wheeler Transformation and Extended BWT

Definition	Notation	Function
BWT input string	s	
EBWT input: multicycle	$\sigma = (s_1)(s_2)\ldots$	`[s1,s2,...]` (a list of strings)
Represent σ as a string		`ebwt_cycles_to_string([s1,s2,...])`
(E)BWT output string	w	
Burrows-Wheeler Transform	$\mathrm{BWT}(s)$	`bwt_encode(s)`
Inverse BWT	$\mathrm{BWT}^{-1}(w)$	`bwt_decode(w)` (see below)
Extended BWT	$\mathrm{EBWT}(\sigma)$	`ebwt_encode([s1,s2,...])`
Inverse EBWT	$\mathrm{EBWT}^{-1}(w)$	`ebwt_decode(w)`
Forwards BWT table	$T_B(s)$	`bwt_encode_table(s)`
table before sorting		`bwt_encode_table(s,sorted=false)`
Inverse BWT table	$T_{IB}(w)$	`bwt_decode_table(w)`
Forwards EBWT table	$T_E(\sigma)$	`ebwt_encode_table([s1,s2,...])`
table before sorting		`ebwt_encode_table([s1,s2,...], sorted=false)`
Inverse EBWT table	$T_{IE}(w)$	`ebwt_decode_table(w)`
# columns in forwards EBWT table	c	`ebwt_encode_num_columns([s1,s2,...])`
# columns in inverse EBWT table	c	`ebwt_decode_num_columns(w)`
Standard permutation of w		`standard_perm(w)` (see below)

Examples:

```
> bwt_encode("00011101");    # Forward BWT
        "10100110"

> bwt_encode("11101000");    # BWT of a rotation gives the same result
        "10100110"

> bwt_decode("10100110");    # Inverse
        "00011101"             This implementation gives the lex least rotation

> ebwt_encode(["00011101"]); # corresponding computation with EBWT
        "10100110"

> ebwt_decode("10100110");
        ["00011101"]

> bwt_decode("baab");        # Some strings are not invertible under BWT
        undefined

> ebwt_decode("baab");       # All strings are invertible under EBWT
        ["aab", "b"]
```

For $\sigma = (0001)(011)(1)$, the unsorted EBWT table is produced by

```
> ebwt_encode_table(["0001","011","1"],sorted=false);
        ["000100010001", "100010001000", "010001000100", "001000100010",
         "011011011011", "101101101101", "110110110110", "111111111111"]
```

and the complete EBWT table $T_E(\sigma)$ is produced by

```
> ebwt_encode_table(["0001","011","1"]);
        ["000100010001", "001000100010", "010001000100", "011011011011",
         "100010001000", "101101101101", "110110110110", "111111111111"]
```

Compute the transform:

```
> ebwt_encode(["0001","011","1"]);
        "10010101"
```

Compute the inverse:

```
> ebwt_decode("10010101");
        ["0001", "011", "1"]
```

`standard_perm(w)` computes the standard permutation of string w, and related quantities:

- Returns a list of four values, `[perm, invperm, cperm, cperm_words]`.
- `perm` is the "standard permutation" of Gessel and Reutenauer in "1-line form" as a list of numbers $0, \ldots, n-1$ in some permuted order.
- `invperm` is the inverse permutation in the same format.
- `cperm` is the cycle form of `perm`, represented as a list of lists.
- `cperm_words` is the strings corresponding to each cycle of `cperm` under the Gessel-Reutenauer bijection.

The example in the paper:

```
> standard_perm("10010101");
        [[1, 2, 4, 6, 0, 3, 5, 7],
         [4, 0, 1, 5, 2, 6, 3, 7],
         [[0, 1, 2, 4], [3, 6, 5], [7]],
         ["0001", "011", "1"]]
```

This represents:

$$\text{standard permutation in 2-line form} \quad \pi = \begin{pmatrix} 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 \\ 1 & 2 & 4 & 6 & 0 & 3 & 5 & 7 \end{pmatrix}$$

$$\text{inverse in 2-line form} \quad \pi^{-1} = \begin{pmatrix} 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 \\ 4 & 0 & 1 & 5 & 2 & 6 & 3 & 7 \end{pmatrix}$$

$$\text{standard permutation in cycle form} \quad \pi = (0,1,2,4)(3,6,5)(7)$$

$$\text{Gessel-Reutenauer bijection maps } w \text{ to this element of } \mathcal{M} \quad \sigma = (0,0,0,1)(0,1,1)(1)$$