Generating combinations by prefix shifts

Frank Ruskey[*] Aaron Williams[†]

Extended Abstract for COCOON 2005

Abstract

We present a new Gray code for combinations that is practical and elegant. We represent combinations as bitstrings with s 0's and t 1's, and generate them with a remarkably simple rule: Identify the shortest prefix ending in 010 or 011 (or the entire bitstring if no such prefix exists) and then rotate (shift) it by one position to the right. Since the rotated portion of the string consists of at most four contiguous runs of 0's and 1's, each successive combination can be generated by transposing only one or two pairs of bits. This leads to a very efficient loopless implementation. The Gray code also has a simple and efficient ranking algorithm that closely resembles that of combinations in colex order. For this reason, we have given a nickname to our order: cool-lex!

1 Background and motivation

An important class of computational tasks is the listing of fundamental combinatorial structures such as permutations, combinations, trees, and so on. Regarding combinations, Donald E. Knuth [8] writes "Even the apparently lowly topic of combination generation turns out to be surprisingly rich, ... I strongly believe in building up a firm foundation, so I have discussed this topic much more thoroughly than I will be able to do with material that is newer or less basic."

The applications of combination generation are numerous and varied, and Gray codes for them are particularly valuable. We mention as application areas cryptography (where they have been implemented in hardware at NSA), genetic algorithms, software and hardware

[*]Dept. of Computer Science, University of Victoria, research supported in part by NSERC. `e-mail: fruskey@cs.uvic.ca`

[†]Dept. of Computer Science, University of Victoria, research supported in part by a NSERC PGS-D. `e-mail: haron@cs.uvic.ca`

testing, statistical computation (e.g., for the bootstrap, Diaconis and Holmes [3]), and, of course, exhaustive combinatorial searches.

As is common, combinations are represented as bitstrings of length $n = s + t$ containing s 0's and t 1's. We denote this set as $\mathbf{B}(s,t) = \{b_1 b_2 \cdots b_n \mid \sum b_i = t\}$. Another way of representing combinations is as increasing sequences of the elements in the combination. We denote this set as $\mathbf{C}(s,t) = \{c_1 c_2 \cdots c_t \mid 1 \le c_1 < c_2 < \cdots < c_t \le s + t\}$.

We consider here the problem of listing the elements of $\mathbf{B}(s,t)$ so that successive bitstrings differ by a prefix that is cyclically shifted by one position to the right. We call such shifts prefix shifts, or rotations, and they may be represented by a cyclic permutation $\sigma_k = (1\ 2\ \cdots\ k)$ for some $2 \le k \le n$, where this permutation acts on the indices of a bitstring.

As far as we are aware, the only other class of strings that has a listing by prefix shifts are permutations, say of $\{1, 2, \ldots, n\}$. In Corbett [1] and Jiang and Ruskey [7] it is shown that all permutations may be listed by prefix shifts. In other words, the directed Cayley graph with generators $(1\ 2), (1\ 2\ 3), (1\ 2\ \cdots\ n)$ is Hamiltonian. In our case we have the same set of generators acting on the indices of the bitstring, but the underlying graph is not vertex-transitive; in fact, it is not regular.

There are many algorithms for generating combinations. The one presented here has the following novel characteristics.

1. Successive combinations differ by a prefix shift. There is no other algorithm for generating combinations with this feature. In some applications combinations are represented in a single computer word; our algorithm is very fast in this scenario. It is also very suitable for hardware implementation.

2. Successive combinations differ by one or two transpositions of a 0 and a 1. There are other algorithms where successive combinations differ by a single transposition (Tang and Liu [10]). Furthermore, that transposition can be further restricted in various ways. For example, so that only 0's are between the transposed bits (Eades and McKay) [5], or such that that the transposed bits are adjacent or have only one bit between (Chase [2]). Along with ours, these other variants are ably discussed in Knuth [8].

3. The list is circular; the first and last bitstrings differ by a prefix shift.

4. The list for (s,t) begins with the list for $(s-1,t)$. Usually, this property is incompatible with Property 3, relative to the elementary operation used to transform one string to the next. For example, colex order has Property 4 but not Property 3.

5. The algorithm can be implemented so that *in the worst case* only a small number of operations are done between successive combinations, *independent* of s and t. Such algorithms are said to be loopless, an expression coined by Ehrlich [6].

110000	111100	111000	123	σ_4
011000	011110	011100	234	σ_2
101000	101110	101100	134	σ_3
010100	110110	110100	124	σ_5
001100	111010	011010	235	σ_4
100100	011101	101010	135	σ_4
010010	101101	010110	245	σ_3
001010	110101	001110	345	σ_3
000110	011011	100110	145	σ_4
100010	101011	110010	125	σ_6
010001	010111	011001	236	σ_2
001001	001111	101001	136	σ_4
000101	100111	010101	246	σ_3
000011	110011	001101	346	σ_3
100001	111001	100101	146	σ_5
		010011	256	σ_3
		001011	356	σ_4
		000111	456	σ_4
		100011	156	σ_5
		110001	126	σ_6

Figure 1: Cool-lex listings $\mathbf{W}'_{42}$, $\mathbf{W}'_{24}$, $\mathbf{W}'_{33}$.

6. Unlike other Gray codes for combinations, this one has a simple ranking function whose running time is $O(n)$ arithmetic operations.

2 Recursive Construction Rule

If $S = s_1, s_2, \ldots, s_m$ is a sequence of strings and x is a symbol, then Sx represents the sequence of strings $Sx = s_1x, s_2x, \ldots, s_mx$. We recursively define the following list of bitstrings.

$$\mathbf{W}_{st} = \mathbf{W}_{(s-1)t}0,\ \mathbf{W}_{s(t-1)}1,\ 1^{t-1}0^s1 \tag{1}$$

As will be proven below this list accounts for all strings in $\mathbf{B}(s,t)$ except for 1^t0^s. To get all of $\mathbf{B}(s,t)$ we define

$$\mathbf{W}'_{st} = 1^t0^s, \mathbf{W}_{st}. \tag{2}$$

Examples of $\mathbf{W}'$ may be found in Figure 1 (the additional columns for $\mathbf{W}'_{33}$ give the corresponding element of $\mathbf{C}(3,3)$ and the rotation σ_k used in transforming one bitstring to the next).

Theorem 2.1. *The list $\mathbf{W}_{st}$ defined in (1) has the following properties.*

- *The list contains each bitstring of $\mathbf{B}(s,t)$ exactly once, except for 1^t0^s.*
- *Successive bitstrings differ by a prefix shift of one position to the right.*
- *Successive bitstrings differ by the tranposition of one or two pairs of bits.*
- $first(\mathbf{W}_{st}) = 01^t0^{s-1}$.
- $last(\mathbf{W}_{st}) = 1^{t-1}0^s1$.

Proof. Our proof is by induction on $n = s+t$. The first property is satisfied since, inductively, $\mathbf{W}_{(s-1)t}0$ is a list of all elements of $\mathbf{B}(s,t)$ that end in a 0, except for $1^t0^{s-1}0$, and $\mathbf{W}_{s(t-1)}1$ is a list of all elements of $\mathbf{B}(s,t)$ that end in a 1, except for $1^{t-1}0^t1$, which is appended.

To prove the remaining properties it is convenient to separate out the cases $t = 1$ and $s = 1$. The interfaces between sublists are indicated below as horizontal lines, and transposed bits are underlined. The list below on the left is for $t = 1$ and on the right for $s = 1$.

$$
\begin{array}{c}
010^{s-2}\ 0 \\
\vdots \\
0^{s-2}0\underline{1}\ \underline{0} \\
\hline\hline
00^{s-2}0\ 1
\end{array}
\qquad\qquad
\begin{array}{c}
\hline
011^{t-2}\ 1 \\
\vdots \\
1^{t-2}\underline{01}\ 1 \\
\hline
11^{t-2}0\ 1
\end{array}
$$

Below we show the lists for the case where $t > 1$ and $s > 1$. The left and right lists are identical, except that the left list illustrates shifts, while the right list illustrates transpositions. The starting and ending bitstring in each sublist is obtained from the induction assumption.

$$
\begin{array}{c}
011^{t-2}10^{s-2}\ 0 \\
\vdots \\
11^{t-2}00^{s-2}1\ 0 \\
\hline
011^{t-2}00^{s-2}\ 1 \\
\vdots \\
1^{t-2}00^{s-2}01\ 1 \\
\hline
11^{t-2}00^{s-2}0\ 1
\end{array}
\qquad\qquad
\begin{array}{c}
011^{t-2}10^{s-2}\ 0 \\
\vdots \\
\underline{1}1^{t-2}\underline{0}0^{s-2}\underline{1}\ \underline{0} \\
\hline
01^{t-2}10^{s-2}0\ 1 \\
\vdots \\
1^{t-2}\underline{0}00^{s-2}\underline{1}\ 1 \\
\hline
1^{t-2}100^{s-2}0\ 1
\end{array}
$$

To verify the second and third properties we need to examine what happens at the interfaces between the lists in (1) as illustrated above. Note that at the two interfaces the successive bitstrings differ by a right rotation of all n positions (although at the second interface we could also think of it as a rotation of the first $n-1$ positions).

Note that two pairs of bits are transposed at the first interface, and one pair of bits at the second interface. Thus the third property is satisfied. Finally, observe that the first and last bitstrings in these lists have the required form. □

To close this section, we make the interesting observation that $last(W_{st}) = \overline{first(W_{ts})}^R$; such equations would allow us to define cool-lex order in other ways.

3 Implementation

Referring back to the proof of Theorem 2.1, we observe that the bits that are transposed at the first interface are $(1, t)$ and $(n-1, n)$, and at the second interface $(t-1, n-1)$. Below we show a recursive implementation of the algorithm; this is followed by an iterative implementation. In both cases, the code that initializes `b` to $1^t 0^s$ and outputs it is omitted; we also assume that $s > 0$ and $t > 0$.

For the recursive version, the array `b` has indexing starting at 1. The initial call is `swap( 1, t+1 ); visit( b ); gen( s, t );`. Since every recursive call is followed by a visit, the algorithm runs in constant amortized time.

```
static void gen ( int s, int t ) {
     if (s > 1) { gen( s-1, t );
         swap( 1, t );  swap( s+t, s+t-1 ); visit( b ); }
     if (t > 1) { gen( s, t-1 );
         swap( t-1, s+t-1 );  visit( b ); }
}
```

We now present the iterative loopless implementation. In this case the array indexing is 0 based. It is useful to maintain a variable x, which is the smallest index for which `b[x-1] == 0` and `b[x] == 1`. In terms of shifts, the code to get the next bitstring and to update `x` is amazingly simple.

```
shift( ++x );
if (b[0] == 0 && b[1] == 1) x = 1;
```

To generate the next bitstring by transpositions it is useful to maintain another variable y, which is the smallest index for which `b[y] == 0`. Referring back to the proof of Theorem 2.1 we observe that in every case `b[x]` becomes `0` and `b[y]` becomes `1`. The test `b[x+1] == 0` determines whether we are at the first or the second interface. If we are at the first interface, then we set `b[x+1]` to `1` and `b[0]` to `0`. It now remains to update x and y. At the second interface they are simply incremented. At the first interface y always becomes

0; also, x is incremented unless $y = 0$, in which case x becomes 1 (see the $t = 1$ case of the proof of Theorem 2.1).

```
static void iterate ( int s, int t ) {
   b[t] = 1;  b[0] = 0;
   visit( b );
   int x = 1, y = 0;
   while (x < n-1) {
      b[x++] = 0;  b[y++] = 1;     /* X(s,t) */
      if (b[x] == 0) {
         b[x] = 1;  b[0] = 0;      /* Y(s,t) */
         if (y > 1) x = 1;         /* Z(s,t) */
         y = 0; }
      visit( b ); } }
```

The structure of the implementation allows us to completely determine the number of times each statement in the code is executed. Call the relevant quantities $X(s,t)$, $Y(s,t)$, and $Z(s,t)$ corresponding to the various statements as shown above. I.e., $Y(s,t)$ is the number of times `b[x] == 0` is true and $X(s,t)$ is the number of times `y > 1` is true. We find that

$$X(s,t) = \binom{s+t}{t} - 1, \quad Y(s,t) = \binom{s+t-1}{t}, \quad Z(s,t) = \binom{s+t-2}{t-1}.$$

4 Ranking Algorithm

Given a listing of combinatorial structures, the *rank* of a particular structure is the number of structures that precede it in the listing.

Colex order is lexicographic order applied to the reversal of strings. It has many uses, for example in Frankl's now standard proof of the Kruskal-Katona Theorem [11]. Given an (s,t)-combination represented as a bitstring $b_1 b_2 \cdots b_n$ the corresponding set elements can be listed as $c_1 < c_2 < \cdots < c_t$ where c_i is the position of the i-th 1 in the bitstring. As is well-known ([8],[11]) in colex order the rank of $c_1 c_2 \cdots c_t$ is

$$\sum_{j=1}^{t} \binom{c_i - 1}{i}. \tag{3}$$

As we see in the statement of the theorem below, in cool-lex order there is a very similar rank function. Let $rank(c_1 c_2 \cdots c_t)$ denote the rank of $c_1 c_2 \cdots c_t \in \mathbf{C}(s,t)$ in our order.

Theorem 4.1. *Let r be the smallest index such that $c_r > r$ (so that $c_{r-1} = r-1$).*

$$rank(c_1 c_2 \cdots c_t) = \binom{c_r}{r} - 1 + \sum_{j=r+1}^{t} \left(\binom{c_j - 1}{j} - 1 \right), \tag{4}$$

Proof. Directly from the recursive construction (1) we have

$$rank(b_1b_2\cdots b_n) = \begin{cases} rank(b_1b_2\cdots b_{n-1}) & \text{if } b_n = 0, \\ \binom{s+t}{t} - 1 & \text{if } b_1b_2\cdots b_n = 1^{t-1}0^s1, \\ \binom{s+t-1}{t-1} - 1 + rank(b_1b_2\cdots b_{n-1}) & \text{otherwise.} \end{cases}$$

Let us now consider the rank in terms of the corresponding list of elements $1 \leq c_1 < c_2 < \cdots < c_t$. The case $rank(b_1b_2\cdots b_n) = rank(b_1b_2\cdots b_{n-1}) = rank(b_1b_2\cdots b_{n-2})$ will continue to apply until $b_{n-k} = 1$; i.e., until $n - k = c_{t-1}$. Hence the number of 0's and 1's to the left of position c_{t-1} in $b_1b_2\cdots b_n$ is $c_{t-1} - 1$, which leads us to the expression below.

$$rank(c_1c_2\cdots c_t) = \begin{cases} \binom{c_t}{t} - 1 & \text{if } c_t = n \text{ and } c_{t-1} = t-1 \\ \binom{c_{t-1}-1}{t-1} - 1 + rank(c_1c_2\cdots c_{t-1}) & \text{otherwise.} \end{cases}$$

As in (3) and (4), the recursion above has the remarkable and useful property that it depends only on t and not on s. In other words, the cool-lex lists begin with cool-lex lists with smaller s values (fewer zeroes). $\square$

The ranking function can also be written recursively, as shown below.

$$rank(c_1c_t\cdots c_t) = rank(c_1c_2\cdots c_{t-1}) + \binom{c_r - 1}{r-1} + r - t - 1. \tag{5}$$

Using standard techniques, as explained for example in [8] the expression in (4) can be evaluated in $O(n)$ arithmetic operations.

5 Final Remarks

Unlike every other recursive Gray code definition, (1) has the remarkable property that it involves no reversal of lists. The list for $C(6,3)$ has been rendered musically by George Tzanetakis and is available on the page `http://www.cs.uvic.ca/~ruskey/Publications/Coollex/Coollex.html`

The algorithm discussed here appears in Knuth's prefasicle [8] (latest version of January 19, 2005). The output of the algorithm is illustrated in Figure 26 on page 16. He refers to the listing as suffix-rotated (since he indexes the bitstrings $b_{n-1}\cdots b_1b_0$). See also Exercise 55 on page 30 and it's solution on page 46.

To conclude the paper we list some open problems:

- Is it possible to generate combinations if the allowed operations are futher restricted? For example, all permutations can be generated by letting the permutations $(1\ 2)$ and $(1\ 2\ \cdots\ n)$ and their inverses act on the indices. Can all combinations be so generated?

- Can the permutations of a multiset be generated by suffix rotations?
- What is the fastest combination generator when carefully implemented? It would be interesting to undertake a comparative evaluation in a controlled environment, say of carefully implemented MMIX programs. Testing should be done, in the three cases, depending on whether the combination is represented by a single computer word, an element of $\mathbf{B}(s,t)$, or an element of $\mathbf{C}(s,t)$.

References

[1] P.F. Corbett, *Rotator Graphs: An Efficient Topology for Point-to-Point Multiprocessor Networks*, IEEE Transactions on Parallel and Distributed Systems, 3 (1992) 622–626

[2] P.J. Chase, *Combination Generation and Graylex Ordering*, Congressus Numerantium, 69 (1989) 215-242.

[3] P. Diaconis and S. Holmes, *Gray codes for randomization procedures* Statistical Computing, 4 (1994) 207–302.

[4] P. Eades, M. Hickey and R. Read, *Some Hamilton Paths and a Minimal Change Algorithm*, Journal of the ACM, 31 (1984) 19-29.

[5] P. Eades and B. McKay, *An Algorithm for Generating Subsets of Fixed Size with a Strong Minimal Change Property*, Information Processing Letters, 19 (1984) 131-133.

[6] G. Ehrlich, *Loopless Algorithms for Generating Permutations, Combinations and Other Combinatorial Configurations*, Journal of the ACM, 20 (1973) 500-513.

[7] M. Jiang and F. Ruskey, *Determining the Hamilton-connectedness of certain vertex-transitive graphs*, Discrete Mathematics, 133 (1994) 159-170.

[8] Donald E. Knuth, *The Art of Computer Programming*, pre-fascicle 4A (a draft of Section 7.2.1.3: Generating all Combinations), Addison-Wesley, 2004, 61 pages (available online at `http://www-cs-faculty.stanford.edu/~knuth/fasc3a.ps.gz`).

[9] F. Ruskey, *Simple combinatorial Gray codes constructed by reversing sublists*, 4th ISAAC (International Symposium on Algorithms and Computation), Lecture Notes in Computer Science, #762 (1993) 201–208.

[10] D.T. Tang and C.N. Liu *Distance-2 Cycle Chaining of Constant Weight Codes*, IEEE Transactions, **C-22** (1973) 176–180.

[11] D. Stanton and D. White, *Constructive Combinatorics*, Springer-Verlag, 1986.