On Canonical Gray Cycles

Stanislav Sykora, Extra Byte, www.ebyte.it
First published in February 2014

This article explores cycles of integers sharing the special property of the 'classical' Gray sequence which make the latter so popular in technical applications, namely the fact that the binary expansions of any two adjacent members differ by exactly one digit. However, there are many cycles having this property: given any even number 2n > 0, it is always possible to find at least one permutation of numbers 0 to 2n which has it. Even accounting for symmetry-induced equivalences (such as rotating or reversing the cycle) , there are for many values of n multiple, genuinely distinct Gray cycles of length 2n, some of which with a potential for useful practical applications. This article focuses on counting the equivalence classes of binary Gray cycles, identified with a set of canonical Gray cycles (CGC). Using a brute-force enumeration algorithm, the number of CGC's was computed up to the cycle length of 32.

Keywords: math, sequence, permutation, cycle, code, Gray code, Gray cycle, CGC, algorithm, counting

Introduction

Gray [1] codes / sequences [2,3] are ordered sets of non-negative integer numbers G(i), i = 0, 1, 2, ..., such that:

a) Any two consecutive numbers, when expressed in binary notation, differ by one and only one bit,
b) The complete sequence is a bijection of the set of non-negative integers onto itself (a permutation).

Gray codes are useful, for example, in devices for digital readout of linear displacements. These consist of a series of binary sensors, which scan "strips" of encoding patterns. A practical realization might consist of a number of black & white strips attached to a moving object which are read as patters of 0's and 1's by an array of fixed sensors (such as photocells).

Consider the schematic representation of one such device, realized in a naïve way:

```
         |
Strip0 <<<<<<<<< o _X_X_X_X_X_X_X_X <<<<<<<<<<<<<<<<<<<<<<<<<<<<
Strip1 <<<<<<<<< o __XX__XX__XX__XX <<<<<<<<< Moving <<<<<<<<<<<
Strip2 <<<<<<<<< o ____XXXX____XXXX <<<<<<<<< object <<<<<<<<<<<
Strip3 <<<<<<<<< o ________XXXXXXXX <<<<<<<<<<<<<<<<<<<<<<<<<<<<
         |
         | 0123456789ABCDEF
```

Fig,1: 4-bit position reading device using natural numbering.

The least significant digit is encoded by Strip0, the most significant one by Strip3. The symbols "○" stand for the readout "photocells" which are fixed while the strips move under them from right to left. The position codes (shown in hexadecimal) form just the series of natural numbers from 0 to 15. In case of slight misalignment of the strips, problematic false readings might occur in the passages from 3 to 4, 7 to 8, and B to C:

One problem with such devices is that the mechanical alignment of the strips and sensors is never perfect. Should it happen at any moment during the motion that two or more bits should change simultaneously, such a misalignment would lead to brief false readings which would be difficult to suppress.

The Gray code is designed to eliminate this problem by making sure that there never occur two "simultaneously" changing bits, while maintaining a unique correspondence between "readout codes" and "positions". Fig.2. shows how it works in practice.

```
         |
Strip0 <<<<<<<<< o _XX__XX__XX__XX_ <<<<<<<<<<<<<<<<<<<<<<<<<<<<
Strip1 <<<<<<<<< o __XXXX____XXXX__ <<<<<<<<< Moving <<<<<<<<<<<
Strip2 <<<<<<<<< o ____XXXXXXXX____ <<<<<<<<< object <<<<<<<<<<<
Strip3 <<<<<<<<< o ________XXXXXXXX <<<<<<<<<<<<<<<<<<<<<<<<<<<<
         |
         | 01326754CDFEAB98
```

Fig,2: 4-bit position reading device using Gray codes.

The same thing like in Fig.2., but the 16 position codes were permuted in a way to satisfy the Gray conditions (a) to (c). The position codes are no longer naturally ordered, but they are still unique. Most importantly, no false reading can occur.

The usefulness of Gray codes actually goes well beyond the above example. Frank Gray himself initially introduced the concept in the context of handling TV signals. Since then it has found ample room in telecommunications, transmission errors recovery, and microprocessor systems (intelligent memory addressing minimizing power consumption and cross talk).

Constructing the classical Gray sequence

There is a simple way of constructing the Gray codes for natural-number sequences of any length. Let us express numbers rigorously in binary and use k = 1, 2, 3, ... to denote the number of digits. We will proceed iteratively, starting with k = 1 and the minimum, finite Gray sequence {0,1}. Now proceed as follows:

Iterative passage:

Increment k and the next Gray code identical to the last one, but with bit k set to "1".

Keep the most significant bit set to "1", while repeating in the lower (k-1) bits all the codes created in the last iteration, but reading them *backwards* (hence the often used term *reflected binary code*) and, obviously (obviously, padding any missing leading bits by zeroes).

Repeat the iteration.

Here is how it works (binary notation; note that the result, up to k=4, are the codes 0-F shown in Fig.2.):

k=1 : 0, 1
k=2 : 11, 10
k=3 : 110, 111, 101, 100
k=4 : 1100, 1101, 1111, 1110, 1010, 1011, 1001, 1000

In every iteration k, the full set of codes has 2^k elements which are a permutation of the first 2^k non-negative integers (0 ... 2^k-1) and, by construction, satisfy the conditions (a) and (b). Moreover, the process can be continued up to any k, thus defining an infinite Gray sequence.

This particular and most frequently used Gray sequence is often referred to just as "the Gray codes" without mentioning that there are an infinity of such sequences. One reason for its popularity is that it can be easily encoded algorithmically and/or hard-wired electronically in FPGA's and the like. Denoting its n-th entry as Gray0(n) and using standard C, we have, in fact:

```
unsigned int NToGray0(unsigned int n) {return (n^(n >> 1));}
```

so that the conversion of an integer to the corresponding Gray0 code requires just a right shift (>>) by one bit and a bitwise XOR (^) of the result with the original number. The inverse operation is a little bit more complicated, though not too much:

```
unsigned int Gray0ToN(unsigned int g) {
  for (unsigned int mask = g>>1; mask != 0; mask >> =1)  g ^= mask;
  return g;
}
```

Check, for example, that NToGray0(5) = 7, and Gray0ToN(7) = 5. We will not go here into other practical aspects of Gray0, such as its hard-wired implementations in silicon, since they are amply treated in the literature.

More definitions

Often the motion to be encoded is the angular displacement of a rotating shaft (shaft encoders). In this case the encoding strips are wrapped around the shaft, the Gray sequence must be of finite length, and an additional condition must be met:

c) The last entry in the finite sequence must differ from the first one also by only one bit.

Any finite sequence of positive integers which meet the conditions (a) and (c) will be called a **Gray cycle**.. A simple example is {6,7,5,4}.

It is desirable (though not necessary) that when a Gray cycle is of length m, it includes all integer numbers from 0 to m-1; in other words, it represents a permutation of the first m non-negative integers (no missing codes). Such sets of Gray codes will be called **basic Gray cycles** (example: {3,7,6,4,5,1,0,2}).

A normal Gray cycle which starts with zero will be called a **normal Gray cycle**. In most practical situations, it makes little or no difference if a Gray cycle is rotated left or right, since the properties (a) and (c) remain intact. Consequently, the rotations can be viewed as defining an equivalence relation in which each equivalence class can be characterized by its unique basic Gray cycle starting with 0.
Example: {0,2,3,7,6,4,5,1}.

Another symmetry relation which implies a redundancy is a complete reversal of the cycle (for example, saying that {0,2,3,7,6,4,5,1} and {0,1,5,4,6,7,3,2} are both basic Gray cycles are clearly two equivalent ways of saying the same thing. All this leads to the following definition:

A normal Gray cycle of length m is said to be a **canonical Gray cycle** (**CGC**) if the number following the staring "0" is smaller than the last number in the cycle (which in fact precedes the starting "0").
Examples: {0,1,5,4,6,7,3,2} is canonical, while {0,2,3,7,6,4,5,1} is not.

From the construction in the preceding Section, it is clear that all starting portions of length 2^k, for any k>0, of the classical infinite sequence Gray0 define CGC's. One wonders, however, whether there exist any CGC's of other lengths and whether they are unique or multiple. To answer this, at least in part, consider explicitly these CGC's of length 10: {0,2,3,7,6,4,5,1,9,8} and {0,2,6,4,5,7,3,1,9,8}. They exemplify that (i) there exist CGC's of lengths other than 2^n and (ii) that there exist distinct CGC's of the same length. The question therefore is not *'whether'*, but *'how many'*.

Counting canonical Gray cycles

First of all, there are no generators for odd values of the cycle length m. This is because, in order to start at some binary bits settings and return to the same combination through a series of m single-bit flips, the number of steps must be even – for every up-flip, there must necessarily exist a down-flip.

S. Sykora, *Canonical Gray Cycles*, Stan's Library V, DOI 10.3247/SL5Math14.001.

For even values of m, for lack of anything better, we will use an ad-hoc, optimized algorithm based on the backtracking idea of Hamiltonian paths theory (see the next Section). It is better than brute force, but not particularly so.

The C++ program CGC listed in the Appendix allows one to examine, in principle, all canonical Gray cycles of a given length m and determines their number C(m). For the first few m, some of these results are shown in Table I.

Table I. Canonical Gray Cycles for small cycle lengths m

C(m) is the number of CGC's, while the third column contains all possible CGC's for $m \le 10$ and the first CGC found for $m \ge 12$. Odd values of m are not listed, because they do not admit any CGC's. The way the CGC's are generated implies that they are sorted by increasing terms values.

m	C(m)	CGC
2	1	0 1
4	1	0 1 3 2
6	1	0 2 3 1 5 4
8	6	0 1 3 2 6 7 5 4
		0 1 3 7 5 4 6 2
		0 1 5 4 6 7 3 2
		0 1 5 7 3 2 6 4
		0 2 3 1 5 7 6 4
		0 2 6 7 3 1 5 4
10	4	0 2 3 7 6 4 5 1 9 8
		0 2 6 4 5 7 3 1 9 8
		0 4 5 7 6 2 3 1 9 8
		0 4 6 2 3 7 5 1 9 8
12	22	0 1 3 7 5 4 6 2 10 11 9 8
14	96	0 1 3 2 6 7 5 4 12 13 9 11 10 8
16	1344	0 1 3 2 6 4 5 7 15 11 9 13 12 14 10 8
18	672	0 2 3 7 5 4 6 14 10 8 12 13 15 11 9 1 17 16
20	3448	0 1 3 7 5 4 6 14 12 8 9 13 15 11 10 2 18 19 17 16
22	12114	0 1 3 2 6 4 20 16 18 19 17 21 5 7 15 11 9 13 12 14 10 8
24	158424	0 1 3 2 6 4 5 13 9 8 12 14 10 11 15 7 23 19 17 21 20 22 18 16
26	406312	0 1 3 2 6 4 5 7 15 11 9 13 12 14 10 8 24 25 17 19 18 22 23 21 20 16
28	4579440	0 1 3 2 6 4 5 7 15 11 9 13 12 14 10 8 24 25 17 19 27 26 18 22 23 21 20 16
30	37826256	0 1 3 2 6 4 5 7 15 11 9 8 10 14 12 13 29 21 17 19 23 22 18 26 27 25 24 28 20 16
32	906545760	0 1 3 2 6 4 5 7 15 11 9 8 10 14 12 13 29 21 17 19 18 22 23 31 27 25 24 26 30 28 20 16
34	> 0	0 2 3 7 5 4 6 14 10 8 9 11 15 13 12 28 20 16 18 19 23 22 30 26 24 25 27 31 29 21 17 1 33 32
36	too hard	Excessive execution time trying to find a first CGC, not to speak about counting all of them

Notes: * Depending on the resolution of your screen, you may need to zoom the Table to display it properly.
** The value for m = 32 is taken from OEIS sequence A066037 (see below).

Final observations

There is a way to describe this problem as a search for Hamiltonian circuits in a particular König-type undirected graph [4-6]. Given m graph vertices labelled by integers from 0 to (m-1), join two vertices i and j by an edge if their binary expansions differ by exactly one bit. Then, evidently, every Hamiltonian circuit in the resulting graph coincides with a CGC. This puts the problem of finding and/or enumerating CGC's into a very challenging NP-complete category.

We know, thanks to the explicit construction discussed above, that for any cycle length of the type 2^k, with k = 1,2,3,... , there exists at least one CGC. The sequence C(m) is therefore infinite.

From Table I it is evident that for small values of m, after a certain initial 'hesitation', the values of C(m) tend to grow fast. However, this by no means guarantees that C(m) can't become 0 for some values of m. When looking only for the first CGC, the execution time is very short (below one second) for any m up to 32, and then suddenly shoots up, making a typical PC appear to 'hang' for about one day before a CGC is found. There appear to be profound dips for values of the type m = 2^k + 2 and it might be that for some higher m of this type no CGC exists. It might even be that for large enough m, except those of the type m = 2^k, no C(m) are non-zero because, for large m, the corresponding graphs become loosely connected (all vertices have ranks of $\log_2(m)\pm1$). All these statements, however, are at present pure conjectures.

Due to the combinatoric nature of the problem, execution times become very long once m exceeds 30 (many hours or even days of PC time). The algorithm presented in the Appendix is an ad-hoc variation on the popular backtracking search [7] for Hamiltonian circuits in undirected graphs. I have attempted *some* optimization, of course, such as avoiding stack-based nested iterations, and expanding all test calls inline. Another boost of performance comes from the observation that the "neighbors" of the starting "0" must be powers of 2, so that the second term in the CGC must be a power of 2 not exceeding m/2. All this, however, had a relatively modest effect (less than an order of magnitude), barely sufficient to boost the manageability of the computations from m = 28 to m = 32.

I have managed to compute C(m) for all m ≤ 30 and listed them in the OEIS sequence A236602. Actually, I have also added the value for cycle length 32 which has been computed (not by me) and published in A066037, listing what are basically our C(m) for m = 2^k. Hence, we know from that source that C(64) = 358382137225708837072O.

The actual CGC's found in this study, one for every even m up to m = 34, are listed in A236603.

The practical importance of Gray cycles is evident for the classical ones with cycle lengths 2^k. Other cycle lengths, such as 10^k, 60^k, 360, might have some practical applications, provided we could compute them, beyond the easy case of m = 10, but they are certainly not essential for such purposes. Rather, they might be useful as examples in the following algorithmic context:

The difficulty encountered in computing the values C(m) stems from the fact that the definition of the CGC's combines a very 'local' constraint imposed upon adjacent members of the cycle with a very 'global' one. In this case, the local constraint is that the binary expansions of neighbors should differ in just one bit, while the global constraint is that the CGC should be a mono-cyclic permutation of the first m integers.

There is little doubt that more efficient algorithms to find / enumerate CGC's should exist. However, I am not aware of any specific ones at this moment [maybe reference 8 will help] and, anyway, brute-force approaches are always useful as robust debugging tests when developing something more sophisticated.

References

[1] Wikipedia, Frank Gray.
[2] Wikipedia, Gray code.
[3] Wolfram Mathworld, Gray Code.
[4] Deo, Narsingh, *Graph Theory with Applications to Engineering and Computer Science*, Prentice Hall, Englewood Cliffs, 1974, ISBN 978-0133634730.
[5] Wikipedia, Hamiltonian path.
[6] Wikipedia, Hamiltonian path problem.
[7] Patel Dipak et al, *Hamiltonian cycle and TSP: A backtracking approach*, International Journal on Computer Science and Engineering (IJCSE), 2, p.1413, 2011.
[8] Chalaturnyk A., *A Fast Algorithm For Finding Hamilton Cycles*, Thesis, University of Manitoba, 2008, available online.

Appendix: the CGC function code

The following is a heavily commented listing of the number-crunching part of the C++ program I have used to compute the reported data:

```
BOOL DiffersByOneBinaryBit(LONG i,LONG j)
//---------------------------------------------------------------------
// Returns true if i and j differ by exactly one binary bit
{
  LONG k = i^j;                  // k = (i XOR j) must be a power of 2
  if (!k) return false;
  return !(k & (k-1));
}

BOOL isPresent(LONG* arr, LONG size, LONG val)
//---------------------------------------------------------------------
// Returns true if val is among the arr elements
{
  LONG k;
  for (k=0; k<size; k++) {if (arr[k]==val) return true;}
  return false;
}

DWORD CGC(LONG m,DWORD mode)
//---------------------------------------------------------------------
// This is the arithmetic core of Stan's CGC functions utility.
// The function returns the number of CGC's found.
// The argument m is the CGC cycle length.
// mode can be 1, 2, or 3  and determines the following behavior:
// mode = 1: As soon as a CGC is found,
//           it is dumped and the function exits, returning 1.
// mode = 2: All found CGC's ar dumped,
//           and the function returns their count.
// mode = 3: All CGC's are  computed, but nothing is dumped,
//           and the function returns the CGC's count.
// In all modes, if no CGC is found, nothing is dumped
// and the returned value is 0.
// The dumping function prototype is
//     void DumpCGC(LONG cyclen,LONG* cgc,DWORD n,DWORD mode);
// If modes 1 or 2 are to be used, DumpCGC must be supplied by User.
{
  DWORD n,cgc1max;
  LONG  k,l,stkidx;
  LONG* cgc = new LONG[m];         // Allocate a new array for the CGC

  cgc1max = 1;                                     // An optimization
  while (2*cgc1max < m) cgc1max *= 2;

  n = 0;
  if (!(m%2)) {                                     // m must be even
```

```
    k = 0;                                                  // Initializations
    l = 0;
    stkidx = k;                                         // Start "stack" index
    cgc[k] = l;                           // First element in CGC must be l=0
    while (++k < m) {
      while (++l < m) {                          // Scan possible cgc[k] values
        if ((k == 1) && (l > cgc1max)) {l = m; break;}
        if (DiffersByOneBinaryBit(cgc[k-1],l)) {    // Local constraint
          if (!isPresent(cgc+1,k-1,l)) {        // May not be used twice
            cgc[k] = l;                          // A good cgc[k] candidate ...
            if (k == (m-1)) {                    // If this is the last k-term:
              if (l >= cgc[1]) {                  // Test the cyclic properties
                if (DiffersByOneBinaryBit(cgc[0],l)) {
                  n++;                                      // Found a good CGC!
                  if (mode < 3) DumpCGC(m,cgc,n,mode);
                }
              }
              l = m;    // Max one l fits the last term (a permutation)
            } else {                                  // An intermediate term:
              stkidx = k;                         // ... remember it on 'stack'
            }
            break;
          }
        }
      }
      if (n && (mode==1)) break;
      if (l == m) {                         // All l's were explored for this k
        if (!stkidx) break;                // There can't be any more solutions
        k = stkidx-1;
        l = npc[stkidx];
        stkidx--;
      } else {
        l = 0;
      }
    }
  }

  delete [] cgc;
  return n;
}
```

History of this document

1 Feb 2014: Assigned a DOI (10.3247/SL5Math14.001) and uploaded the article online.

29 Feb 2014: Corrected the code on lines 29-31 of page 8, which now reads

```
k = stkidx-1;
l = npc[stkidx];
stkidx--;
```

instead of the incorrect

```
stkidx--;
k = stkidx-1;
l = npc[stkidx];
```

The bug caused no problems in the cases discussed in this article, but would be detrimental in future extensions (tested by counting Hamiltonian cycles in complete graphs).