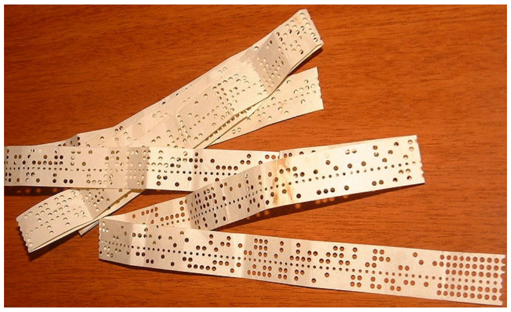
Gray Codes



Contents

1 Exhaustive generation

Now lets return to consider *exhaustive* generation. We have already discussed *lexicographical* ordering of binary strings– which is essentially like an odometer. We are particularly interested in *minimal change algorithms* in which each object is generated from the previous by a minimal change, however that is measured.

1.1 Questions to ask for any exhaustive generation scheme

Given a combinatorial class $\mathcal{A}_n$, let $C = C(\mathcal{A}_n) = [C_1, C_2, \dots]$ be a permutation of the elements. We recall the notion of a ranking operator on a class $\mathcal{A}$ which is a bijective map rank which takes an element $\alpha \in \mathcal{A}_n$ and returns a positive integer $r \in [1..A_n]$.

What is the successor rule? Given an element, is it easy to determine the next element in the sequence? (without tracing through a stored version of all elements, of course...)

What is the unranking function or encoding rule for C? Given r, can we quickly determine the r-th element in C.

What is the ranking function/ decoding rule for C?

Our first, fundamental class of exhaustive generation algorithms are called *Gray codes*.

2 Gray Codes

Distance for binary strings is the number of positions in which the binary strings differ. This is called the Hamming distance. A Gray code is an ordering of binary strings in which two consecutive strings have distance 1. The usual binary encoding of integers is *not* a Gray code since, for example, when we switch from $2^k - 1$ to 2^k, we will change $k + 1$ bits.

However, we can modify this so that we iteratively change the least significant bit that will form a new code word:

$$\begin{array}{cccc}
0 & 0 & 0 & 0 \\
0 & 0 & 0 & 1 \\
0 & 0 & 1 & 1 \\
0 & 0 & 1 & 0 \\
0 & 1 & 1 & 0 \\
0 & 1 & 1 & 1 \\
 & & \vdots &
\end{array}$$

However, in this case, the last string will be 1111, which is quite far from 0000. Now, there are several different ways to exhaustively describe binary string using Gray codes.

2.1 Gray codes are useful in practice

Imagine your binary string describes the present state of a large manufacturing machine. You would like to test all settings on your machine. Using a Gray code to do this means that you only need to change one single setting each time. If changing a setting takes time, for example requires cleaning a part or something physical, considerable efficiency is gained by using a Gray code. Your machine may impose other constraints on your code, so we will also investigate other strategies for finding Gray codes.

2.2 Reflected Binary Gray Code (RBC)

The Gray code was described by Frank Gray of Bell labs who patented their use for shaft encoders in 1953. We recursively describe a scheme as follows. Let us denote by $R(n)$ the order of all binary strings of length n generated by the scheme.

In the first step list 0 then 1:

$$R(1) = 0, 1$$

Generate the next step by listing the code so far, then the code in the reverse order. Prepend a 0 to the listing in original orientation, and a 1 to the elements listed in reverse order:

$$R(2) = 00, 01, 11, 10$$

Repeat this process until the desired length is acheived:

$$R(3) = 000, 001, 011, 010, 110, 111, 101, 100$$

Once more:

$$R(4) = \begin{array}{cccc}
0 & 0 & 0 & 0 \\
0 & 0 & 0 & 1 \\
0 & 0 & 1 & 1 \\
0 & 0 & 1 & 0 \\
0 & 1 & 1 & 0 \\
0 & 1 & 1 & 1 \\
0 & 1 & 0 & 1 \\
0 & 1 & 0 & 0 \\
1 & 1 & 0 & 0 \\
1 & 1 & 0 & 1 \\
1 & 1 & 1 & 1 \\
1 & 1 & 1 & 0 \\
1 & 0 & 1 & 0 \\
1 & 0 & 1 & 1 \\
1 & 0 & 0 & 1 \\
1 & 0 & 0 & 0
\end{array}$$

In this generation scheme the last element and the first *also* differ only by 1. A Gray code is cyclic if the first and last element differ by 1.

Proposition. *$R(n)$ is a cyclic Gray code*

Proof. There are two things we need to show. First that every binary word appears exactly once and second that the distances (including between the first and last) are always 1.

We will do this by induction. The base case is $n = 1$ which is clear.

Take $n > 1$. By induction the elements of $R(n-1)$ are all binary strings of length $n-1$. Thus the first half of $R(n)$ consists of 2^{n-1} distinct binary strings of length n beginning with 0, and the second half of $R(n)$ consists of 2^{n-1} distinct binary strings of length n beginning with 1. Therefore $R(n)$ consists of 2^n distinct binary strings of length n. But this is all of them, so each binary string appears exactly once in $R(n)$.

By induction the difference between successive elements of $R(n-1)$ is always 1, so the difference between the ith and $(i+1)$st elements of $R(n)$ is 1 for

$$i \in \{0, 1, \ldots, 2^{n-1} - 2, 2^{n-1}, \ldots, 2^n - 1\}$$

It remains to check the middle and the first and last. For the middle

$$d(R(n)_{2^{n-1}-1}, R(n)_{2^{n-1}}) = d(0R(n-1)_{2^{n-1}-1}, 1R(n-1)_{2^n-2^{n-1}-1}) = D(0R(n-1)_{2^{n-1}-1}, 1R(n-1)_{2^{n-1}-1}1$$

For the first and last

$$d(R(n)_0, R(n)_{2^n-1}) = d(0R(n-1)_0, 1R(n-1)_{2^n-(2^n-1)-1}) D(0R(n-1)_0, 1R(n-1)_0) = 1$$

□

Letting $d(w)$ be the Hamming weight of the binary string w, that is the number of nonzero entries of w this gives the following successor algorithm

Algorithm: GraySuccessor

```
input: n, w.   w is a binary word of length n

result = w
if d(w) is even
   flip the last bit of result
else
   j=n
   while result(j) = 0 and j > 0
      j = j-1
   if j=1
      return no successor
   flip (j-1)th bit of result

return result
```

Note that the while loop finds the rightmost 1 and then we operate on the bit before it. For example if $n = 4$ and $w = 0010$ then we are not even, so we end up in the loop, and the while loop terminates with $j = 3$, so we swap bit 2 to get the successor 0110.

Exercise. Prove by induction that the successor algorithm works

We can also encode this code by the sequence of positions that are changed, called the transition sequence:

$$(0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0).$$

when indexing with 0 on the right, or with 0 on the left this would be

$$(3, 2, 3, 1, 3, 2, 3, 0, 3, 2, 3, 1, 3, 2, 3).$$

Is there a way to have direct access to the k^{th} element in the sequence? Let us denote this string by $r_n(k)$. For example, $r_n(1) = 0^n$, and $r_n(2^n) = 10^{n-1}$. For which k is $r_n(k) = 1^n$?

2.3 Unranking

To describe this, let us first recall the "exclusive or" operator (XOR) operator:

$$0 \oplus 0 = 1 \oplus 1 = 0 \quad \text{and} \quad 0 \oplus 1 = 1 \oplus 0 = 1.$$

To encode an integer k with $1 \leq k \leq 2^n$ into the word $C_k = (c_{n-1}, \dots, c_1, c_0)$ we first write $k-1$ in binary: $k - 1 = (b_{n-1}, b_{n-2}, \dots, b_0)$[1]. Then compute the c_j

$$c_{n-1} = b_{n-1}$$

$$c_j = b_j \oplus b_{j+1}, \quad j = 0, 1, \dots, n-1.$$

Example. For example, $r_4 = (12) = r_4(1100_2) = 1010$, which we can confirm in the above figure, the 13-th string is 1010.

3 Other Gray Codes

3.1 Balanced Transition Counts

If we look at the transition sequence, most of the time, we change the rightmost bit, and there is only one place we change the rightmost (twice if we count cyclically). For any cyclic n-bit Gray code C we can keep track of the transition count, a vector $(k_{n-1}, k_{n-2}, \dots, k_0)$ where k_i is the number of times position i transitions. Prove that each k_i is even (consider it cyclically) and that the average value is $2^n/n$. It can be shown that for every $n = 2^m$ there exists a Gray code where the transition counts are equal. This is a difficult construction.

3.2 Beckett-Gray code

Another interesting type of Gray code is the BeckettGray code. The BeckettGray code is named after Samuel Beckett, an Irish playwright especially interested in symmetry. One of his plays, "Quad", was divided into sixteen time periods. At the end of each time period, Beckett wished to have one of the four actors either entering or exiting the stage; he wished the play to begin and end with an empty stage; and he wished each subset of actors to appear on stage exactly once. Clearly, this meant the actors on stage could be represented by a 4-bit binary Gray code. Beckett placed an additional restriction on the scripting, however: he wished the actors to enter and exit such that the actor who had been on stage the longest would always be the one to exit. The actors could then be represented by a first in, first out queue data structure, so that the first actor to exit when a dequeue is called for is always the first actor which was enqueued into the structure. Beckett was unable to find a BeckettGray code for his play, and indeed, an exhaustive listing of all possible sequences reveals that no such code exists for n = 4. Computer scientists interested in the mathematics behind BeckettGray codes have found these codes very difficult to work with. It is today known that codes exist for $n = \{2, 5, 6, 7, 8\}$ and they do not exist for $n = \{3, 4\}$.

[1]That is, $k - 1 = b_0 + 2b_1 + 4b_2 + \dots 2^{n-1}b_{n-1}$