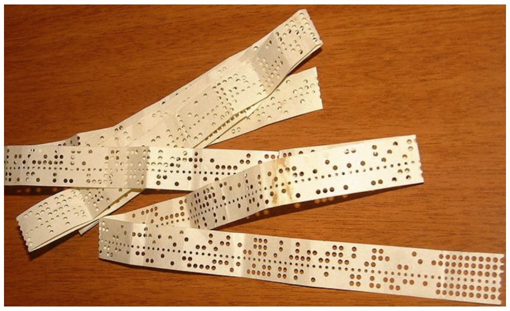
Gray Codes



Contents

1 Exhaustive generation – Minimal Change Order

Now lets reconsider *exhaustive* generation. We have already discussed *lexicographical* ordering of binary strings and of k-subsets. Here we are particularly interested in *minimal change orders* in which each object is generated from the previous object by making a "minimal change".

1.1 Questions to ask regarding exhaustive generation schemes

Given a combinatorial class $\mathcal{A}$, and $n \geq 0$, let $C_n = C(\mathcal{A}_n) = [c_1, c_2, \dots, c_{A_n}]$ be a permutation of the elements in $\mathcal{A}_n$ (the elements of size n in $\mathcal{A}$). The sequence of permutations $C = (C_0, C_1, C_2, \dots)$ is an *ordering rule* for $\mathcal{A}$. For example the *lexicographic order* is an ordering rule that we have already seen.

For $n \geq 0$, the *unranking operator* for the permutation $C(\mathcal{A}_n) = [c_1, c_2, \dots, c_{A_n}]$ is the bijection UNRANK which maps each integer $r \in [A_n] = \{1, 2, \dots A_n\}$ to the object $c_n \in \mathcal{A}_n$. The *ranking operator* for C_n is the inverse bijective map $\text{RANK} : \mathcal{A}_n \to [A_n]$ That is, for $\alpha \in \mathcal{A}_n$ we have $\text{RANK}(\alpha) = r$ if $c_r = \alpha$.

Here are some questions one can address.

Minimal Change Order Can we find a *minimal change order* $[c_1, c_2, \dots, c_{A_n}]$ for $\mathcal{A}_n$? This is a permutation where, for $i = 2, 3, \dots, A_n$, the element c_i can be obtained from its predecessor c_{i-1} by making the "smallest" possible change.

Successor/Predecessor rule? Let C_n be an ordering of $\mathcal{A}_n$. Given an element $\alpha \in \mathcal{A}_n$, can we easily determine successor (or predecessor) of α (without storing or searching the list C_n)?

Fast Unranking (i.e. Encoding) For a given ordering rule $C(\mathcal{A}_n) = (c_1, \dots c_{A_n})$, can we quickly construct the r-th element $c_r = \text{UNRANK}(r)$ for $r = 1, 2, \dots, A_n$?

Fast Ranking (i.e. Decoding) For a given ordering rule $C(\mathcal{A}_n) = (c_1, \dots c_{A_n})$, can we quickly determine $\text{RANK}(\alpha)$ for $\alpha \in \mathcal{A}_n$?

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

2 Gray Codes

Let A be an alphabet. The **Hamming distance** between two n-bit binary words in A^n is defined to be the number of positions in which the two strings differ. A **(binary) Gray code** is an ordering of n-bit binary words in which every two consecutive strings have Hamming 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.

Example. Each column of the following table is a 3-bit binary Gray code beginning with 000.

000	000
001	001
011	011
010	010
110	110
111	100
101	101
100	111

A Gray code is **cyclic** if its last word and its first word are also at Hamming distance 1. For example, the first Gray code listed above is cyclic, but the second one is not cyclic.

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.

There are many other applications of binary Gray codes in engineering, computer science, statistics etc.

2.2 Reflected Binary Gray Code (RBC)

The most natural n-bit binary Gray code, which we denote by $R(n)$, is called the **Reflected Binary Code (RBC)**. The RBC was described by Frank Gray of Bell Labs in a 1953 patent regarding electronic communication. Here are the first three RBC's

$$R(1) = 0, 1, \quad R(2) = 00, 01, 11, 10 \quad R(3) = 000, 001, 011, 010, 110, 111, 101, 100$$

We can recursively describe the n-bit RBC as follows.

$$R(1) = 0, 1 \qquad \text{and} \qquad R(n+1) = 0R(n), 1R^r(n) \quad \text{for } n = 1, 2, \dots.$$

Here $0R(n)$ is the list obtained from $R(n)$ by pre-appending "0" in front of each of the words in $R(n)$, and $1R^r(n)$ is the list obtained by writing $R(n)$ in *reverse order* to get $R^r(n)$, then pre-appending "1" to each of the words in $R^r(n)$.

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

Proof. There are three things we will to show. First that every binary word appears exactly once in $R(n)$, second that the distances between successive words in $R(n)$ (including the first and last word) is always 1. We do this by induction on n. These two properties are obviously true for the base case $n = 1$.

Let $n \geq 1$. By the induction hypothesis, every n-bit binary word appears exactly once in $R(n)$. Therefore every $(n+1)$-bit binary word that begins with "0" appears exactly once in $0R(n)$, and every $(n+1)$-bit binary word that begins with "1" appears exactly once in $1R^r(n)$. Therefore $R(n+1) = 0R(n), 1R^r(n)$ lists each $(n+1)$-bit binary word exactly once.

We now show the second claim. By the induction hypothesis, the difference between successive elements of $R(n)$ is always 1, so the difference between the ith and $(i+1)$st elements of $R(n+1)$ is 1 for

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

It remains to compare the middle two words, $R(n+1)_{2^n-1}$ and $R(n+1)_{2^n}$, and to compare the first and last words, $R(n+1)_0$ and $R(n+1)_{2^{n+1}-1}$. By the construction, each of the words $R(n+1)_{2^n-1}$ and $R(n+1)_{2^n}$ is obtained from the last word in $R(n)$ by pre-appending either a 0 or a 1 (respectively). Therefore these two words are at hamming distance 1.

Similarly each of $R(n+1)_0$ and $R(n+1)_{2^n-1}$ is obtained from $R(n)_0$ by pre-appending either 0 or 1, so these two words are also at Hamming distance 1. ☐

2.2.1 RBC Successor rule:

Letting $d(w)$ be the Hamming weight of the binary string w, that is the number of nonzero entries of w. An obvious algorithm will determine $d(w)$ in linear time. We describe an algorithm which determines the RBC-successor of an n-bit binary word $w = w_{n-1}w_{n-1}\dots w_1w_0$.

Algorithm: GraySuccessor

```
input: n, w = w[n-1],w[n-2], ... , w[0]  where each w[i] is in {0,1}

result = w
if d(w) is even
   result[0] = 1-result[0]  (flip the last bit of result)
else
   j=0
   while result[j] = 0 do
      j = j+1
      if j=n-1
         return "no successor"
   result[j+1] = 1 - result[j+1]
return result
```

Note that if w has odd Hamming weight, then the while loop finds the rightmost 1 in w (if it exists), then flips the bit just before it. For example if $n = 4$ and $w = (w_3w_2w_1w_0) = 0010$, then $d(w) = 1$, so we execute the while loop, which terminates with $j = 1$, we flip bit w_2 and output the successor 0110.

Exercise. Prove by induction that the successor algorithm works correctly.

Transition sequence:
We can efficiently describe any binary Gray code starting with $00\dots0$ by the sequence of positions that are changed between successive pairs of words. This sequence called the transition sequence of the Gray code. For example, the transition sequences for the 3-bit and 4-bit RBC are as follows (assuming w_0 is the rightmost bit of a word w).

$$T_3 = (0, 1, 0, 2, 0, 1, 0), \qquad T_4 = (0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0).$$

Exercise. Is there an easy way to find the transition sequence for the $(n+1)$-bit RBC from that of the n-bit RBC?

2.2.2 RBC Unranking/Ranking

Is there a way to directly construct the k^{th} element in the n-bit RBC? Let us denote this string by $\textsc{Unrank}(k)$. For example, $\textsc{Unrank}(1) = 0^n$, and $\textsc{Unrank}(2^n) = 10^{n-1}$.

Conversely, is there a direct to compute $\textsc{Rank}(w)$ for an n-bit word w in RBC. For for example, how can we compute $\textsc{Rank}(11\dots1)$, which is the position of the word 1^n in the RBC?

The answer to both questions is "yes", with simple algorithms that rely on the "exclusive or" operator (XOR):

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

To **encode** an integer k with $1 \le k \le 2^n$ into the word $w = (w_{n-1}, \dots, w_1, w_0)$ we first write $k-1$ in binary: $k - 1 = (b_{n-1}, b_{n-2}, \dots, b_0)$ [1]. Then compute the w_j as follows

$$\begin{aligned} w_{n-1} &= b_{n-1} \\ w_j &= b_j \oplus b_{j+1}, \quad \text{for } j = 0, 1, \dots, n-1. \end{aligned}$$

Example. For example, the 13th word in the 4-bit RBC is found as follows.

$$\begin{aligned} 13 - 1 &= 8 + 4 = 1100_2 \\ w(3) &= 1, \quad w(2) = 1 \oplus 1 = 0, \quad w(1) = 1 \oplus 0 = 1, \quad w(0) = 0 \oplus 0 = 0 \\ \textsc{Unrank}(13) &= w_4 w_2 w_1 w_0 = 1010. \end{aligned}$$

How do we **decode**? The fact that $a \oplus b = c$ if and only if $aoplusc = b$ makes it just as easy to rank words in the RBC. We can easily find the position of a given binary word $w = (w_{n-1}, w_{n-2}, \dots, w_1, w_0)$ in the RBC as follows.

```
$b_{n-1} = w_{n-1}$
For $j = n-2, n-3, \dots, 0$ do:
    $b_j = b_{j+1} \oplus c_j$
Output: $\textsc{Rank}(w) = 1 + (2^{n-1} b_{n-1} + 2^{n-2} b_{n-2} + \cdots + b_0)$.
```

Note: In the above procedure, we have that $b_j = w_{n-1} \oplus w_{n-2} \oplus \cdots \oplus w_j$, for $0 \le j < n$.

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 Beckett–Gray code. The Beckett–Gray 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

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

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 Beckett–Gray 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 Beckett–Gray 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 $n = \{3, 4\}$.