

# A Loopless Gray Code for Minimal Signed-Binary Representations

Gurmeet Singh Manku<sup>1</sup> and Joe Sawada<sup>2</sup>

<sup>1</sup> Google Inc., USA,

manku@cs.stanford.edu <http://www.cs.stanford.edu/~manku>

<sup>2</sup> University of Guelph, Canada,

sawada@cis.uoguelph.ca <http://www.cis.uoguelph.ca/~sawada>

**Abstract.** A string  $\dots a_2a_1a_0$  over the alphabet  $\{-1, 0, 1\}$  is said to be a minimal signed-binary representation of an integer  $n$  if  $n = \sum_{k \geq 0} a_k 2^k$  and the number of non-zero digits is minimal. We present a loopless (and hence a Gray code) algorithm for generating all minimal signed binary representations of a given integer  $n$ .

## 1 Introduction

A string  $\dots a_2a_1a_0$  is said to be a signed-binary representation (SBR) of an integer  $n$  if  $n = \sum_{k \geq 0} a_k 2^k$  and  $a_k \in \{-1, 0, 1\}$  for all  $k$ . A *minimal* SBR has the least number of non-zero digits. For example, 45 has five minimal SBRs: 101101, 110 $\bar{1}$ 01, 10 $\bar{1}$ 0 $\bar{1}$ 01, 10 $\bar{1}$ 00 $\bar{1}$  $\bar{1}$  and 1100 $\bar{1}$  $\bar{1}$ , where  $\bar{1}$  denotes  $-1$ . Our main result is a *loopless* algorithm that generates *all* minimal SBRs for an integer  $n$  in Gray code order. See Fig. 1 for an example. Our algorithm requires linear time for generating the first string. Thereafter, only  $O(1)$  time is required in the worst-case for identifying the portion of the current string to be modified for generating the next string<sup>3</sup>.

Volumes 3 and 4 of Knuth's *The Art of Computer Programming* are devoted entirely to algorithms for generation of combinatorial objects. For the output of such an algorithm to be considered a Gray code, successive objects must differ by a constant amount. However, the time required to obtain each new object may be  $\omega(1)$ . A generation algorithm is said to be loopless if after the initial object is generated, successive objects may be obtained in  $O(1)$  time in the worst-case. For a survey of Gray code generation algorithms, see Savage [20].

The earliest algorithm for listing all minimal SBRs is due to Ganesan and Manku [8]; however they did not consider the efficiency of implementing their algorithm. By modifying their technique, Sawada [21] was able to generate all minimal SBRs in constant amortized time. Additionally, the output constitutes a Gray code. However, the algorithm is not loopless, since successive strings require linear time in the worst case.

```
110100 $\bar{1}$  $\bar{1}$ 0 $\bar{1}$ 
10 $\bar{1}$ 0100 $\bar{1}$  $\bar{1}$ 0 $\bar{1}$ 
10 $\bar{1}$  $\bar{1}$ 00 $\bar{1}$  $\bar{1}$ 0 $\bar{1}$ 
10 $\bar{1}$  $\bar{1}$ 0 $\bar{1}$ 010 $\bar{1}$ 
10 $\bar{1}$ 010 $\bar{1}$ 010 $\bar{1}$ 
11010 $\bar{1}$ 010 $\bar{1}$ 
110011010 $\bar{1}$ 
10 $\bar{1}$ 0011010 $\bar{1}$ 
10 $\bar{1}$ 00110011
1100110011
11010 $\bar{1}$ 0011
10 $\bar{1}$ 010 $\bar{1}$ 0011
10 $\bar{1}$  $\bar{1}$ 0 $\bar{1}$ 0011
```

**Fig. 1.** A Gray code listing of minimal SBRs for 819. Successive strings differ in three adjacent positions.

<sup>3</sup> See <http://www.cs.stanford.edu/~manku/projects/graycode/index.html> for source code in C.

Our approach is novel — we first identify the *canonical* minimal SBR (see §2 for its definition). The canonical SBR is split into disjoint “chains”. Individual chains are handled by a Gray code algorithm which never outputs certain forbidden strings (see §4). The cross-product of all the chains is handled by a generalization of the Binary Reflected Gray Code (BRGC) [3, 10] (see §3 and §5). A detailed history of SBRs is presented in §6.

## 2 A Loopless Gray Code for Minimal SBRs

From earlier work by Sawada [21], we know that any minimal signed binary representation (SBR) for an integer  $n$  can be transformed into another minimal SBR for the same integer by repeated application of the following re-write rules:  $10\bar{1} \rightarrow 011$ ,  $011 \rightarrow 10\bar{1}$ ,  $\bar{1}01 \rightarrow 0\bar{1}\bar{1}$ , and  $0\bar{1}\bar{1} \rightarrow \bar{1}01$ . Our strategy for listing minimal SBRs in Gray code order is the following. We study the structural properties of a specific minimal SBR, popularly known as the *canonical* SBR. We then develop a procedure for listing all strings that result from repeated application of the four re-write rules to the canonical SBR.

**DEFINITION (Canonical SBR).** *Let  $S$  denote the binary representation of a given integer, padded with two leading zeros. For instance, integer 45 would correspond to the string  $S = 00101101$ .  $S_{\text{canonical}}$  is the unique minimal SBR for  $S$  such that the product of any two adjacent digits is 0. Thus we never have  $11$ ,  $\bar{1}\bar{1}$ ,  $\bar{1}1$  or  $1\bar{1}$  as a substring. For example,*

$$S = 00101011111010000001010110101000010100$$

$$S_{\text{canonical}} = 010\bar{1}0\bar{1}0000\bar{1}010000010\bar{1}0\bar{1}0\bar{1}0101000010100$$

$S_{\text{canonical}}$  has been used by previous authors (Reitwiesner [19], Chang and Tsao-Wu [6], Jedwab and Mitchell [12] and Prodinger [18]). In fact,  $S_{\text{canonical}}$  for integer  $n$  can be obtained by “bit-wise subtracting  $n/2$  from  $3n/2$ ” (Prodinger [18]). Starting with  $S_{\text{canonical}}$  is critical to the simplicity of our approach.

**DEFINITION (Blocks).** *A maximally long bit-sequence of  $(01)^+$  and  $(0\bar{1})^+$  in  $S_{\text{canonical}}$  is called a block. The following string has eight blocks (each block has been underlined):*

$$\underline{01} \underline{0\bar{1}0\bar{1}} 000 \underline{0\bar{1}} \underline{01} 0000 \underline{01} \underline{0\bar{1}0\bar{1}0\bar{1}} \underline{0101} 000 \underline{0101} 00$$

**DEFINITION (Chains).** *A chain is a maximally long sequence of two or more adjacent blocks. The following string has three chains (each chain has been underlined):*

$$\underline{01} \underline{0\bar{1}0\bar{1}} 000 \underline{0\bar{1}} \underline{01} 0000 \underline{01} \underline{0\bar{1}0\bar{1}0\bar{1}} \underline{0101} 000 0101 00$$

Two chains are separated by one or more 0s. Therefore, none of the four rewrite rules, when applied to one chain, affects another chain. This proves the following:

**Theorem 1.** *The set of minimal SBRs of  $S$  corresponds to the cross product of the sets of minimal SBRs for individual chains of  $S_{\text{canonical}}$ .*

We now develop two loopless algorithms: one for generating the minimal SBRs of a chain in Gray code order (see §5), and another for generating the cross-product of Gray codes (see §3).

### 3 Gray Codes for Cross-Products

Consider the cross product of  $m$  combinatorial objects:  $X_m \times X_{m-1} \times \cdots \times X_1$ , where object  $X_i$  has  $t_i \geq 2$  members which can be listed in Gray code order. Clearly, there is a 1-1 correspondence between members of the cross product and tuples of the form  $(a_m, a_{m-1}, \dots, a_1)$ , where  $a_i \in [1, t_i]$  represents the  $a_i$ -th object in the Gray code of  $X_i$ . When each  $t_i = 2$ , one possible Gray code for the set of tuples is the Binary Reflected Gray Code (BRGC) [10]. A generalization of the BRGC, developed by Bitner, Ehrlich, and Reingold [3], handles arbitrary values of  $t_i \geq 2$ . Procedure BRGC (displayed in Fig. 2) is such an algorithm.

Procedure BRGC maintains three tuples:  $(a_m, a_{m-1}, \dots, a_1)$  is the *current-tuple*,  $(d_m, d_{m-1}, \dots, d_1)$  is the *direction-tuple*, and  $(p_{m+1}, p_m, \dots, p_1)$  is the *pointer-tuple*.

INITIALIZE initializes the three tuples. The current-tuple has  $a_i = 1$  or  $a_i = t_i$ , chosen arbitrarily. The direction-tuple has initial value  $d_i = 1$  if  $a_i = 1$ ; otherwise  $d_i = -1$ . The pointer-tuple has initial value  $(m+1, m, m-1, \dots, 1)$ .

NEXT( $i$ ) updates  $a_i \leftarrow a_i + d_i$ .

IS\_TERMINAL( $i$ ) returns TRUE iff  $(a_i = t_i \text{ and } d_i = 1)$  or  $(a_i = 1 \text{ and } d_i = -1)$ .

The pointer-tuple lies at the heart of procedure BRGC. If  $p_1 = m+1$ , procedure BRGC terminates. Otherwise, let  $i = p_1$ . Then  $a_i$ , the  $i$ -th member of the current-tuple, is modified. The direction-tuple indicates whether to increment ( $d_i = 1$ ) or decrement ( $d_i = -1$ ) the value of  $a_i$ .

Sample output produced by the algorithm is shown in Table 1(A).

Procedure BRGC can easily be adapted to generate members of  $X_m \times X_{m-1} \times \cdots \times X_1$  in Gray code order. Clearly, such an algorithm is loopless if the algorithm that generates members of each  $X_i$  in Gray code order is loopless.

### 4 Gray Codes for Cross-Products with Forbidden Tuples

Let  $R_m$  denote the set of  $m$ -tuples  $(a_m, a_{m-1}, \dots, a_1)$  satisfying

- 1)  $\forall m \geq i \geq 1 : a_i \in [1, t_i], \quad \text{with } t_i \geq 2$
- 2)  $\forall m \geq i > 1 : (a_i = t_i) \Rightarrow (a_{i-1} = 1)$

For example, with  $t_3 = 2$ ,  $t_2 = 3$ , and  $t_1 = 3$ ,  $R_m$  consists of 3-tuples listed in Table 1(B). We now develop a loop-free algorithm for listing  $R_m$  in Gray code order. This algorithm will be used in §5 for listing minimal SBRs of chains.

**BRGC**

```

INITIALIZE

WHILE TRUE DO
  last ← 1
  i ← plast
  IF (i = m + 1) THEN exit

  NEXT(i)

  IF (IS_TERMINAL(i)) THEN
    di ← -di
    j ← i + 1
    pi ← pj
    pj ← j

  IF (i ≠ last) THEN plast ← last

```

**Fig. 2.** A generalization of the Binary Reflected Gray Code [3,10]. See Table 1(A) for sample output.

| (A) With BRGC |       |       |       |       |       |       |       | (B) With BRGC-RESTRICT |       |       |       |       |       |       |       |       |       |       |       |
|---------------|-------|-------|-------|-------|-------|-------|-------|------------------------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|
| $a_3$         | $a_2$ | $a_1$ | $p_4$ | $p_3$ | $p_2$ | $p_1$ | $d_3$ | $d_2$                  | $d_1$ | $a_3$ | $a_2$ | $a_1$ | $p_4$ | $p_3$ | $p_2$ | $p_1$ | $d_3$ | $d_2$ | $d_1$ |
| 1             | 3     | 1     | 4     | 3     | 2     | 1     | 1     | -                      | 1     | 1     | 3     | 1     | 4     | 3     | 2     | 1     | 1     | -     | 1     |
| 1             | 3     | 2     | 4     | 3     | 2     | 1     | 1     | -                      | 1     | 1     | 2     | 1     | 4     | 3     | 2     | 1     | 1     | -     | 1     |
| 1             | 3     | 3     | 4     | 3     | 2     | 2     | 1     | -                      | -     | 1     | 2     | 2     | 4     | 3     | 2     | 1     | 1     | -     | 1     |
| 1             | 2     | 3     | 4     | 3     | 2     | 1     | 1     | -                      | -     | 1     | 2     | 3     | 4     | 3     | 2     | 2     | 1     | -     | -     |
| 1             | 2     | 2     | 4     | 3     | 2     | 1     | 1     | -                      | -     | 1     | 1     | 3     | 4     | 3     | 3     | 1     | 1     | 1     | -     |
| 1             | 2     | 1     | 4     | 3     | 2     | 2     | 1     | -                      | 1     | 1     | 1     | 2     | 4     | 3     | 3     | 1     | 1     | 1     | -     |
| 1             | 1     | 1     | 4     | 3     | 3     | 1     | 1     | 1                      | 1     | 1     | 1     | 1     | 4     | 3     | 2     | 3     | 1     | 1     | 1     |
| 1             | 1     | 2     | 4     | 3     | 3     | 1     | 1     | 1                      | 1     | 2     | 1     | 1     | 4     | 4     | 2     | 1     | -     | 1     | 1     |
| 1             | 1     | 3     | 4     | 3     | 2     | 3     | 1     | 1                      | -     | 2     | 1     | 2     | 4     | 4     | 2     | 1     | -     | 1     | 1     |
| 2             | 1     | 3     | 4     | 4     | 2     | 1     | -     | 1                      | -     | 2     | 1     | 3     | 4     | 3     | 2     | 4     | -     | 1     | -     |
| 2             | 1     | 2     | 4     | 4     | 2     | 1     | -     | 1                      | -     |       |       |       |       |       |       |       |       |       |       |
| 2             | 1     | 1     | 4     | 4     | 2     | 2     | -     | 1                      | 1     |       |       |       |       |       |       |       |       |       |       |
| 2             | 2     | 1     | 4     | 4     | 2     | 1     | -     | 1                      | 1     |       |       |       |       |       |       |       |       |       |       |
| 2             | 2     | 2     | 4     | 4     | 2     | 1     | -     | 1                      | 1     |       |       |       |       |       |       |       |       |       |       |
| 2             | 2     | 3     | 4     | 4     | 2     | 2     | -     | 1                      | -     |       |       |       |       |       |       |       |       |       |       |
| 2             | 3     | 3     | 4     | 3     | 4     | 1     | -     | -                      | -     |       |       |       |       |       |       |       |       |       |       |
| 2             | 3     | 2     | 4     | 3     | 4     | 1     | -     | -                      | -     |       |       |       |       |       |       |       |       |       |       |
| 2             | 3     | 1     | 4     | 3     | 2     | 4     | -     | -                      | 1     |       |       |       |       |       |       |       |       |       |       |

**Table 1.** Output of BRGC (Fig 2) and BRGC-RESTRICT (Fig 3) for  $t_3 = 2$ ,  $t_2 = 3$ ,  $t_1 = 3$ . The initial tuple is  $(a_3, a_2, a_1) = (1, 3, 1)$ . The output is generated after each iteration of the WHILE loop. For simplicity we use '-' to represent -1.

Let  $G_m$  denote a Gray code for  $R_m$ . Then the reversal of  $G_m$ , denoted  $\overline{G}_m$ , is also a Gray code. We define  $G_m$  recursively as follows. The base cases are  $G_0 = ()$ , the empty tuple, and  $G_1 = (1), (2), \dots, (t_1)$ . For  $m \geq 1$ ,  $G_{m+1}$  depends upon the parity (odd/even) of both  $t_{m+1}$  and  $t_m$ . Four cases arise; the sequence of  $(m+1)$ -tuples for  $G_{m+1}$  for the four cases is defined below.

| (even, even)                 | (even, odd)          | (odd, even)                  | (odd, odd)           |
|------------------------------|----------------------|------------------------------|----------------------|
| $1\overline{G}_m,$           | $1\overline{G}_m,$   | $1G_m,$                      | $1G_m,$              |
| $2\overline{G}_m,$           | $2\overline{G}_m,$   | $2\overline{G}_m,$           | $2\overline{G}_m,$   |
| $3\overline{G}_m,$           | $3\overline{G}_m,$   | $3G_m,$                      | $3G_m,$              |
| $4\overline{G}_m,$           | $4\overline{G}_m,$   | $4\overline{G}_m,$           | $4\overline{G}_m,$   |
| $\dots$                      | $\dots$              | $\dots$                      | $\dots$              |
| $t_m\overline{G}_m,$         | $t_m\overline{G}_m,$ | $t_m\overline{G}_m,$         | $t_m\overline{G}_m,$ |
| $t_{m+1}1\overline{G}_{m-1}$ | $t_{m+1}1G_{m-1}$    | $t_{m+1}1\overline{G}_{m-1}$ | $t_{m+1}1G_{m-1}$    |

The notation  $xG_i$  denotes a sequence of tuples with  $i+1$  members: the first member of each tuple is  $x$ ; the remaining members of the tuple constitute  $G_i$ . The last tuple in  $G_m$  is the same as the first tuple in  $\overline{G}_m$  and *vice versa*. Thus, since the first tuple in each listing begins with a one,  $G_{m+1}$  for  $m \geq 1$  is indeed a Gray code for  $R_{m+1}$ .

| <b><u>BRGC-RESTRICT</u></b>                                | Procedure INITIALIZE:                                   |
|--|---|
| INITIALIZE   | FOR $i \leftarrow m + 1$ DOWNTO 1 DO $p_i \leftarrow i$ |
| WHILE TRUE DO  | $a_m \leftarrow d_m \leftarrow 1$                       |
| last $\leftarrow$ MAP(1)                                   | IF (EVEN( $t_m$ )) THEN $rev \leftarrow$ <b>true</b>    |
| $i \leftarrow$ MAP( $p_{\text{last}}$ )                    | ELSE $rev \leftarrow$ <b>false</b>                      |
| IF ( $i = m + 1$ ) THEN <b>exit</b>                        | FOR $i \leftarrow m - 1$ DOWNTO 1 DO                    |
| NEXT( $i$ )  | IF $rev =$ <b>false</b> THEN                            |
| IF (IS_TERMINAL( $i$ )) THEN                               | $a_i \leftarrow d_i \leftarrow 1$                       |
| $d_i \leftarrow -d_i$                                      | IF (EVEN( $t_i$ )) THEN $rev \leftarrow$ <b>true</b>    |
| $j \leftarrow$ MAP( $i + 1$ )                              | ELSE  |
| $p_i \leftarrow p_j$                                       | $a_i \leftarrow t_i$                                    |
| $p_j \leftarrow j$   | $d_i \leftarrow -1$                                     |
| IF ( $i \neq$ last) THEN $p_{\text{last}} \leftarrow$ last | $i \leftarrow i - 1$                                    |
|  | $a_i \leftarrow d_i \leftarrow 1$                       |
|  | IF (EVEN( $t_i$ )) THEN $rev \leftarrow$ <b>false</b>   |

**Fig. 3.** A loopless algorithm for listing restricted cross products. See Table 1(B) for sample output.

**Theorem 2.** Procedure BRGC-RESTRICT in Fig. 3 is a loopless algorithm for producing the Gray code  $G_m$ .

BRGC-RESTRICT (Fig. 3) differs from BRGC (Fig. 2) in two important aspects:

1. The initial string  $(a_m, a_{m-1}, \dots, a_1)$  has to be initialized appropriately (see procedure INITIALIZE). We begin by assigning  $a_m \leftarrow 1$ . The recursive definition of  $G_m$  then helps us determine the initial values for each  $a_i$ , where  $m - 1 \geq i \geq 1$ . To do this we need only keep track of whether or not  $a_i$  is the first member in the first  $i$ -tuple of  $G_i$  or  $\overline{G}_i$ . The variable  $rev$  is used to determine the list. Recall that the direction  $d_i$  is initialized to 1 if  $a_i = 1$ . If  $a_i = t_i$ , then  $d_i$  is initialized to  $-1$ . The initialization for the “pointer-tuple”  $p$  is the same as before:  $(m + 1, m, m - 1, \dots, 1)$ .
2. We employ a function MAP which is defined as follows:

$$\text{MAP}(i) = \begin{cases} i + 1 & \text{if } (m > i \geq 1) \text{ and } (a_i = 1) \text{ and } (a_{i+1} = t_{i+1}) \\ i & \text{otherwise} \end{cases}$$

If MAP( $i$ ) always returns  $i$ , then BRGC-RESTRICT would be identical to BRGC.

An interesting special case corresponds to  $t_i = 2$  for all  $i$ . Then  $G_m$  consists of  $m$ -digit strings using the digits  $\{1, 2\}$  in which 22 is a forbidden substring. The total number of such strings equals the  $(m + 1)^{\text{st}}$  Fibonacci number.

## 5 A Loopless Gray Code for Chains

We begin with two examples for illustration of our approach.

EXAMPLE (Chain with 2 Blocks). Let  $B_2B_1 = (0\bar{1})^s(01)^t$ . A rewrite rule is applicable only where the two blocks join:  $\bar{1}01 \rightarrow 0\bar{1}\bar{1}$ , to obtain  $(0\bar{1})^{s-1}00\bar{1}\bar{1}(01)^{t-1}$ . Now, we could apply the inverse rule ( $0\bar{1}\bar{1} \rightarrow \bar{1}01$ ) to obtain the previous string, or we can apply the same rule again to the unique substring  $\bar{1}01$  in the new representation. This pattern will repeat until we reach the end of the chain. The number of minimal SBRs for this chain is  $t + 1$  and is independent of  $s$ . As an example, if  $s = 2$  and  $t = 3$ , then the 4 minimal SBRs of  $0\bar{1}0\bar{1}010101$  will be:  $0\bar{1}0\bar{1}010101, 0\bar{1}00\bar{1}\bar{1}0101, 0\bar{1}00\bar{1}0\bar{1}\bar{1}01$  and  $0\bar{1}00\bar{1}0\bar{1}0\bar{1}\bar{1}$ . Only  $B_1$  is changing, except for the rightmost digit of  $B_2$  that changes after the first rewrite.  $\square$

EXAMPLE (Chain with 3 Blocks). Without loss of generality, let  $B_3B_2B_1 = (0\bar{1})^s(01)^t(0\bar{1})^u$ . In this case, we can again apply the rewrite rules between  $B_3$  and  $B_2$  as with the two block case, but now we can also apply similar rewrite rules between  $B_2$  and  $B_1$ . The only difference is that the rewrite rules between  $B_2$  and  $B_1$  can only be applied if the state of  $B_2$  has not been altered to its final state where it ends with  $\bar{1}\bar{1}$ . In that case, no rewrite rules are possible between the two blocks (block  $B_1$  must remain in its initial form:  $(0\bar{1})^u$ ). If we ignore the leftmost block, observe that this problem is an instance of the restricted cross products (where  $m = 2$ ) described in §4.  $\square$

To generalize the above observations, we define

$$s(k, \ell) = \begin{cases} (01)^k & \text{if } \ell = 1 \\ (\bar{1}0)^{\ell-2}\bar{1}\bar{1}(01)^{k-\ell+1} & \text{if } 1 < \ell \leq k + 1 \end{cases}$$

For block  $B_i = (01)^k$  (that is not the leftmost block of a chain), the sequence  $s(k, 1), s(k, 2), \dots, s(k, k + 1)$  corresponds to the  $k + 1$  different strings that the block  $B_i$  may cycle through. The string  $\bar{s}(k, \ell)$  is defined similarly, with 1 and  $\bar{1}$  interchanged throughout the string. Examples:

$$\begin{array}{ll} s(1, 1) = 01 & \bar{s}(1, 1) = 0\bar{1} \\ s(1, 2) = \bar{1}\bar{1} & \bar{s}(1, 2) = 11 \end{array} \quad \begin{array}{ll} s(4, 1) = 01010101 & \bar{s}(4, 1) = 0\bar{1}0\bar{1}0\bar{1}\bar{1} \\ s(4, 2) = \bar{1}\bar{1}010101 & \bar{s}(4, 2) = 110\bar{1}0\bar{1}\bar{1} \\ s(4, 3) = \bar{1}0\bar{1}\bar{1}0101 & \bar{s}(4, 3) = 10110\bar{1}\bar{1} \\ s(4, 4) = \bar{1}0\bar{1}0\bar{1}\bar{1}01 & \bar{s}(4, 4) = 1010110\bar{1} \\ s(4, 5) = \bar{1}0\bar{1}0\bar{1}0\bar{1}\bar{1} & \bar{s}(4, 5) = 10101011 \end{array}$$

Using these strings we can now formally map the problem of cycling through all minimal SBRs of a chain  $B_{m+1}B_m \cdots B_1$  to the problem of generating restricted  $m$ -tuples. Without loss of generality assume that  $m$  is odd and that each  $B_i$  is initially defined as follows:

$$\begin{array}{lll} B_{m+1} = \bar{s}(k_{m+1}, 1) & = & (0\bar{1})^{k_{m+1}}, \\ B_m = s(k_m, 1) & = & (01)^{k_m}, \\ B_{m-1} = \bar{s}(k_{m-1}, 1) & = & (0\bar{1})^{k_{m-1}}, \\ \dots & & \dots \\ B_2 = s(k_2, 1) & = & (01)^{k_2}, \\ B_1 = \bar{s}(k_1, 1) & = & (0\bar{1})^{k_1}. \end{array}$$

Then a listing of all minimal SBRs for the chain is a subset of the cross-product of strings in blocks  $B_m, B_{m-1}, \dots, B_1$ , satisfying two constraints for  $m \geq i > 1$ :

- (1) If the string in block  $B_i$  equals  $s(k_i, k_i + 1)$ , then the string in block  $B_{i-1}$  must equal  $\bar{s}(k_{i-1}, 1)$ .
- (2) If the string in block  $B_i$  equals  $\bar{s}(k_i, k_i + 1)$ , then the string in block  $B_{i-1}$  must equal  $s(k_{i-1}, 1)$ .

A Gray code for the chain can be obtained by setting  $t_i = k_i + 1$  for  $m \geq i \geq 1$  and using BRGC-RESTRICT outlined in §4. There is a 1-1 correspondence between tuples generated by BRGC-RESTRICT and strings assigned to blocks. A tuple  $(a_m, a_{m-1}, a_{m-2}, \dots, a_1)$  generated by BRGC-RESTRICT corresponds to the following configuration: string  $s(k_m, a_m)$  in block  $B_m$ , string  $\bar{s}(k_{m-1}, a_{m-1})$  in block  $B_{m-1}$ , string  $s(k_{m-2}, a_{m-2})$  in block  $B_{m-2}$ , and so on. The only special consideration is that rightmost bit in the leftmost block  $B_{m+1}$  must be changed to 0 iff  $B_m$  is not in its original state. This is a trivial constant time operation.

Since BRGC-RESTRICT (Fig. 3) is loopless, we have a loopless algorithm to list all minimal SBRs for a given chain. For cross-product of chains (see Theorem 1) we apply procedure BRGC (Fig. 2).

**Theorem 3.** *A listing of all minimal SBRs for a given integer  $n$  can be generated by a loopless algorithm.*

## 6 A Brief History of Signed Binary Representations

Signed-digit representations have been investigated by both mathematicians and computer scientists (see Hwang [11], Parhami [16] and Knuth [13]). Signed-binary representations using the digits  $\{-1, 0, 1\}$  were first investigated by Reitwiesner [19] and Avizienis [2] in the context of digital hardware. Reitwiesner presented an algorithm for identifying the *canonical* signed-binary representation, which is that representation in which no two adjacent digits are non-zero. Over the years, similar algorithms have been re-discovered by several authors (Chang and Tsao-Wu [6], Jedwab and Mitchell [12] and Prodinger [18]). A technique for identifying *all* minimal signed-binary representations, not just the canonical, was discovered by Ganesan and Manku [8]. Sawada [21] adapted this technique to list all minimal SBRs in Gray code order in constant amortized time.

The average weight of minimal signed-binary representations of  $b$ -bit numbers approaches  $b/3$  for large  $b$ . This result has been re-discovered several times, using different proof techniques (Reitwiesner [19], Arno and Wheeler [1], Prodinger [18] and Ganesan and Manku [8]).

Sloane and Plouffe's sequence M0103 and Sloane's sequence A007302 correspond to the weights of minimal signed-binary representations of natural numbers. Sloane's Sequence A005578 are numbers  $n$  at which the weight of minimal signed-binary representations of  $n$  increases. Sloane's sequence A057526 is the number of zeros in minimal signed-binary representations of natural numbers.

For  $m \geq 2$ ,  $(\dots a_2 a_1 a_0)_m$  is said to be a "signed-digit representation" of  $n$  if  $n = \sum_{k \geq 0} a_k m^k$  and  $m_k \in \{0, \pm 1, \pm 2, \dots, \pm (m-1)\}$ . A minimal representation

has the least number of non-zero digits. The general case  $m \geq 2$  has appeared in early work by Avizienis [2]. Clark and Liang [7] defined a *canonical* representation as one satisfying two additional constraints: (a)  $|a_{i+1} + a_i| < m$  for all  $i$ , and (b)  $|a_i| < |a_{i+1}|$ , if  $a_{i+1}a_i < 0$ , where  $|a_i|$  denotes the absolute value of  $a_i$ . Such a representation is also known as a *generalized non-adjacent form* (GNAF) since it possesses the property that no two consecutive digits are non-zero for  $m = 2$ . The GNAF for any integer is minimal and unique. An algorithm for identifying the GNAF was presented in [7]. The average weight for  $b$ -digit numbers was shown to be asymptotically  $\frac{m-1}{m+1}b$  by Arno and Wheeler [1]. Wu and Hasan [26] derive closed-form formulae for the same. These results were re-discovered by Ganesan and Manku [8].

## 6.1 Fast Exponentiation

Fast computation of  $x^n \bmod r$  is very valuable in cryptography (see surveys by Koç [14] and Gordon [9]). Exponentiation can be studied in terms of addition chains and addition-subtraction chains.

An addition chain for integer  $n$  is a sequence of values  $a_0 = 1, a_1, a_2, \dots, a_r = n$  with the property that for each  $i > 0$ , there exist  $j$  and  $k$  such that  $a_i = a_j + a_k$ . Then  $x^n$  can be computed with  $r$  multiplications. See Knuth [13] for a survey of addition chains. The best known lower-bound is  $\log_2 n + \log_2 H(n) - 2.13$  by Schönhage [22]. An upper bound for the length of addition chains is  $\lfloor \log_2 n \rfloor + H(n)$ , where  $H(n)$  denotes the Hamming weight of  $n$  (the number of 1-bits in binary representation of  $n$ ). The upper bound is realized by the folklore “fast-multiplication algorithm”. For a randomly chosen  $b$ -bit exponent,  $b/2$  bits are 1 on average; so the expected number of multiplications is  $3b/2$ . Several papers propose heuristics for reducing the average by discovering short addition chains (see Bos and Coster [4] and Yacobi [27], for example).

For evaluating  $x^n \bmod r$  when  $x$  and  $r$  are fixed *a priori*, we can pre-compute  $x^{-1} \bmod r$ , enabling efficient “division” as well. Further, in elliptic curve cryptography, computing  $x^{-1} \bmod r$  is as costly as computing  $x \bmod r$ . This leads us to the idea of addition-subtraction chains (described below), which reduces the average number of multiplications far below  $3b/2$ .

An addition-subtraction chain for integer  $n$  is a sequence of values  $a_0 = 1, a_1, a_2, \dots, a_r = n$  with the property that for each  $i > 0$ , there exist  $j$  and  $k$  such that  $a_i = \pm a_j \pm a_k$ . Then  $x^n$  can be computed with  $r$  multiplications/divisions. Signed-binary representations correspond to addition-subtraction chains. For  $b$ -bit exponents, approximately  $b/3$  bits are  $\pm 1$ ; so the average number of multiplications/divisions is roughly  $4b/3$ . Higher bases lead to further savings.

Addition-subtraction chains are useful for fast exponentiation in groups (Wu and Hasan [25], Brickell *et al* [5]). Their usefulness in elliptic curve cryptography was first pointed out by Morain and Olivos [15]. Conversion of an integer in binary to its minimal signed-digit representation is popularly known as *recoding*. Efficient software/hardware implementation of recoding presents its own unique challenges. This has led to a variety of recoding algorithms and gener-

alizations of signed-digit representations by the cryptography community. For a good overview of recoding literature, see Phillips and Burgess [17].

## 6.2 Routing in Chord and CM-2

Weitzman [24] studied routing in the Connection Machine CM-2, developed by Thinking Machines in 1980s. CM-2 was a massively parallel computer using a hypercube-based inter-connection network for routing. Every processor could send a message to another processor a fixed distance  $\pm 2^i$  away for any  $i \geq 0$ . Weitzman discovered that  $F(n)$ , the optimal cost of communication between two processors distance  $n$  away, was given by  $F(0) = 0$ ,  $F(2^k) = 1$  and  $F(n) = 1 + \min(F(n - 2^k), F(2^{k+1} - n))$ , for  $2^k < n < 2^{k+1}$ . The relationship between  $F(n)$  and signed-binary representations was exposed by Ganesan and Manku [8]. They studied a peer-to-peer routing network called Chord [23]. In its simplest form, Chord is an undirected graph on  $2^b$  nodes arranged in a circle, with edges connecting pairs of nodes that are  $2^k$  positions apart for any  $k \geq 0$ . The shortest path for clockwise distance  $d$  can be identified by computing a minimal signed-binary representation of  $d'$  defined as follows [8]:

$$d' = \begin{cases} d & \text{if } d \leq \lfloor 2^b/3 \rfloor \\ 2^b - d & \text{if } d > \lfloor 2^{b+1}/3 \rfloor \\ d \text{ or } 2^b - d & \text{otherwise} \end{cases}$$

1 and  $\bar{1}$  in the signed-binary representation correspond to clockwise and anti-clockwise traversals of Chord edges respectively. A variety of algorithms for solving the problem are presented in [8]. One of them is “LEFT-TO-RIGHT BIDIRECTIONAL GREEDY”, which is identical to Weitzman’s algorithm.

## References

1. Steven Arno and Ferrell S Wheeler. Signed digit representations of minimal hamming weight. *IEEE Transactions on Computers*, 42(8):1007–1010, August 1993.
2. Algirdas A Avizienis. Signed-digit number representations for fast parallel arithmetic. *IRE Transactions on Electronic Computers*, 10:389–400, 1961.
3. James R Bitner, Gideon Ehrlich, and Edward M Reingold. Efficient generation of the binary reflected Gray code and its applications. *Communications of the ACM*, 19(9):517–521, September 1976.
4. J Bos and M Coster. Addition chain heuristics. In *Advances in Cryptology: CRYPTO 89 (LCNS No 435)*, pages 400–407, 1989.
5. E F Brickell, D M Gordon, K S McCurley, and D B Wilson. Fast exponentiation with precomputation. In *Proc. EUROCRYPT '92*, pages 200–207, 1992.
6. S H Chang and N Tsao-Wu. Distance and structure of cyclic arithmetic codes. In *Proc. Hawaii International Conference on System Sciences*, volume 1, pages 463–466, 1968.
7. W E Clark and J J Liang. On arithmetic weight for a general radix representation of integers. *IEEE Transactions on Information Theory*, 19:823–826, November 1973.

8. Prasanna Ganesan and Gurmeet Singh Manku. Optimal routing in Chord. In *Proc. 15th ACM-SIAM Symposium on Discrete Algorithms (SODA 2004)*, pages 169–178, January 2004.
9. Daniel M Gordon. A survey of fast exponentiation methods. *J of Algorithms*, 27(1):129–146, April 1998.
10. F Gray. Pulse code communications. U S Patent 2,632,058 (March 17, 1953), 1953.
11. Kai Hwang. *Computer Arithmetic: Principles, Architecture and Design*. John Wiley and Sons, Inc., 1979.
12. J Jedwab and C J Mitchell. Minimum weight modified signed-digit representations and fast exponentiation. *Electronic Letters*, 25(17):1171–1172, 1989.
13. Donald E Knuth. *Seminumerical Algorithms*, volume 2 of *The Art of Computer Programming*. Addison-Wesley, Reading, Massachusetts, 3 edition, 1997.
14. Çetin Kaya Koç. High-speed RSA implementation. RSA Labs, November 1994.
15. François Morain and Jorge Olivos. Speeding up the computations on an elliptic curve using addition-subtraction chains. *RAIRO Informatique Théorique et Applications*, 24(6), 1990.
16. B Parhami. Generalized signed-digit number systems: A unifying framework for redundant number representations. *IEEE Transactions on Computers*, 39:89–98, 1990.
17. Braden Phillips and Neil Burgess. Minimal weight digit set conversions. *IEEE Transactions on Computers*, 53(6):666–677, June 2004.
18. Helmut Prodinger. On binary representations of integers with digits  $-1, 0, 1$ . *INTEGER: The Electronic Journal of Combinatorial Number Theory*, 0, 2000.
19. G W Reitwiesner. Binary arithmetic. *Advances in Computers*, 1:231–308, 1960.
20. Carla Savage. A survey of combinatorial Gray codes. *SIAM Review*, 39(4):609–625, 1997.
21. Joe Sawada. A Gray code for binary subtraction. In *2nd Brazilian Symposium on Graphs, Algorithms and Combinatorics (GRACO 2005)*, 2005.
22. Schönhage. A lower bound for the length of addition chains. *Theoretical Computer Science*, 1:1–12, 1975.
23. Ion Stoica, Robert Morris, D Liben-Lowell, David R Karger, M Frans Kaashoek, F Dabek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup protocol for Internet applications. *IEEE/ACM Transactions on Networking*, 11(1):17–32, 2003.
24. A Weitzman. Transformation of parallel programs guided by micro-analysis. In B Salvy, editor, *Algorithms Seminar, 1992-1993*, pp. 155–159, Institut National de Recherche en Informatique et en Automatique, France, Rapport de Recherche, No. 2130 (Summarized by Paul Zimmermann), 1993.
25. H Wu and M A Hasan. Efficient exponentiation of a primitive root in  $GF(2^m)$ . *IEEE Transactions on Computers*, 46(2):162–172, February 1997.
26. Huapeng Wu and M Anwar Hasan. Closed-form expression for the average weight of signed-digit representations. *IEEE Transactions on Computers*, 48(8):848–851, August 1999.
27. Yacov Yacobi. Exponentiating faster with addition chains. In *Advances in Cryptography – EUROCRYPT 90: Workshop on the Theory and Application of Cryptographic Techniques*, page 222, 1990.