Gray Codes for Reflectable Languages

Yue Li* Joe Sawada†

March 18, 2008

Abstract

We classify a type of language called a reflectable language. We then develop a generic algorithm that can be used to list all strings of length n for any reflectable language in Gray code order. The algorithm generalizes Gray code algorithms developed independently for k-ary strings, restricted growth strings, and k-ary trees, as each of these objects can be represented by a reflectable language. Finally, we apply the algorithm to open meanderic systems which can also be represented by a reflectable language.

1 Introduction

The term *Gray code* originally referred to a specific ordering of length n binary strings, patented by Frank Gray [4], where consecutive bitstrings differ by a single bit. The following is an example of this listing for $n = 3$ where the bit that differs with the previous element is underlined:

$$000,\ 00\underline{1},\ 0\underline{1}1,\ 01\underline{0},\ \underline{1}10,\ 11\underline{1},\ 1\underline{0}1,\ 10\underline{0}.$$

Today, the term Gray code refers more generally to an exhaustive listing of any combinatorial object where each successive object differs by some constant amount. In 1997, Savage [9] surveyed many of these Gray code algorithms and since then many more have been developed for various combinatorial objects. For a significant number of these algorithms, a common strategy of *reflecting* subtrees is applied - a technique that is similar in spirit to the original binary reflected Gray code [4].

To illustrate this notion of reflecting subtrees we consider the set S_3 of all length 3 strings over the alphabet $\{a, b, c\}$ with no bb substring. A straight forward recursive algorithm can be used to generate such strings in lexicographic order by following a computation tree like the one in Figure 1. In the computation tree, each leaf represents a unique string that is

*Computing and Information Science, University of Guelph, Canada. email: yli04@uoguelph.ca

†Computing and Information Science, University of Guelph, Canada. Research supported by NSERC. email: jsawada@uoguelph.ca

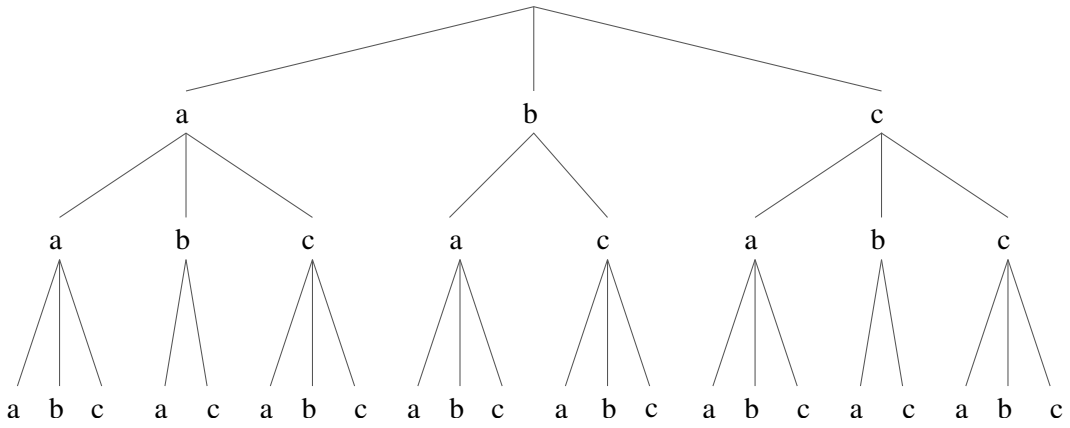

Figure 1: Computation tree to generate S_3 in lexicographic order.

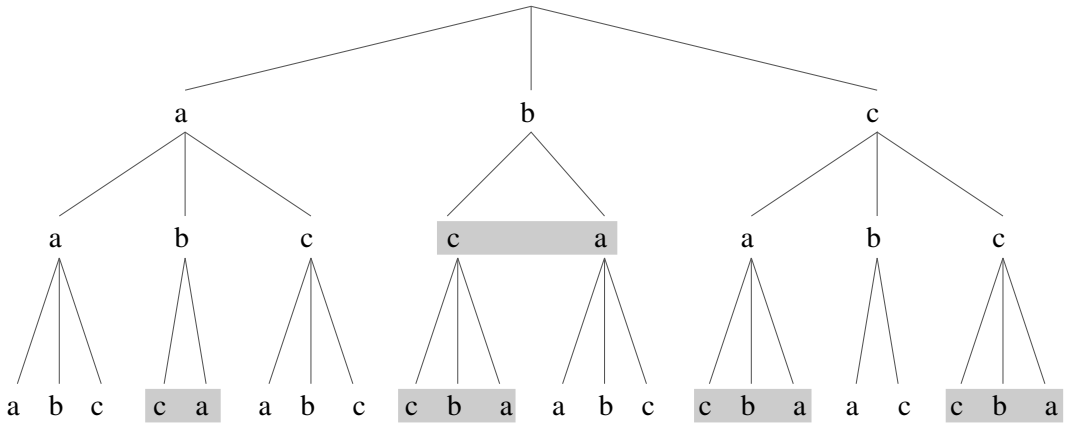

Figure 2: Computation tree to generate S_3 in Gray code order.

obtained by tracing the path from the root to the leaf. Observe that the resulting listing is not a Gray code since in some cases successive strings may differ in all positions. However, by reflecting (reversing) the order of the children at particular nodes, as illustrated in Figure 2, we can obtain an ordering of S_3 that is a Gray code.

Some objects for which this strategy has been applied to include: binary [4] and k-ary strings [5, 13], restricted growth functions [3, 7] and tails [8], and binary [11] and k-ary trees [12, 14]. In this paper we generalize what various representations for these objects have in common by introducing the notion of reflectable languages. We then provide a generic algorithm that can generate all length n words for any reflectable language in Gray code order. As a new application, we apply the algorithm to open meandric systems.

2 Reflectable Languages

Definition 2.1. *A language L over the alphabet Σ is said to be* reflectable *if for every $i > 1$ there exists two characters x_i and y_i in Σ such that if $w_1 w_2 \cdots w_{i-1}$ is a prefix of a word in L then both $w_1 w_2 \cdots w_{i-1} x_i$ and $w_1 \cdots w_{i-1} y_i$ are also prefixes of words in L.*

Recall the language S_3 defined in the previous section as the set of all length 3 strings over $\{a, b, c\}$ with no bb substring. It is reflectable by considering $x_i = a$ and $y_i = c$. As another example, consider the language $L = \{a, aa, ac, aaa, aab, aac, aca, acb, acd\}$ over $\Sigma = \{a, b, c, d\}$. Observe that this language is also reflectable by considering $x_2 = a, y_2 = c$ and $x_3 = a, y_3 = b$.

On the other hand, the language $L' = \{a, aa, ac, aaa, aab, aac, abc, aca, acb\}$ is not reflectable since $abc \in L'$, but no matter what we use for x_3 and y_3 the strings abx_3 and aby_3 can not both be in L'.

It turns out that many common combinatorial objects can be represented by reflectable languages. In the following subsections we give examples of such reflectable languages by demonstrating their x_i and y_i values.

2.1 Binary strings, k-ary strings, and variants

Letting $\Sigma = \{0, 1, 2, \ldots, k-1\}$, the set of all k-ary strings for $k \geq 2$ trivially form a reflectable language by considering $x_i = 0$ and $y_i = 1$.

A generalization of k-ary strings is to consider elements of the product space $S = S_1 \times S_2 \times \cdots \times S_n$ where each $S_i = \{0, 1, \ldots, r_i - 1\}$ for $i = 1, 2, \ldots, n$. If each $r_i \geq 2$, then such a space will again correspond to a reflectable language by considering $x_i = 0$ and $y_i = 1$.

Strings with a forbidden substring α are a variation on k-ary strings that have been studied in [10]. If the forbidden substring α is composed from a subset of $k - 2$ characters in the alphabet, then the language of strings with forbidden substring α is reflectable. This follows from the definition by assigning any two characters from the alphabet that do not appear in α to x_i and y_i respectively. As an example, recall that S_3 corresponds to a reflectable language.

2.2 Restricted growth strings

Restricted growth strings are strings of non-negative integers $w_1 \cdots w_n$ satisfying $w_1 = 0$ and $w_i \leq 1 + max\{w_1, w_2, \ldots, w_{i-1}\}$. There is a well known bijection between restricted growth strings and set partitions [9]. By letting $x_i = 0$ and $y_i = 1$ for each i, observe that restricted growth strings of length n are reflectable.

A slight generalization of restricted growth strings are *restricted growth tails* [8], which are strings of non-negative integers $w_1 \cdots w_n$ satisfying $w_1 \leq k$ and $w_i \leq 1 + max\{w_1, w_2, \ldots, w_{i-1}, k - 1\}$. Observe that this generalization also corresponds to a reflectable language by letting $x_i = 0$ and $y_i = 1$.

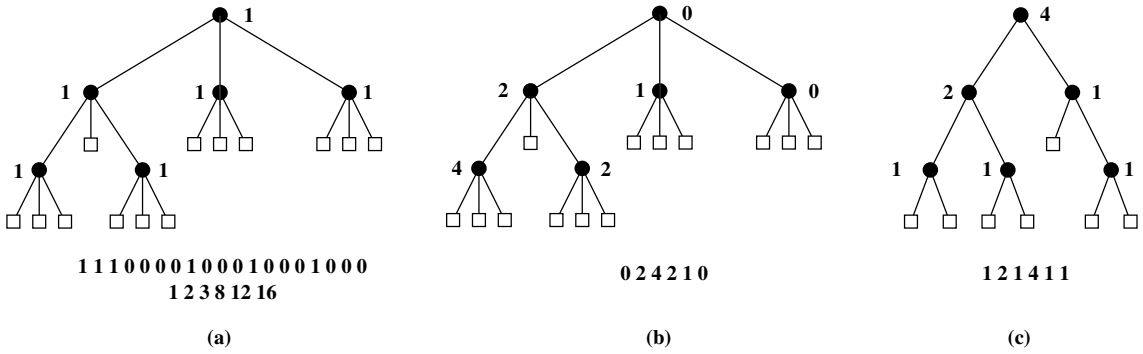

Figure 3: (a) Zaks representation $n = 6$, $k = 3$. (b) Right distance sequence $n = 6$, $k = 3$. (c) Weight sequence for a binary tree with $n = 6$.

2.3 Binary and k-ary trees

A k-ary tree is a tree where each internal node has k ordered subtrees. A common bit-sequence representation for k-ary trees is obtained by visiting a tree in pre-order where a 1 is assigned to each internal node and a 0 is assigned to each leaf. This representation is often attributed to Zaks [15]. Using this pre-order traversal, a unique bit-sequence of length $kn+1$ is obtained for each k-ary tree with n internal nodes. Since there are so many zeros in the bit sequence representation, it is often useful to use an alternate representation where only the positions of the ones are recorded. For example, a tree and its bit sequence representation is given in Figure 3(a). Using this representation a sequence $a_1 a_2 \cdots a_n$ will correspond to a k-ary tree if $a_1 = 1$ and for each $i > 0$ we have $a_{i-1} < a_i \leq k(i-1)+1$ [12]. Observe that such a set such strings corresponds to a reflectable language where $x_i = k(i-1)$ and $y_i = k(i-1)+1$.

Another representation is presented in [14] where the nodes in a k-ary tree are defined recursively as follows: the root node is assigned 0, then for each child $i = 1, \ldots, k$ from left to right we assign the value $k - i$ plus the value of its parent. A pre-order traversal yields what is called the right-distance sequence for a k-ary tree. An example is illustrated in Figure 3(b). Using this representation, any sequence corresponding to a tree with $n \geq 1$ nodes can be extended to a tree with $n+1$ nodes by appending any value between 0 and $k-1$ [14]. Thus, the set of all right distance sequences is a reflectable language where $x_i = 0$ and $y_i = 1$.

In the special case of binary trees ($k = 2$), we can assign a weight to each vertex corresponding to the number of leaves in its left subtree. The sequence that results by recording these weights via an in-order traversal is the weight sequence introduced by Pallo [6]. An example is illustrated in Figure 3(c). Pallo also shows a sequence $w_1 w_2 \ldots w_n$ will correspond uniquely to the weight sequence for a binary tree with n nodes if for each i we have $1 \leq w_i \leq i$, and for each $i - w_i + 1 \leq j \leq i$ we have $i - w_i \leq j - w_j$. Notice that this set of weight sequences corresponds to a reflectable language where $x_i = 1$ and $y_i = i$.

```
procedure GrayCode (t)
    if (t > n) then Process(w)
    else
        r := w_t                    //w_t WILL BE EITHER x_t OR y_t
        GrayCode(t+1)

        for each z ∈ Σ−{x_t, y_t} such that w_1 ··· w_{t−1} z is a prefix of some word in L_n
            w_t := z
            GrayCode(t+1)

        if (r = x_t) then w_t := y_t
        else w_t := x_t
        GrayCode(t+1)
end
```

Figure 4: Algorithm, GrayCode(t), to list all words of length n from a reflectable language L in Gray code order.

2.4 Open meandric systems

We demonstrate that open meandric systems can be represented by a reflectable language in Section 4.

3 A Simple Gray Code Algorithm

In this section, we present a simple recursive algorithm to list L_n (words of length n for some reflectable language L) in Gray code order. This algorithm generalizes independently developed algorithms for: binary strings [4], k-ary strings and cross products [5, 13], restricted growth strings [3, 7], restricted growth tails [8], and binary [11] and k-ary trees [12, 14]. These individual algorithms are still of interest, however, since they often include extra efficiency considerations that are specific to each object. For instance, adding data structures to make the algorithm run in constant amortized time or demonstrating a loop-free implementation.

The basic idea behind the generic recursive algorithm is to apply the simple idea of reflecting particular subtrees that was discussed earlier. To do this, we use the special characters x_i and y_i as the first and last children of each node at level $i-1$. The order of the other characters (children) does not matter. This way, at the start of each recursive call we can be sure that that the previous word generated had either the character x_i or y_i at position i. Pseudocode is shown in Figure 4. The word being generated is stored in $w = w_1 w_2 \cdots w_n$. The current position is given by the parameter t and the variables x_i and y_i are specific to the reflectable language under consideration as described in Definition 2.1. To run the algorithm $w_2 \cdots w_n$ is initialized to $x_2 x_3 \cdots x_n$, then for each $z \in \Sigma$ that starts a word in L_n we assign $w_1 := z$ and call GrayCode(2).

To illustrate the algorithm, again recall the language S_3 which consists of all length 3 strings over the alphabet $\{a, b, c\}$ with no bb substring. By applying $x_i = a$ and $y_i = c$, the computation tree that results from applying algorithm GrayCode(t) is given in Figure 2.

Theorem 3.1. *For any reflectable language L and given integer n, the algorithm GrayCode(t) will produce a list of all words L_n in Gray code order.*

PROOF: First, it should be clear that every word in L_n is generated exactly once by the algorithm. Thus, we need only show that successive words in the resulting listing differ in exactly one position. Consider any two such successive words $\alpha = a_1 \cdots a_n$ and $\beta = b_1 \cdots b_n$. Suppose that their first common ancestor in the computation tree is at level i. Then $a_1 \cdots a_i = b_1 \cdots b_i$ and $a_{i+1} \neq b_{i+1}$. Now since the first recursive call in GrayCode(t) does not change the value for current position t and since the last recursive call never leads to a dead end, we must have $a_{i+2} \cdots a_n = b_{i+2} \cdots b_n$. Thus, every pair of successive words generated will differ in exactly one position, i.e., the listing is a Gray code. □

3.1 Analyzing the generic Gray code algorithm

To analyze the generic Gray code algorithm GrayCode(t), we perform an amortized analysis comparing the running time to the number of objects generated. For this analysis, the best we can hope for is an algorithm that runs in constant amortized time. One assumption made in this analysis is that the time taken by the function Process(w) is constant, since for many applications this may be the case.

Now, focusing on the algorithm itself, observe that every non-leaf node in the computation tree has at least two children since recursive calls must be made for both x_t and y_t. Also, observe from the prefix test in the **for** loop that there will be no dead-ends, which means that every leaf will be at level n of the computation tree and will correspond to a word in L_n. Thus, if the time taken at each internal node of computation is proportional to the number of children (recursive calls made), then the overall running time will be proportional to the number of nodes in the computation tree. Since the branching factor of each internal node is at least 2, the number of leaves (words generated) will be greater than the number of internal nodes in which case the algorithm will run in constant amortized time.

Theorem 3.2. *The algorithm GrayCode(t) runs in constant amortized time if the following two conditions hold:*

1. *checking whether or not $w_1 \cdots w_{t-1} z$ is a prefix of some word in L_n takes $O(1)$ time and*

2. *each internal node in the computation tree has $\Omega(|\Sigma|)$ children.*

In several of the examples of reflectable languages seen so for, the alphabet symbols that are possible at certain positions in each word may vary. Thus, in an analysis it may be more appropriate to consider an alphabet Σ_i for each level $1 \leq i \leq n$ of the computation tree. The size of these alphabets could then be applied to Theorem 3.2 rather than the more general alphabet Σ.

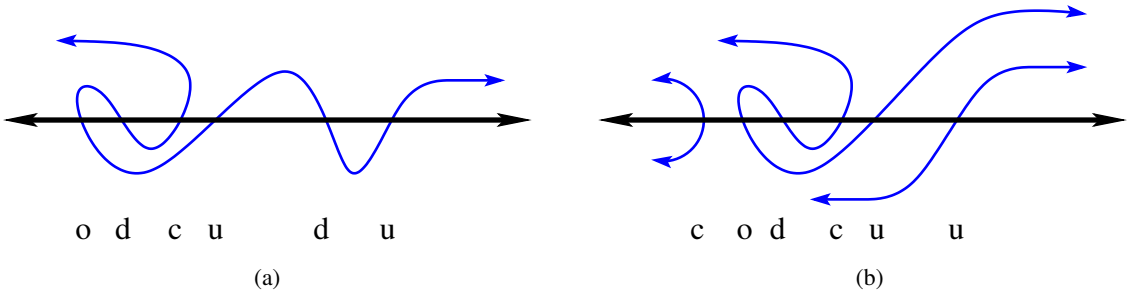

Figure 5: (a) An open meander of order 6. (b) An open meandric system of order 6 with 3 curves.

4 A New Application: Open Meandric Systems

An *open meander* can be thought of as an infinite meandering river which passes beneath a series of bridges of an infinite straight road going from west to east. For example, Figure 5(a) illustrates an open meander with 6 bridges. Using a curve to represent the river and a line to represent the road, we can generalize the notion of an open meander by allowing multiple non-intersecting curves to cross the line. Such a configuration is called an *open meandric system* (OMS). The *order* of an OMS is defined to be the number of times the curves cross the line. An example of an open meandric system of order 6 with 3 curves is shown in Figure 5(b) .

Enumeration sequences for open meandric systems were studied by Bacher [1] and a fast algorithm for generating open meandric systems appears in [2]. The latter paper uses the alphabet $\Sigma = \{$u,d,o,c$\}$ to represent the four different types of crossings: u=up, d=down, o=open, c=close. Using this representation, each OMS of order n can be represented uniquely by a word of length n over Σ as illustrated in Figure 5. Observe that any OMS of order n can always be extended into an OMS of order $n+1$ by appending either an o, d, or u. Only when adding a c to an existing OMS is it possible that that a previously open curve can become closed [2]. Thus, the language of all OMSs is reflectable by setting x_i=d and y_i=u.

The algorithm in [2] to generate all OMSs runs in constant amortized time due to the introduction of data structures that can test when it is possible to append a c to an existing OMS in constant time. Therefore, it is possible to directly apply our generic Gray code algorithm to convert their lexicographic algorithm into a Gray code algorithm. By applying the same data structures the resulting Gray code algorithm will achieve the same asymptotic running time as the original algorithm. The result also follows from Theorem 3.2.

Corollary 4.1. *A Gray code for open meandric systems of order n can be generated in constant amortized time.*

As an illustration of the resulting Gray code algorithm, we show a partial computation tree for $n = 4$ in Figure 6 . The dead ends are shown by the dotted edges. For example, the words oc, uoc and oudc are all invalid OMSs because they include a closed curve.

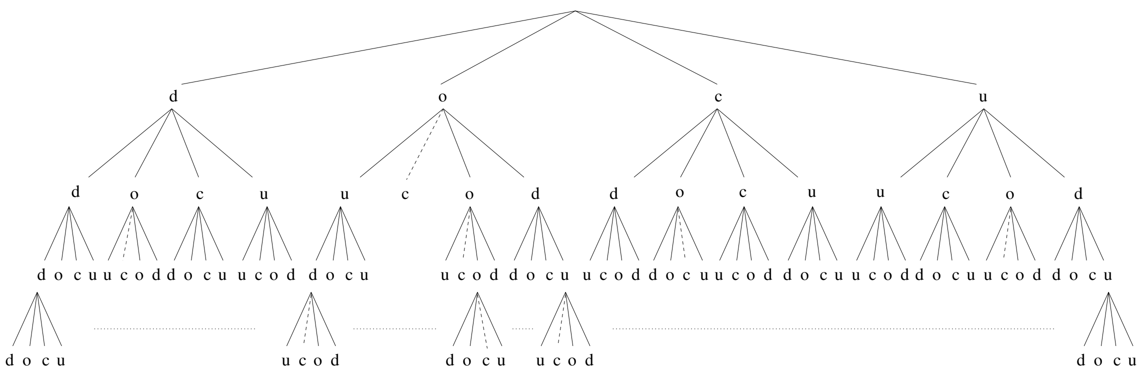

Figure 6: The partial tree for OMS of order 4 that results from applying the generic Gray code algorithm with x_i=d and y_i=u.

References

[1] R. Bacher. Meander algebras. *Prepublication de l'Institut Fourier*, 478, 1999.

[2] B. Bobier and J. Sawada. A fast algorithm to generate open meandric systems. *Manuscript https://www.cis.uoguelph.ca/pubs/meander.pdf*, 2007.

[3] G. Ehrlich. Loopless algoirthms for generating permutations, combinations, and other combinatorial configurations. *Journal of the ACM*, 20:500–513, 1973.

[4] F. Gray. Pulse code communication. *U.S. Patent*, 2632058, 1953.

[5] G. Manku and J. Sawada. A loopless Gray code for minimal signed-binary representations. In *Proc. 13th Annual European Symposium on Algorithms (ESA 2005) LNCS*, pages 438–447, Oct 2005.

[6] J. M. Pallo. Enumerating, ranking and unranking binary trees. *The Computer Journal*, 29(2):171–175, 1986.

[7] F. Ruskey. *Combinatorial Generation.* Manuscript, 2001.

[8] F. Ruskey and C. Savage. Gray codes for set partitions and restricted growth tails. *Australasian Journal of Combinatorics*, 10:85–96, 1994.

[9] C. Savage. A survey of combinatorial Gray codes. *SIAM Review*, 39(4):605–629, 1997.

[10] M. Squire. Gray codes for a-free strings. *Electronic Journal of Combinatorics*, 3(1), 1996.

[11] V. Vajnovszki. On the loopless generation of binary tree sequences. *Information Processing Letters*, 68:113–117, 1998.

[12] D. Roelants van Baronaigien. A loopless Gray-code algorithm for listing k-ary trees. *Journal of Algorithms*, 35:100–107, 2000.

[13] S. Williamson. *Combinatorial for Computer Science*. Computer Science Press, 1985.

[14] R. Wu, J. Chang, and Y. Wang. Ranking, unranking and loopless Gray-codes generation of t-ary trees. *Manuscript*, 2007.

[15] S. Zaks. Generation and ranking or k-ary trees. *Information Processing Letters*, 14(1):44–48, 1982.