Efficient Oracles for Generating Binary Bubble Languages

J. Sawada[*]
School of Computer Science
University of Guelph, Canada
jsawada@uoguelph.ca

A. Williams[†]
School of Computer Science
University of Guelph, Canada
haron@uvic.ca

Submitted: Oct 14, 2010; Accepted: Feb 3, 2012; Published: Feb 23, 2012
Mathematics Subject Classification: 68W32, 05A99

Abstract

A simple meta-algorithm is provided to efficiently generate a wide variety of combinatorial objects that can be represented by binary strings with a fixed number of 1s. Such objects include: k-ary Dyck words, connected unit interval graphs, binary strings lexicographically larger than ω, those avoiding 10^k for fixed k, reversible strings and feasible solutions to knapsack problems. Each object requires only a very simple object-specific subroutine (oracle) that plugs into the generic cool-lex framework introduced by Williams. The result is that each object can be generated in amortized $O(1)$-time. Moreover, the strings can be listed in either a conventional co-lexicographic order, or in the cool-lex Gray code order.

Keywords: Bubble language, Gray code, cool-lex, unit interval graph, knapsack, reversible strings, CAT algorithm, necklace, Lyndon word

1 Introduction

The exhaustive generation of combinatorial objects has become an important area of algorithmic research and is a major theme in Knuth's latest volume of The Art of Computer Programming [3, 4, 5]. In the amortized sense, the best performance one can achieve for a generation algorithm

[*]Research supported by NSERC
[†]Research supported by Roncesvalles Post-doc

is one that runs in Constant Amortized Time. Such an algorithm is said to be CAT. In addition to performance, it is often desirable to have successive elements in a listing differ by only a constant amount of change. Such algorithms are called *combinatorial Gray codes*; the objects are said to be listed in *Gray code order*.

One of the major drawbacks in this research area is a shortage of general frameworks that unite generation algorithms for a wide variety of objects. This is not unusual since focusing on the specific properties of each object is often required to make an algorithm efficient. A rare exception is the theory outlined in [7], which describes a large collection of binary *bubble languages* that all share a common property (described in Section 2). Examples of bubble languages include: combinations, necklaces, Lyndon words, Dyck words and connected unit interval graphs. In each case, the binary strings representing the objects have a fixed *density* or *weight* – the number of 1s in the strings are fixed. One of the primary results in that paper is the development of a framework to exhaustively list the strings for any bubble language in Gray code order. All that is needed for each bubble language is a unique subroutine that computes the bubble lower bound described in Section 2. Using the framework, the strings can be output in either a standard co-lexicographic order or in a newly defined *cool-lex* order.

This paper builds on the recursive cool-lex framework [7] by providing efficient object-specific subroutines that we call *oracles* for the bubble languages listed in Section 2. For each such object, excepting necklaces and Lyndon words, the result is a generation algorithm that runs in constant amortized time. Efficient oracles for necklaces and Lyndon words are provided in [10]; they are significantly more complex than the ones given in this paper. By applying the cool-lex framework and the necklace oracle, an efficient construction of de Bruijn cycles for fixed-weight binary strings is given in [8].

The remainder of the paper is organized as follows. In Section 2, bubble languages are defined and examples are provided. In Section 3, we recall the recursive cool-lex framework from [7] and add an alternate string representation (run-length blocks) that will be useful in some of our oracles. In Section 4, we provide a generic oracle for any bubble language with a membership tester. It is illustrated with necklaces and Lyndon words. In Section 5, we explicitly provide efficient oracles for many fixed-density bubble languages. In most cases the oracles require only several lines of code to implement and in all cases they immediately yield CAT algorithms for their respective language. In Section 6 we outline how to obtain Gray codes for the objects in constant amortized time when the density restriction is removed. We conclude the paper with brief summary in Section 7.

2 Bubble languages

A binary language **L** is said to be a *bubble language* if it satisfies one of the following properties [7]:

first-01: if $\alpha \in \mathbf{L}$, then swapping its first 01 (if it exists) to 10 yields another string in **L**, or

first-10: if $\alpha \in \mathbf{L}$, then swapping its first 10 (if it exists) to 01 yields another string in **L**.

Some interesting examples of bubble languages include:

first-01 bubble languages
- combinations
- strings with forbidden 01^k
- strings with $\leq k$ inversions from 1^*0^*
- strings with $\leq k$ transpositions from 1^*0^*
- strings $\geq$ some string ω
- strings $>$ or $\geq$ their reversal
- strings $\geq$ their complemented reversal
- necklaces (largest rotation)
- aperiodic necklaces (largest rotation)
- k-ary Dyck words
- ordered forests with $\leq k$ trees
- linear extensions of a B-poset
- connected unit interval graphs
- feasible solutions to 0-1 knapsack.

first-10 bubble languages
- combinations
- strings with forbidden 10^k
- strings with $\leq k$ inversions from 0^*1^*
- strings with $\leq k$ transpositions from 0^*1^*
- strings $\leq$ to some string ω
- strings $<$ or $\leq$ their reversal
- strings $\leq$ their complemented reversal
- necklaces (smallest rotation)
- Lyndon words

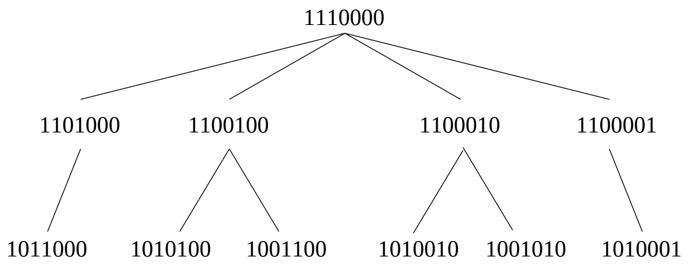
In [7], a framework is provided to generate any bubble language of strings with *fixed-density* (the number of 1s is fixed). The elegance of the generic algorithm is its ability to switch between languages by simply altering a function that computes the "bubble lower bound". Given a string $\alpha = 1^s0^t\gamma$ in a binary language **L**, the *bubble lower bound* is the smallest non-negative integer j such that $1^{s-1}0^{t-j}10^j\gamma \in \mathbf{L}$. To be precise, we assume γ is empty or begins with 1, and $t > 0$. These two restrictions ensure that α can be written uniquely as $1^s0^t\gamma$.

Example. The following first-01 bubble language consists of all strings of length 7 and density 3 that are greater than or equal to $\omega = 1001010$. The strings are listed in co-lex order as illustrated in Figure 1.

$$\begin{aligned}\mathbf{L} \;=\; \{\, & 1110000, 1101000, 1011000, 1100100, 1010100, 1001100, 1100010, \\ & 1010010, 1001010, 1100001, 1010001 \,\}\end{aligned}$$

Consider the string $\alpha = 1100010$ where $s = 2$, $t = 3$, and $\gamma = 10$. The bubble lower bound for α is 1 since (i) $10010\gamma \geq \omega$ and (ii) $10001\gamma < \omega$.

The generic algorithm from the framework can list a bubble language's strings in either co-lex order or the cool-lex Gray code where successive strings differ by at most 2 swaps. However, the



Pre−order	Post−order
1110000	1011000
1101000	1101000
1011000	1010100
1100100	1001100
1010100	1100100
1001100	1010010
1100010	1001010
1010010	1100010
1001010	1010001
1100001	1100001
1010001	1110000
co−lex	**cool−lex Gray code**

Figure 1: Computation tree for strings of length 7 and density 3 that are greater than or equal to $\omega = 1001010$.

efficiency is dependent on the time required to determine the bubble lower bound for the specific bubble language. As a first step, we provide a generic oracle that applies to any bubble language $\mathbf{L}$ as long as it is provided with a membership tester that determines whether or not a given $\alpha \in \mathbf{L}$. In other words, the problem of producing a Gray code for a bubble language is reduced to providing a membership tester for it. As an application, we apply the generic oracle to fixed-density necklaces and Lyndon words to obtain a $O(n)$ amortized time generation algorithm for these strings. For the remaining bubble languages outlined earlier in this section, we present language specific $O(1)$ time oracles except for the feasible solutions to the knapsack problem; however, for this language a simple analysis shows that the oracle we provide is efficient in the amortized sense. As a result each language can be generated in Constant Amortized Time: the algorithms are CAT.

3 Recursive framework

Given a non-empty *first-01 bubble language* $\mathbf{L}$ consisting of strings with length n and density d, the following recurrence can be used to produce its cool-lex Gray code [7]:

$$\mathbf{C}(s,t,\gamma) = \begin{cases} \mathbf{C}(s-1,1,10^{t-1}\gamma), \mathbf{C}(s-1,2,10^{t-2}\gamma), \ldots, \mathbf{C}(s-1,t-j,10^{j}\gamma), 1^s0^t\gamma & \text{if } s>0 \\ 0^t\gamma & \text{if } s=0 \end{cases}$$

where j is the smallest non-negative integer such that $1^{s-1}0^{t-j}10^j\gamma \in \mathbf{L}$. In this recurrence γ represents a fixed suffix and each recursive term prepends a string of the form 10^i to γ. In particular, $\mathbf{C}(d, n-d, \epsilon)$ will produce the Gray code for $\mathbf{L}$.

Observe that if the first line of the recurrence is altered so the last term $1^s0^t\gamma$ is moved to the front, then we obtain co-lex order. As an illustration consider the computation tree in Figure 1 for strings with length $n = 7$ and density $d = 3$ that are greater than or equal to $\omega = 1001010$. Each node in the computation tree $\alpha = 1^s0^t\gamma$ corresponds to the string that gets output directly from a recursive call to $\mathbf{C}(s,t,\gamma)$. By traversing the tree in post-order, we obtain the cool-lex Gray code. If the tree is traversed in pre-order, we obtain co-lex order.

```
procedure GenBubble($s, t$: int)
int $i, j$
    if $s > 0$ and $t > 0$ then
        $j :=$ Oracle($s, t$)
        for $i := t - 1$ downto $j$ do
            Swap($a_s$, $a_{s+t-i}$)
            GenBubble($s - 1, t - i$)
            Swap($a_s$, $a_{s+t-i}$)
    Visit()
end.
```

Figure 2: Simple recursive algorithm to list all strings in the bubble language **L** in cool-lex Gray code order.

The simple recursive algorithm GenBubble(s, t) shown in Figure 2 produces the cool-lex Gray code for a given first-01 bubble language **L**. The string $\alpha = a_1 \cdots a_n$ that is visited during each recursive call is initialized to $1^d 0^{n-d}$ and is maintained globally. The function Oracle(s, t) determines the oracle lower bound of the current string $\alpha = 1^s 0^t \gamma$ and it takes s and t as parameters for efficiency. For each iteration of the **for** loop, α is updated by swapping the 1 in position s and the 0 in position $s + t - i$. This single swap effectively prepends the string 10^i to γ as described in the recurrence. Observe that recursive calls are only made if $t > 0$, so there is no need to call an oracle if $t = 0$. This allows for this special case to be omitted from all the oracles described later in this paper. The initial call is GenBubble$(d, n - d)$. To output the strings in co-lex order move the function Visit() from the end of the function GenBubble(s, t) to the beginning.

To produce the cool-lex Gray code for a *first-10 bubble language* simply complement the 0s and 1s in the recurence. Algorithmically this means initializing $\alpha := 0^{n-d} 1^d$ and calling GenBubble$(n - d, d)$.

Since every recursive call of GenBubble(s, t) visits a string in **L**, we obtain the following theorem:

THEOREM 1 *[7] If the total amount of computation required by all calls to Oracle(s, t) for a given bubble language* **L** *is proportional to the number of strings in* **L***, then the algorithm GenBubble(s, t) will generate all strings in* **L** *in $O(1)$ amortized time.*

The main focus of this paper is to apply this theorem by finding $O(1)$ time oracles for a variety of different bubble languages. For fixed-density binary strings with no restrictions (i.e., combinations) the oracle simply returns 0; however most bubble languages will require some additional data structure information in order to obtain an efficient oracle. A common structure is outlined in the next subsection which will also enable us to produce alternate output representations efficiently.

3.1 Run-length blocks

A more compact representation for a binary string is its run-length encoding. In particular, a binary string can be represented by a series *blocks* which are maximal substrings of the form 1^*0^*. Each block B_i is composed of two integers (s_i, t_i) representing the number of 1s and 0s respectively. For example, the string 00011010100011001 can be represented by $B_6B_5B_4B_3B_2B_1 = (0,3)(2,1)(1,1)(1,3)(2,2)(1,0)$ where $s_6 = 0$ and $t_6 = 3$.

To maintain this run-length representation within GenBubble(s, t), let the blocks be stored in $B_c \cdots B_1$ where c denotes the number of blocks required to represent α. Initially $B_1 = (d, n-d)$ and $c = 1$ since the first string is 1^d0^{n-d}. Observe that the values for s_c and t_c equal s and t respectively at the start of each recursive call. There are two cases for updating the blocks. If $i = 0$ and $c > 1$ then a 1 is moved from the leading block (s_c, t_c) to the block $c - 1$: s_{c-1} is incremented and s_c is decremented. Otherwise, a new block of the form 10^i is created: B_c splits into the two blocks $B_{c+1} = (s_c - 1, t_c - i)$ and $B_c = (1, i)$. After making the recursive call these actions must be undone to restore the blocks for the next iteration of the **for** loop. These updates can be accomplished by inserting the following code fragments:

Insert before recursive call	**Insert after recursive call**
if $i = 0$ **and** $c > 1$ **then**	**if** $i = 0$ **and** ($c > 2$ **or** $B_1 \neq (1,0)$) **then**
$s_{c-1} := s_{c-1} + 1$	$s_{c-1} := s_{c-1} - 1$
$s_c := s - 1$	$s_c := s$
else	**else**
$B_c := (1, i)$	$B_{c-1} := (s, t)$
$B_{c+1} := (s-1, t-i)$	$c := c - 1$
$c := c + 1$	

Besides giving a compact representation of the current string, this extra data structure will be critical to the efficiency for some of the oracles described in the following section.

3.2 Shifts and swaps

In addition to outputting the Gray code as a sequence of strings or blocks, it is also possible to output the strings as a sequence of left-shifts of a single bit, or as a sequence of the 1 or 2 swaps that are required to go from one string to the next [7]. An example of these various outputs is given in Table 1. To output these shifts or swaps using only a constant amount of extra time, two extra global variables to and $from$ must be maintained. The value for $from$ is initialized to $d + 1$ and the variables are updated as follows:

Strings	**Blocks**	**Left Shifts**	**Swaps**
1 0 1 1 0 0 0	(1,1) (2,3)	shift(4,2)	swap(2,4)
1 1 0 1 0 0 0	(2,1) (1,3)	shift(3,2)	swap(2,3)
1 0 1 0 1 0 0	(1,1) (1,1) (1,2)	shift(5,2)	swap(4,5) and swap(2,3)
1 0 0 1 1 0 0	(1,2) (2,2)	shift(4,2)	swap(3,4)
1 1 0 0 1 0 0	(2,2) (1,2)	shift(4,2)	swap(2,4)
1 0 1 0 0 1 0	(1,1) (1,2) (1,1)	shift(6,2)	swap(5,6) and swap(2,3)
1 0 0 1 0 1 0	(1,2) (1,1) (1,1)	shift(4,2)	swap(3,4)
1 1 0 0 0 1 0	(2,3) (1,1)	shift(4,2)	swap(2,4)
1 0 1 0 0 0 1	(1,1) (1,3) (1,0)	shift(7,2)	swap(6,7) and swap(2,3)
1 1 0 0 0 0 1	(2,4) (1,0)	shift(3,2)	swap(2,3)
1 1 1 0 0 0 0	(3,4)	shift(7,3)	swap(3,7)

Table 1: Different outputs available to Visit(c) for the cool-lex Gray code of binary strings of length 7 and density 3 that are greater than or equal to $\omega = 1001010$. The shifts and swaps are relative to the previous string in the Gray code cyclicly.

Insert before recursive call	**Insert after recursive call**
if $i < t-1$ **then** $from := s+t-i$ $to := s$	$from := s+t-i$ $to := s$

To output the sequence of left-shifts, the function Visit() can print "shift(from, to)" indicating that the bit in position $from$ gets shifted into position to. To output the swaps a global counter $total$ is maintained to indicate how many strings have already been generated. This allows us to test for a special case when considering the first string. Using this extra variable along with the run-length block data structure, the following code fragment will output the sequence of swaps:

```
**if** $a_{to} = 1$ **or** $total = 0$ **then** Print(" swap($to, from$) ")
**else if** $a_{to+1} = 1$ **then** Print(" swap($from-1, from$) and swap($to, to+s_{c-1}$) ")
**else** Print(" swap($from-1, from$)").
```

4 A generic oracle

In this section we provide a generic oracle that can be applied to any bubble language **L** provided there is a *membership tester* Member(**L**, α) that determines whether or not $\alpha \in \mathbf{L}$. Then we provide a short analysis that shows if the tester requires $O(m)$ time, then the language can be generated in $O(m)$ amortized time. As an example, we apply the generic oracle to necklaces and Lyndon words.

The basic idea behind the generic approach is to test each possible value j to determine the smallest value such that $1^{s-1}0^{t-j}10^j\gamma \in \mathbf{L}$. Two strategies are: (1) start at $j = 0$ and increment

j until we find a string in $\mathbf{L}$ or (2) start at $j = t - 1$ and decrement j until we find a string that does *not* belong to $\mathbf{L}$. Depending on the language, one method may be more efficient than the other. However from an analytical standpoint we choose the second method since every successful membership test can be accounted to a unique string in the language. Pseudocode for this generic oracle can be described as follows:

```
procedure Oracle($s$, $t$: int)
int $j$
    $j := t - 1$
    while $j \geq 0$ and Member($\mathbf{L}$, $1^{s-1}0^{t-j}10^j\gamma$) do $j := j - 1$
    return $j + 1$
end.
```

Note that the string $1^{s-1}0^{t-j}10^j\gamma$ can be obtained from the current string $\alpha = 1^s0^t\gamma$ in constant time by swapping the bits in positions s and $s + t - j$.

THEOREM 2 *If Member($\mathbf{L}$, α) is a membership tester that runs in $O(m)$ time for a given bubble language $\mathbf{L}$, then $\mathbf{L}$ can be generated in cool-lex Gray code order or colex order in $O(m)$ amortized time.*

PROOF: Since each recursive call to Gen(s, t) visits a string in the language, we must show that the total number of membership tests is proportional to the number of strings visited. Clearly each call to the oracle will have at most one *unsuccessful* test which can be mapped to the string α visited during the recursive call in which the oracle was called. Additionally, each *successful* membership test also corresponds to a unique string in the language – one that gets generated within the **for** loop of the current recursive call. Thus, for each string visited, we have accounted at most $O(m)$ work from all oracle calls which implies a $O(m)$ amortized algorithm. $\square$

4.1 Necklaces and Lyndon words

Necklaces are often described as the lexicographically smallest string in an equivalence class of strings under rotation. Using this representation, aperiodic necklaces are known as Lyndon words. Fixed-density necklaces and Lyndon words are both 10-bubble languages. Using the lexicographically largest element as representative, the fixed-density necklaces and aperiodic necklaces are both 01-bubble languages. We will focus on the lexicographically smallest string as representative.

The construction of a constant time oracle for fixed-densiy necklaces and Lyndon words appears to be a very challenging task and we leave it as an open problem. Instead, in Figure 3 we present a $\Theta(m)$ membership tester based on the work of Duval [1]. The tester starts by concatenating two copies of the input string α together and then requires at most a linear scan of the resulting string.

```
function Member(L, $\alpha$) returns boolean
int $i, p$
    for $i := 1$ to $n$ do $a_{n+i} := a_i$
    $a_{2n+1} := -1$
    $i := 2$
    $p := 1$
    while $a_{i-p} \leq a_i$ do
        if $a_{i-p} < a_i$ then $p := i$
        $i := i + 1$
    if $i \leq 2n$ then return FALSE
    if $p < n$ and L = LYNDON WORDS then return FALSE
    return TRUE
end.
```

Figure 3: A (combined) membership tester derived from [1] to determine whether or not α is a necklace or a Lyndon word.

COROLLARY 1 *Fixed-density necklaces and Lyndon words can be generated in either co-lex or cool-lex Gray code order in $O(n)$ amortized time.*

Gray codes for fixed-density necklaces have previously been discovered by Wang and Savage [16] and Ueda [13]; however, these algorithms do not use the lexicographic smallest representative and hence do not apply to Lyndon words. Furthermore, their Gray codes are not cyclic. A recent result [10] provides a more efficient oracle than the one given here which results in a CAT algorithm.

5 Efficient oracles

In this section we detail oracles for the bubble languages listed in Section 1 (except necklaces and Lyndon words). For most languages, extra information needs to be maintained within the recursive framework to obtain an efficient oracle. The information specific to a given language is detailed in the relevant subsection and in each case the information can be maintained in $O(1)$ time. In some cases the oracles will apply to two related languages. For instance, all strings with forbidden substring 01^k is a first-10 bubble language and all strings with forbidden substring 10^k is a first-01 bubble language. The oracle for each language is the same, but the initialization for the algorithm must reflect the correct initial string as noted in Section 3.

Important Assumptions: By the nature of the recurrence in GenBubble(s, t), assume that γ is either empty or begins with 1 (for first-01 bubble languages). Also the cases when $s = 0$ or $t = 0$ are already handled by GenBubble(s, t) and hence these cases do not need to be considered by the oracles. Additionally, assume that the languages in question are non-empty since an initial call to GenBubble(s, t) will always produce the string 1^d0^{n-d}.

5.1 k-ary trees (k-ary Dyck words)

A k-ary Dyck word with density d is a binary string with d 1s and $d(k-1)$ 0s such that every prefix has $\leq k-1$ 0s for every 1. When $k=2$ this means that no prefix has more 0s than 1s. k-ary Dyck words are known to be equivalent to k-ay trees with k internal nodes. In the case when $k=2$, Dyck words are equivalent to many structures counted by the Catalan numbers including *balanced parentheses strings* [12]. Dyck words are a first-01 bubble language.

Our goal is to determine the *smallest* non-negative value j such that $1^{s-1}0^{t-j}10^{j}\gamma$ is a Dyck word given that $1^{s}0^{t}\gamma$ is a Dyck word. In other words we want to find the smallest non-negative j such that: $t-j \leq (s-1)(k-1)$. Thus, the oracle for Dyck words returns $\mathsf{max}(0, t-(s-1)(k-1))$.

COROLLARY 2 *k-ary Dyck words can be generated in either co-lex or cool-lex Gray code order in $O(1)$ amortized time.*

5.2 Ordered forests with $\leq k$ trees

A prefix is *balanced* if it contains the same number of 1s as 0s. Balanced parentheses strings with $\leq k$ balanced (non-empty) prefixes represent ordered forests with $\leq k$ trees and form a first-01 bubble language. To construct an efficient oracle for ordered forests, we maintain an extra parameter bal that indicates the number of non-empty balanced prefixes in α. This parameter can be updated by replacing the recursive call in $\mathsf{GenBubble}(s,t)$ with:

```
**if** $s-1=t-i$ **then** GenBubble$(s-1, t-i, bal+1)$
**else** GenBubble$(s-1, t-i, bal)$
```

Initially, $\alpha = 1^{d}0^{d}$ so the initial call is $\mathsf{GenBubble}(d,d,1)$.

Our goal is to determine the *smallest* non-negative value j such that $1^{s-1}0^{t-j}10^{j}\gamma$ is a balanced parentheses string with at most k balanced prefixes given that $0^{s}1^{t}\gamma$ is a balanced parentheses string with bal balanced prefixes. The oracle for k-ary Dyck words when $k=2$ gives an initial lower bound of $\mathsf{max}(0, t-s+1)$. To make sure the number of allowable balanced prefixes is not violated, consider two special cases when $bal=k$. If $s=t$ and $s>1$ then a new balanced prefix is introduced when $j=1$. Thus, the oracle returns 2. Similarly, if $s=t+1$ a new balanced prefix is introduced when $j=0$. Thus, in this case the oracle returns 1. These cases can be summarized as follows:

```
**procedure** Oracle$(s, t, bal$: **int**)
    **if** $bal=k$ **and** $s=t$ **and** $s>1$ **then return** 2
    **if** $bal=k$ **and** $s=t+1$ **then return** 1
    **return** max$(0, t-s+1)$
**end.**
```

COROLLARY 3 *Ordered forests with $\leq k$ trees can be generated in either co-lex or cool-lex Gray code order in $O(1)$ amortized time.*

5.3 Forbidden substring 01^k or 10^k

Fixed-density strings with no substring of the form 01^k (a first-01 bubble language) has a straightforward oracle that takes advantage of the run-length block data structure. We need only be careful when $s_{c-1} = k - 1$ since if an additional 1 is appended to this block then the resulting string will contain the forbidden substring. Thus, if $s_{c-1} = k - 1$ the oracle returns 1; otherwise it returns 0.

Fixed-density strings with no substring of the form 01^k form a first-01 bubble language and we can apply the same oracle.

COROLLARY 4 *Fixed-density strings with forbidden substring 01^k or 10^k can be generated in either co-lex or cool-lex Gray code order in $O(1)$ amortized time.*

These strings have been studied for specific values of k as outlined in [11].

5.4 Strings with $\leq k$ inversions

An *inversion* (with respect to 1*0*) in a string $\alpha = a_1 \cdots a_n$ is any $a_i = 0$ and $a_j = 1$ such that $i < j$. For example the string $a_1 \cdots a_6 = 100101$ has 5 inversions: (a_2, a_4), (a_2, a_6), (a_3, a_4), (a_3, a_6), (a_5, a_6). Observe that the number of inversions in a string is the minimum number of *adjacent* transpositions required to obtain a string of the form 1^*0^*. Fixed-density strings with at most k inversions form a first-01 bubble language.

To construct an oracle for this language, we maintain an additional parameter inv in GenBubble(s, t) that stores the number of inversions in the current string. By prepending the prefix 10^i to γ during a recursive call, $t - i$ new inversions are introduced into the string. Thus, the parameter inv can be maintained by replacing the recursive call with GenBubble$(s, t, inv{+}t{-}i)$ and initially calling GenBubble$(s, t, 0)$.

Our goal is to determine the *smallest* non-negative value j such that $1^{s-1}0^{t-j}10^j\gamma$ has at most k inversions given that $1^s0^t\gamma$ has inv inversions. In other words, we want to find the smallest j such that $inv + (t - j) \leq k$. Thus, the oracle returns Max(0, $t - k + inv$).

An *inversion* (with respect to 0^*1^*) in a string $\alpha = a_1 \cdots a_n$ is any $a_i = 1$ and $a_j = 0$ such that $i < j$. Fixed-density strings with at most k of these inversions form a first-10 bubble language and we can apply the same oracle.

COROLLARY 5 *Fixed-density strings with at most k inversions can be generated in either co-lex or cool-lex Gray code order in $O(1)$ amortized time.*

5.5 Strings with $\leq k$ transpositions to sort

As previously mentioned, another way to look at a string with k inversions is that it requires k *adjacent* transpositions to sort the string into the form 1^*0^*. If the "adjacent" criteria is removed, then we can consider a bound k on the number of transpositions required to sort a string. For example, while the string 100101 requires 5 adjacent transpositions (it has 5 inversions), it requires only 2 transpositions to sort it: namely swapping the 0s in positions 2 and 3 with the 1s in position 4 and 6. Fixed-density strings requiring at most k transpositions to *sort* the string form a first-01 bubble language.

To construct an efficient oracle for this language, we maintain an additional parameter $sort$ that keeps track of how many 1s occur past position d in the string. This is done by replacing the recursive call with:

```
**if** $s+t-i > d$ **then** GenBubble($s-1, t-i, sort+1$)
**else** GenBubble($s-1, t-i, sort$)
```

The initial call is GenBubble($d, n-d, 0$).

Our goal is to determine the *smallest* non-negative value j such that $1^{s-1}0^{t-j}10^j\gamma$ has at most k 1s past position d given that $0^s1^t\gamma$ has exactly $sort$ 1s past position d. If $sort = k$ then $(s-1)+(t-j)+1 \leq d$ which means the oracle returns Max(0, $s+t-d$). Otherwise, the oracle returns 0.

Fixed-density strings that require at most k transpositions to sort a string into the form 0^*1^* are a first-10 bubble language. To apply the aforementioned oracle, replace d with $n-d$ in the oracle.

COROLLARY 6 *Fixed-density strings that can be sorted with at most k transpositions can be generated in either co-lex or cool-lex Gray code order in $O(1)$ amortized time.*

5.6 Linear extensions of B-posets

If the maximum position of the i-th one in a fixed-density string of length n is bounded by pos_i, then the string is a linear extension of a B-poset [7]. An oracle for this object simply returns Max(0, $s+t-pos_s$) since $(s-1)+(t-j)+1 \leq pos_s$.

COROLLARY 7 *Linear extensions of B-posets can be generated in either co-lex or cool-lex Gray code order in $O(1)$ amortized time.*

Another Gray code for linear extensions is given in [6].

5.7 Strings that are $\geq$ or $\leq$ a given string ω

Strings that are greater than or equal to a given $\omega = w_1 w_2 \cdots w_n$ form a first-01 bubble language. To obtain an efficient oracle for this language, we maintain an extra parameter $flag$ that is set to TRUE if and only if the current suffix γ of α is greater than or equal to the suffix of the same length in ω. To efficiently maintain this extra parameter, we need to know whether or not two bits w_i and w_j belong to the same block. To check this in constant time we pre-compute the block number associated with each bit and store it in $b_1 \cdots b_n$. For example the string $w_1 \cdots w_9 = 111010011$ with 3 blocks has the corresponding block number sequence: $b_1 \cdots b_9 = 333322211$.

To maintain this *flag* as 10^i is prepended to γ, the recursive call is replaced with:

```
**if** $w_{s+t-i} = 0$ **then** GenBubble($s-1, t-i$, TRUE)
**else if** $i = 0$ **or** ($w_{s+t-i+1} = 0$ **and** $b_{s+t-i+1} = b_{s+t}$) **then** GenBubble($s-1, t-i, flag$)
**else** GenBubble($s-1, t-i$, FALSE)
```

The initial call with the new parameter is GenBubble($d, n-d$, TRUE).

Using this extra parameter, we still need to compare the prefix of α with the prefix of ω efficiently. Thus we also pre-compute the first 2 blocks of ω: (u, v) and (y, z). From our example string $w_1 \cdots w_9 = 111010011$, these blocks are $(u, v) = (3, 1)$ and $(y, z) = (1, 2)$ respectively.

Using these blocks, our goal is to determine the *smallest* non-negative value j such that $1^{s-1}0^{t-j}10^j\gamma \geq \omega$ where ω starts with $1^u0^v1^y0^z$. Thus if $s-1 < u$ then the oracle returns t. Also if $s-1 > u$ or $s-1 = u$ and $t < v$ then clearly the oracle returns 0. Otherwise $s-1 = u$ and $t \geq v$. This will lead to two possible values for the bubble lower bound: $t-v$ or $t-v+1$. Clearly the latter case will result in a string larger than ω since α's prefix would become $(u, v-1)$. Thus, we determine when setting $j = t-v$ will result in a string that is greater than or equal to ω. This will be the case if $flag$ = TRUE and either (i) $t = v$ or (ii) $y = 1$ and $t - v \leq z$. These observations can be simplified and summarized as follows:

```
**procedure** Oracle($s, t, flag$: **int**)
    **if** $s-1 < u$ **then return** $t$
    **if** $s-1 > u$ **or** $t < v$ **then return** 0
    **if** ($t = v$ **or** ($y = 1$ **and** $t-v \leq z$)) **and** $flag$ **then return** $t-v$
    **return** $t-v+1$
**end.**
```

Strings that are less than or equal to ω form a first-10 bubble language. For this language we can apply the same oracle but when considering the bits within the string ω, we replace the two bit comparisons with 0 to 1. Also, the pre-computed block information in ω must be of the form 0*1* instead of 1*0*.

COROLLARY 8 *Fixed-density strings that are greater (less) than or equal to a string ω can be generated in either co-lex or cool-lex Gray code order in $O(1)$ amortized time.*

5.8 Strings that are $\geq$ or $>$ or $\leq$ or $<$ their reversal

Fixed-density strings that are greater than (or equal) to their reversal form a first-01 bubble language. First we provide an oracle that includes palindromes (strings equal to their reversal) and then add details that will allow the oracle to reject the palindromes. For our discussion let the reversal of a string α be denoted α^R. The oracle described here requires the run-length block encoding data structure outlined in Section 3.1.

At the start of each recursive call $\alpha = 1^s0^t\gamma$ and $\alpha \geq \alpha^R$. We are trying to determine the *smallest* non-negative value j such that $1^{s-1}0^{t-j}10^j\gamma$ is also greater than or equal to its reversal. If the length of γ is greater than $s+t$ then let δ denote the string γ with the last $s+t$ bits removed. When testing a given j, if the reversal of γ also starts $1^{s-1}0^{t-j}10^j$ then we need to determine whether or not $\delta \geq \delta^R$. To obtain this information in constant time we maintain an additional parameter $flag$ in the recursion that is TRUE if and only if $\delta \geq \delta^R$. As an example, let $\alpha = 110000011001110000100$ where $s = 2, t = 5$ and $\gamma = 11001110000100$. Then $\delta = 1100111$ and $flag =$ FALSE since $\delta < \delta^R$. To efficiently update this new parameter we may need to efficiently determine which block a given 1 (not in the first block) belongs to. Thus, the global array b is maintained that stores which block a specific 1 (not in the first block) belongs to. From our example string α, the 1 in position 8 belongs to the 2nd block and thus $b_8 = 2$. Similarly $b_9 = 2$ and $b_{12} = 3$.

During a recursive call, as 10^i is prepended to γ, we need to update $flag$. Let $p = s+t-i$ which denotes the position of the 1 (from the 10^i) in the updated string α. Then if $q = n-p+1$, we compare the strings $a_pa_{p+1}\cdots a_{p+i}$ with $a_qa_{q-1}\cdots a_{q-i}$. To perform this test efficiently, first test $a_p = 1$ against a_q. If $a_q = 0$ then we set $flag$ to TRUE; otherwise we check further. If $i = 0$ then $flag$ remains unchanged, but if $i > 0$ then test $a_{p+1} = 0$ with a_{q-1}. If $a_{q-1} = 1$ then set $flag$ to FALSE. Otherwise since $a_{p+1}\cdots a_{p+i} = 0^i$ test if the same is true for $a_{q-i}a_{q-i+1}\cdots a_{q-1}$. Because $a_q = 1$ and $a_{q-1} = 0$, the 1 in position a_q is at the start of the block b_q. The number of consecutive 0s before this 1 is given by t_{b_q+1}. Thus if $t_{b_q+1} \geq i$ the value for $flag$ remains unchanged. Otherwise set $flag$ to FALSE. There is one special case where $p > n/2$ in which case δ is empty so $flag$ remains TRUE.

From our discussion, to update the value for *flag* the recursive call is replaced with the following:

```
$p := s + t - i$
$q := n - p + 1$
$b_p := c - 1$
```

```
if $p > n/2$ or $a_q = 0$ then GenBubble($s-1, t-i$, TRUE)
else if $i = 0$ or ($a_{q-1} = 0$ and $t_{b_q+1} \geq i$) then GenBubble($s-1, t-i, flag$)
else GenBubble($s-1, t-i$, FALSE)
```

The initial call with the added parameter is GenBubble($d, n-d$, TRUE).

Once again, recall that we are trying to determine the *smallest* non-negative value j such that $\beta = 1^{s-1}0^{t-j}10^j\gamma$ is greater than or equal to its reversal. Using $flag$ we obtain a $O(1)$ time oracle by comparing $1^{s-1}0^{t-j}10^j$ with the start of the reversal $0^{t_1}1^{s_1}0^{t_2}1^{s_2}0^{t_3}$. We examine two cases depending on t_1 being careful when $c = 1$ or $c = 2$.

Case 1: $t_1 = 0$. In this case we compare $1^{s-1}0^{t-j}10^j$ with $1^{s_1}0^{t_2}1^{s_2}0^{t_3}$. If $s-1 > s_1$ then the oracle returns 0. If $s-1 < s_1$ then the oracle returns t. The interesting case is when $s-1 = s_1$. As a special case when $c = 2$ the oracle simply returns $\lfloor (t+1)/2 \rfloor$. Otherwise observe that $\beta > \beta^R$ if $j = t - t_2 + 1 \geq 0$ and $\beta < \beta^R$ if $j = t - t_2 - 1 \geq 0$. Thus, consider what happens if $j = t - t_2$. If $j < 0$ then the oracle simply returns 0. Also notice that if $j = 0$ and $flag =$ TRUE then $\beta = \beta^R$. Otherwise, in order for $\beta \geq \beta^R$, we have $s_2 = 1$, $t_3 \geq j$ and $flag =$ TRUE. The following subroutine (which we use again later) handles these latter cases with the call TestJ($t - t_2, flag, s_2, t_3$):

```
procedure TestJ($j$, $flag$, $ones$, $zeros$: int)
    if $j < 0$ or ( $j = 0$ and $flag$ ) then return 0
    if $ones = 1$ and $zeros \geq j$ and $flag$ then return $j$
    return $j + 1$
end.
```

Case 2: $t_1 > 0$. If $s > 1$ then β will start with a 1 for any j. Thus the oracle returns 0. Otherwise $s = 1$ and we compare $0^{t-j}10^j$ with $0^{t_1}1^{s_1}0^{t_2}$. In a special case if $c = 1$ then β the oracle returns $\lfloor n/2 \rfloor$. For the remaining cases we encounter a problem similar to the previous case which can be handled with the call TestJ($t - t_1, flag, s_1, t_2$).

The following summarizes the oracle for reversible strings:

```
procedure Oracle($s$, $t$, $flag$: int)
    if $t_1 = 0$ then
        if $s - 1 > s_1$ then return 0
        if $s - 1 < s_1$ then return $t$
        if $c = 2$ return $\lfloor (t+1)/2 \rfloor$
        return TestJ($t - t_2$, $flag$, $s_2$, $t_3$)
    if $s > 1$ then return 0
    if $c = 1$ then return $\lfloor n/2 \rfloor$
    return TestJ($t - t_1$, $flag$, $s_1$, $t_2$)
end.
```

Since fixed-density strings that are less than or equal to their reversal form a first-10 bubble language we can apply the exact same oracle.

COROLLARY 9 *Fixed-density strings that are greater (less) than or equal to their reversal can be generated in either co-lex or cool-lex Gray code order in $O(1)$ amortized time.*

Reversible strings have also been called "neckties" and Gray codes for them are presented in [15].

5.8.1 No palindromes

If we consider fixed-density strings that are strictly greater than their reversals, which are also a first-01 bubble language, then the following modifications are required:

- When updating $flag$, if $\delta = \epsilon$, then $flag$ = FALSE. Thus set $flag$:= FALSE in the recursive call when $p > n/2$.
- In the oracle when $t_1 = 0$ we make two changes. First when $c = 2$ and $s - 1 = s_1 + 1$ we return 1. Second, for the case when $c = 2$ and $s - 1 = s_1$, the oracle returns $t/2 + 1$ instead of $\lfloor (t+1)/2 \rfloor$.
- In the oracle when $t_1 > 0$ we also make 2 changes. First when $c = 1$ and $s = 2$ we return 1. Second for the case when $c = 1$ and $s = 0$ the oracle returns $\lfloor (n+1)/2 \rfloor$ instead of $\lfloor n/2 \rfloor$.

COROLLARY 10 *Fixed-density strings that are strictly greater (less) than to their reversal can be generated in either co-lex or cool-lex Gray code order in $O(1)$ amortized time.*

5.9 Strings that are $\geq$ or $>$ or $\leq$ or $<$ their complemented reversal

Fixed-density strings that are greater than or equal to their *complemented* reversals form a first-01 bubble language if $d \geq \lceil n/2 \rceil$. These strings will be useful later when we consider connected unit interval graphs. The construction of an oracle for this language is very similar to the previous language for reversals and once again uses the run-length block data structure. As defined in the previous subsection, we will use the term δ and will require the global array b that maintains the index of the blocks for each 1 in the string α. We maintain a *flag* which is set to TRUE if and only if δ (as defined in the previous section) is greater than or equal to its complemented reversal.

Updating the $flag$ for complemented reversals is very similar to reversals. In particular, to update the value for *flag* as we add each 10^i to γ, replace the recursive call with the following:

```
    $p := s + t - i$
    $q := n - p + 1$
    $b_p := c - 1$
    **if** $p \geq n/2$ **or** $a_q = 1$ **then** GenBubble($s-1, t-i$, TRUE)
    **else if** $i = 0$ **or** ($a_{q-1} = 1$ **and** $s_{b_{q-1}} \geq i$) **then** GenBubble($s-1, t-i$, *flag*)
    **else** GenBubble($s-1, t-i$, FALSE)
```

The initial call with this added parameter is GenBubble($d, n-d$, TRUE).

Using this extra parameter $flag$, we now describe the oracle for complemented reversals. Recall we are trying to determine the smallest non-negative value for j such that $\beta = 1^{s-1}0^{t-j}10^j\gamma$ is greater than or equal to its complemented reversal. So in particular we need to compare $1^{s-1}0^{t-j}10^j$ with the reversed complemented suffix $1^{t_1}0^{s_1}1^{t_2}0^{s_2}$. For the special case when $c = 1$, clearly a 1 can be moved into the last position a_n so the oracle returns 0. If $s - 1 > t_1$, then the oracle also returns 0; if $s - 1 < t - 1$ then the oracle returns t. The interesting case is when $s - 1 = t_1$. When $c = 2$, there is a special case: if $t - 1 \leq s_1$ then the oracle returns 0. Otherwise we can re-use the function TestJ(j,*flag,ones,zeros*) presented in the previous subsection by calling TestJ($t - s_1$, *flag*, t_2, s_2). The resulting oracle can be summarized as follows:

```
**procedure** Oracle($s$, $t$, *flag*: **int**)
    **if** $c = 1$ **or** $s - 1 > t_1$ **then return** $0$
    **if** $s - 1 < t_1$ **then return** $t$
    **if** $c = 2$ **and** $t - 1 \leq s_1$ **return** $0$
    **return** TestJ($t - s_1$, *flag*, $t_2, s_2$)
**end.**
```

Since fixed-density strings that are less than or equal to their complemented reversal form a first 10-bubble language we can apply the exact same oracle.

COROLLARY 11 *Fixed-density strings that are greater (less) than or equal to their complemented reversal can be generated in either co-lex or cool-lex Gray code order in $O(1)$ amortized time.*

Note that a string cannot equal its complemented reversal unless $n = 2d$. Thus, this result applies to strings that are strictly greater (less) than their complemented reversals when $d > n/2$.

5.10 0-1 Knapsack

Consider a knapsack with capacity C and a set of n items sorted by non-decreasing weights $w_1 w_2 \cdots w_n$. The set of all subsets of d items whose total weight does not exceed the capacity form a first-01 bubble language. For example, consider a knapsack with capacity $C = 22$ and 5 items with weights 2, 4, 6, 6, and 15 respectively. Then the 5 feasible solutions with 3 items, where a 1 indicates that we are including the item, are: 11100, 11010, 11001, 10110, and 01110.

To obtain an efficient oracle we maintain an extra parameter $avail$ that represents the available capacity from using the current d items. As the string 10^i is prepended to γ, we are effectively removing the s-th item from the knapsack and inserting the $s + t - i$-th item. Thus to maintain this parameter, replace the recursive call with GenBubble$(s - 1, t - i, avail - w_{s+t-i} + w_s)$ where the initial call is GenBubble$(d, n - d, C - \sum_{k=1}^{d} w_k)$.

Using this extra parameter, the oracle determines the heaviest item (up to item $s + t$) that can be swapped with the s-th item so we do not violate the capacity C. This can be done as follows:

```
procedure Oracle($s$, $t$, avail: int)
int $j$
    $j := t$
    while $j > 0$ and $avail \geq w_{s+t-j+1} - w_s$ do $j := j - 1$
    return $j$
end.
```

Clearly this oracle does not run in $O(1)$ time. However observe that for every successful iteration of the **while** loop, we will generate a recursive call in GenBubble(s, t, *avail*). This means we will still achieve a CAT algorithm.

COROLLARY 12 *Feasible solutions to the 0-1 knapsack problem with exactly d items can be generated in either co-lex or cool-lex Gray code order in $O(1)$ amortized time.*

A comprehensive discussion on knapsack problems is given in [2].

5.11 Closure properties of oracles

Bubble languages are closed under both union and intersection [7] (with respect to either the first-01 bubble or first-10 bubble property). Thus if there are oracles for two first-01 bubble languages $\mathbf{L}_1$ and $\mathbf{L}_2$, then an oracle for $\mathbf{L}_1 \cup \mathbf{L}_2$ will be the minimum of the two oracles and an oracle for $\mathbf{L}_1 \cap \mathbf{L}_2$ will be the maximum of the two oracles.

As an example, connected unit interval graphs with d vertices can be expressed as a first-01 bubble language of length $n = 2d$ and density d. This language is the intersection of balanced parentheses strings with 1 balanced prefix and strings that are greater than or equal to their complemented reversals [7]. Since there exists $O(1)$ time oracles for these two languages, we obtain a $O(1)$ time oracle for connected unit interval graphs.

COROLLARY 13 *Connected unit interval graphs can be generated in either co-lex or cool-lex Gray code order in $O(1)$ amortized time.*

Another Gray code for connected unit interval graphs is also presented in [9].

6 Layering densities

In the previous section we provided efficient oracles for a number of fixed-density bubble languages that allows the languages to be generated in $O(1)$ amortized time. For many of the objects, there are corresponding bubble languages when the density is *not* fixed. For such languages, we can concatenate the cool-lex Gray codes together over all densities to obtain Gray codes that are ordered by density. Alternatively if we output the *even* densities in increasing order followed by the *odd* densities in decreasing order, then we will obtain a *cyclic* Gray code [7].

THEOREM 3 *The following objects can be listed in a cyclic Gray code order in $O(1)$ amortized time:*

first-01 bubble languages

- *strings with forbidden* 01^k
- *strings with* $\leq k$ *inversions from* 1^*0^*
- *strings with* $\leq k$ *transpositions from* 1^*0^*
- *strings* $\geq$ *some string* ω
- *strings* $\geq$ *their reversal*
- *strings* $>$ *their reversal*
- *feasible solutions to 0-1 knapsack.*

first-10 bubble languages

- *strings with forbidden* 10^k
- *strings with* $\leq k$ *inversions from* 0^*1^*
- *strings with* $\leq k$ *transpositions from* 0^*1^*
- *strings* $\leq$ *to some string* ω
- *strings* $\leq$ *their reversal*
- *strings* $<$ *their reversal*

7 Summary

We have provided a recursive framework to efficiently generate many fixed-density bubble languages in cool-lex Gray code order. Moreover, we have reduced the problem of generating the Gray code for a particular bubble language to the problem of providing a membership tester for it.

Question: Are there other interesting bubble languages and do they have efficient oracles?

Acknowledgements

Thanks to Frank Ruskey for many helpful discussions and valuable comments.

References

[1] J.P. Duval, Factorizing words over an ordered alphabet, J. Algorithms, Vol. 4 No. 4 (1983) 363-381.

[2] H. Kellerer, U. Pferschy and D. Pisinger, *Knapsack Problems*, Springer, 2004.

[3] D. E. Knuth, The Art of Computer Programming, Volume 4: Generating All Tuples and Permutations, Fascicle 2, Addison-Wesley, 2005.

[4] D. E. Knuth, The Art of Computer Programming, Volume 4: Generating all Combinations and Partitions, Fascicle 3, Addison-Wesley, 2005.

[5] D. E. Knuth, The Art of Computer Programming, Volume 4: Generating All Trees; History of Combinatorial Generation, Fascicle 4, Addison-Wesley, 2006.

[6] G. Pruesse and F. Ruskey, Generating linear extensions fast, SIAM Journal on Computing, Vol.23 No. 2 (1994) 373-386.

[7] F. Ruskey, J. Sawada and A. Williams, Binary bubble languages and cool-lex Gray codes, Journal of Combinatorial Theory, Series A, 119(1):155-169, 2012.

[8] F. Ruskey, J. Sawada and A. Williams, De Bruijn sequences for fixed-weight binary strings, SIAM Journal on Discrete Mathematics, 2012, to appear.

[9] T. Saitoh, K. Yamanaka, M. Kiyomi and R. Uehara, Random generation and enumeration of proper interval graphs, WALCOM '09: Third International Workshop on Algorithms and Computation, LNCS 5431 (2009) 177-189.

[10] J. Sawada and A. Williams, A Gray code for fixed-density necklaces and Lyndon words in constant amortized time, Theoretical Computer Science, 2012, to appear.

[11] N. Sloane, The on-line encyclopedia of integer sequences, `http://www.research.att.com/njas/sequences`, sequence numbers A000071, A004070, A008937, A055216, A107066, A107065.

[12] R. Stanley, *Enumerative Combinatorics*, Cambridge University Press, 1997.

[13] T. Ueda, Gray codes for necklaces, Discrete Math., Vol. 219 No. 1-3 (2000) 235-248.

[14] V. Vajnovszki and T. Walsh, A loop-free two-close Gray-code algorithm for listing k-ary Dyck words, Journal of Discrete Algorithms, Vol. 4 No. 4 (2006) 633–648.

[15] T. Wang and F. Ruskey, Generating neckties: algorithms Gray codes and parity differences, (1993), preprint.

[16] T. Wang and C. Savage, A Gray code for necklaces of fixed density, SIAM J. Discrete Math., Vol. 9 No. 4 (1996) 654-673.

[17] A. Williams, *Shift Gray codes*, PhD thesis in Computer Science, University of Victoria, 2009.