BINARY BUBBLE LANGUAGES AND COOL-LEX ORDER[*]

FRANK RUSKEY[†], JOE SAWADA[‡], AND AARON WILLIAMS[§]

Abstract. A bubble language is a set of binary strings with a simple closure property: The first 01 of any string can be replaced by 10 to obtain another string in the set. Natural representations of many combinatorial objects are bubble languages. Examples include binary string representations of k-ary trees, unit interval graphs, linear-extensions of B-posets, binary necklaces and Lyndon words, and feasible solutions to knapsack problems. In co-lexicographic order, fixed-density binary strings are ordered so that their suffixes of the form 10^i occur (recursively) in the order $i = \max, \max -1, \ldots, \min +1, \min$ for some values of max and min. In *cool-lex order* the suffixes occur (recursively) in the order $\max -1, \ldots, \min +1, \min, \max$. This small change has significant consequences. We prove that the strings in any bubble language appear in a Gray code order when listed in cool-lex order. This Gray code may be viewed from two different perspectives. On the one hand, successive binary strings differ by one or two transpositions, and on the other hand, they differ by a shift of some substring one position to the right. This article also provides the theoretical foundation for many efficient generation algorithms, as well as the first construction of fixed-density de Bruijn sequences; results that will appear in subsequent papers.

Key words. combinatorial objects, generation, Gray codes, algorithms, necklaces, reversible strings, Lyndon words, balanced parentheses, trees, Dyck words, interval graphs, knapsack problem

1. Introduction.

Binary strings provide natural representations for the instances of many combinatorial objects, as illustrated by Figure 1.1.

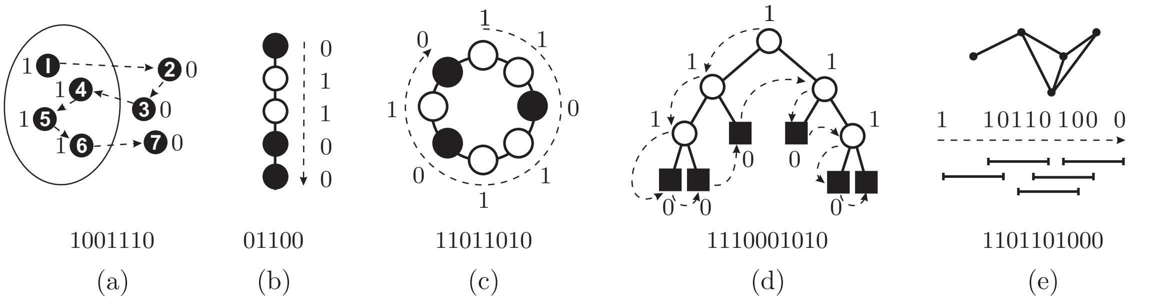


Fig. 1.1: Combinatorial objects and string representations: (a) combinations (bitwise inclusion), (b) reversible strings with two colors (largest reversal), (c) necklaces with two colors (largest clockwise rotation), (d) binary trees (pre-order traversal omitting last 0), and (e) connected unit interval graph (intervals represented by parentheses).

Although the combinatorial objects in Figure 1.1 are diverse, their string representations share a basic property: If the first 01 is replaced by 10, then the resulting string represents another instance of the same combinatorial object. Similarly, in other combinatorial objects the first 10 can be replaced by 01. This motivates our definition of bubble languages, which generalize all of the objects found in Figure 1.2.

All strings in this article are binary, with $\mathbf{B}(n)$ denoting the set with length n and $\mathbf{B}_d(n)$ denoting the set with length n and density d. Subsets of $\mathbf{B}_d(n)$ are said to have *fixed-density*.

Cool-lex is a variation of co-lexicographic order for $\mathbf{B}_d(n)$. In co-lex, strings are recursively ordered by suffixes of the form 10^i for $i = n-d, n-d-1, \ldots, 0$. In cool-lex, strings are instead ordered using $i =$

[*] Research supported in part by NSERC discovery grants.

[†] Department of Computer Science, University of Victoria, PO Box 3010 STN CSC, Victoria BC, V8W 3N4, Canada ruskey@cs.uvic.ca

[‡] School of Computer Science, University of Guelph, 217 Reynolds, Guelph ON, N1G 2W1, Canada jsawada@uoguelph.ca

[§] School of Mathematics and Statistics, Carleton University 1125 Colonel By Drive, Ottawa ON, K1S 5B6, Canada haron@uvic.ca

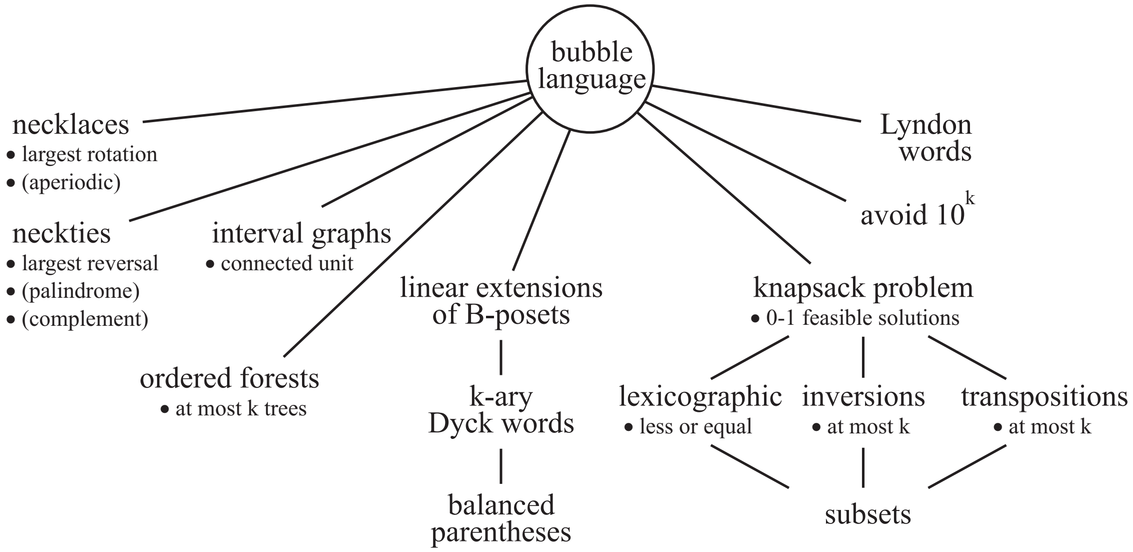


Fig. 1.2: Examples of bubble languages. The figure omits variations obtained by bit-complements such as strings avoiding 01^k, strings lexicographically greater than or equal to a given a string, and necklaces using their smallest rotation.

$n{-}d{-}1, n{-}d{-}2, \ldots, 0, n{-}d$. This small change creates a *Gray code*, meaning that successive strings differ by a constant amount, regardless of the values of n and d (see Ruskey and Williams [18] [19]). Furthermore, a simple rule creates this Gray code one string at a time: If $\alpha \in \mathbf{B}_d(n)$ has a prefix of the form 1^s0^t1x where x is a single bit and $t > 0$, then the next string in cool-lex order replaces this prefix by $x1^s0^t1$. The Gray code ends with the two strings that do not have such a prefix, namely $1^{n-d-1}0^d1$ and $1^{n-d}0^d$. Figure 1.3 compares co-lex and cool-lex order for $\mathbf{B}_5(9)$.

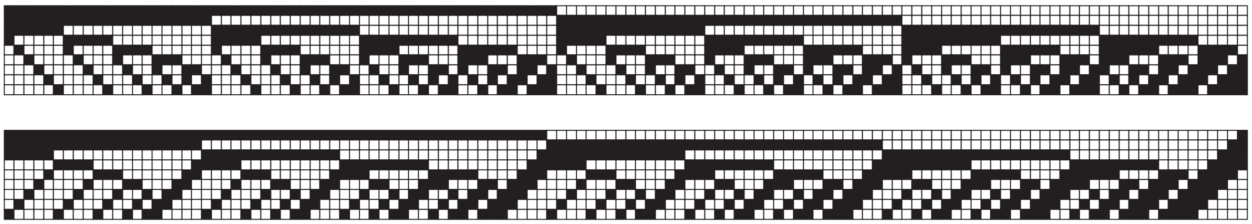


Fig. 1.3: $\mathbf{B}_5(9)$ in co-lex order (top) and cool-lex order (bottom). Columns encode strings and are read from top-to-bottom where 0 and 1 are black and white squares respectively, and successive strings are read from right-to-left.

1.1. Results. This article proves that cool-lex order provides a Gray code for every fixed-density bubble language. More specifically, if $\mathbf{L} \subset \mathbf{B}_d(n)$ is a bubble language, then a Gray code is obtained by ordering the strings in $\mathbf{L}$ according to their relative order in cool-lex. In these Gray codes, successive strings differ by a prefix-replacement that can be described as the transposition of one or two pairs of bits, or by shifting a single bit to the left. These cool-lex Gray codes are also *cyclic* since one transposition (or shift) transforms the last string into the first. For this reason they can be *layered* to create Gray codes for strings with a range of densities (or lengths).

This new framework is used in subsequent articles to create constant amortized time algorithms for generating all of the specific bubble languages discussed in this paper [22], including the first constant amortized time Gray code algorithm for fixed-density necklaces and Lyndon words [23]. An algorithm generates a language $\mathbf{L}$ in *constant amortized time* if it visits each successive string in $\mathbf{L}$ in O(1) amortized time. The cool-lex Gray code is also the basis for the first explicit construction of a fixed-density de Bruijn sequence [17]. The construction can be generated efficiently, with successive blocks of n bits being created in amortized O(1)-time while using only O($n \log n$)-space [23].

1.2. Combinatorial Generation. *Combinatorial generation* is devoted to Gray codes, universal cycles, and efficient generation of various combinatorial objects. Due to its long history and the variety of combinatorial objects, it is a difficult subject to summarize. For example, Section 7.2 of *The Art of Computer Programming* offers excellent coverage on "Generating all possibilities" but requires over 400 pages (see Knuth [6, 7, 8]). This subsection outlines some general methods in the research area.

One way to add cohesion to the research area is to consider more general combinatorial objects. For example, Gray codes and efficient algorithms for permutations [24] were extended to permutations of a multiset [10] [26] [28] [32], which in turn were extended to linear-extensions of partially-ordered sets [16] [2] [9], and finally to the basic words of an anti-matroid [14]. The approach in this article differs from this standard evolution since it introduces a new class of combinatorial objects. The simplicity of bubble languages allow us to prove results for individual objects that were previously the topic of more difficult articles (for example, see Wang and Savage [31], and Ueda [27] for the previous results on fixed-density necklaces). This general approach to combinatorial generation was also followed in the thesis by Williams [33].

Combinatorial generation has also been enriched by using the same technique to create Gray codes and efficient algorithms for multiple combinatorial objects. For example, *reverse search* has led to multiple Gray codes (see Saitoh et al [20] for proper interval graphs). The *twisted lexico* computation tree has also created Gray codes and efficient algorithms for multiple combinatorial objects (see Takaoka for multiset permutations [26]) and generalizes the recursive structure of the reflected Gray code (see Gray [5] and Knuth [6]). The reflected Gray code inspired Gray codes for *reflectable language* (see Li and Sawada [12]) and their efficient generation algorithms (see Xiang et al [34]). A different approach was taken by Vajnovszki [29], who showed that a Gray code for Lyndon words is obtained by using the relative order of strings in the reflected Gray code. This sublist approach is used in this article.

The ECO framework enumerative combinatorics has been used to obtain Constant amortized algorithms for a variety of objects (see Bacchelli et al [1]). More restrictive than a constant amortized time algorithm is a *loopless algorithm* that generates successive strings in worst-case O(1)-time. Loopless algorithms for multiple combinatorial objects were obtained by Walsh [30], who extended those initially given by Ehrlich [4].

A de Bruijn sequence is a circular string of length 2^n containing each binary strings of length n exactly once as a substring. Universal cycles were introduced in Graham et al [3] as natural generalizations of de Bruijn sequences, and they proved the existence of universal cycles for a number of combinatorial objects. Since that time, there has been a tradition of showing that universal cycles exist for classes of combinatorial objects by proving that their associated de Bruijn graphs are Eulerian. For example, see the results of Moreno et al [13] and LaBounty-Lay et al [11]. However, one aspect that is commonly missing from these articles is a discussion of how to efficiently create an individual universal cycles without constructing the underlying graph, which often has exponential size.

1.3. Article Outline. This article is organized as follows. Section 2 introduces bubble languages, and describes their basic properties. Section 3 discusses cool-lex, and proves that the order gives a Gray code for any bubble language. Section 4 contains recursive algorithms that generate the strings in an arbitrary bubble language. Section 5 shows that each of the objects mentioned in Figure 1.2 is a bubble language.

2. Bubble Languages.

This section defines bubble languages in Section 2.1 and the bubble poset in Section 2.2. Basic properties of bubble languages are given in Section 2.3. Finally, Section 2.4 gives a recursive formula for generating the strings in an arbitrary fixed-density bubble language.

2.1. Definitions.

A set of binary strings $\mathbf{L}$ is a binary bubble language if it satisfies one of the following two properties:

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

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

More specifically, a language $\mathbf{L}$ is a *first-01 bubble language* if it satisfies the first-01 property, and is a *first-10 bubble language* if it satisfies the first-10 property. In some situations it is helpful to differentiate between these two concepts. This is especially true in Section 5, where we prove that the following combinatorial objects can be represented by bubble languages

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 other situations there is no need to differentiate between the two varieties, and so we focus on first-01 bubble languages. In particular, the unqualified term *bubble language* means first-01 bubble language for the remainder of this document.

To illustrate a specific bubble language, consider the language $\mathbf{L} \subseteq \mathbf{B}(5)$ containing strings ≥ 10110

$$\mathbf{L} = \{10110, 10111, 11000, 11001, 11010, 11011, 11100, 11101, 11110, 11111\}. \tag{2.1}$$

Notice $\mathbf{L}$ is a bubble language since it satisfies the stronger condition that replacing any 01 by 10 will result in another string ≥ 10110. The above example also illustrates an important point stated in Remark 2.1. Since replacing any 01 by 10 does not change the length or the density of a string, then bubble languages can be partitioned into fixed-density subsets. For example, $\mathbf{L}$ in (2.1) partitions into bubble languages over its various $\mathbf{B}_d(n)$ below

$$\{11000\} \cup \{10110, 11001, 11010, 11100\} \cup \{10111, 11011, 11101, 11110\} \cup \{11111\}.$$

Remark 2.1. *A language is a bubble language if and only if its subsets over each $\mathbf{B}_d(n)$ are bubble languages.*

2.2. Bubble Poset. Bubble languages over $\mathbf{B}_d(n)$ are the ideals of a partially ordered set that we call the *bubble poset* $\mathcal{P}(n, d)$. To describe the poset we make the following definitions. As is common, x^y represents y consecutive copies of symbol x. If $\alpha = 1^d0^{n-d}$, then we say α is *terminal* and $\tau(\alpha) = \alpha$. Otherwise, $\tau(\alpha)$ is obtained by replacing the first 01 in α with 10. Elements of $\mathcal{P}(n, d)$ are $\mathbf{B}_d(n)$, and the cover relations are defined by $\tau(\alpha) \prec \alpha$ where α is not terminal. Figure 2.1 shows the Hasse diagram of $\mathcal{P}(6, 3)$ with the ideal for reversible strings in bold.

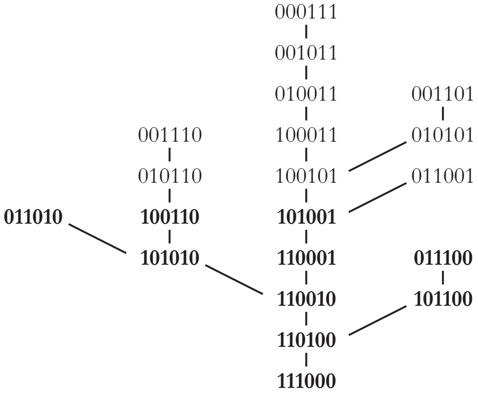


Fig. 2.1: The bubble poset $\mathcal{P}(3, 3)$, with its reversible string ideal in bold.

Figure 2.1 illustrates that $\mathcal{P}(n, d)$ is always a tree with 1^d0^{n-d} as the unique minimum element, and that its ideals are the subtrees that contain this minimum. This setting makes it easier to count the number of fixed-density first-01 bubble languages, with [33] showing that there are more than 10^{36} of them over $\mathbf{B}_5(10)$ alone.

2.3. Properties. This section proves three basic lemmas for bubble languages. The first lemma provides various closure properties. If $\mathbf{L}$ is a set of strings and γ is a string, then the *quotient* of $\mathbf{L}$ and γ is

$$\mathbf{L}/\gamma = \{\alpha \mid \alpha \cdot \gamma \in \mathbf{L}\}.$$

For example, $\mathbf{L}/10 = \{101, 110, 111\}$ given $\mathbf{L}$ from (2.1). Notice that we use $\cdot$ for string concatenation.

Lemma 2.1 (Closure). *If $\mathbf{L}$ and $\mathbf{L}'$ are bubble languages and γ is a string, then $\mathbf{L} \cup \mathbf{L}'$, $\mathbf{L} \cap \mathbf{L}'$, and $\mathbf{L}/\gamma$ are bubble languages.*

Proof. The intersection and unions of ideals of any poset are also ideals of that poset, so the first two closure properties are true.

Let $\beta \in \mathbf{L}/\gamma$. Then $\beta\gamma \in \mathbf{L}$, and thus $\tau(\beta\gamma) \in \mathbf{L}$. There are two cases to consider. First suppose that β is not terminal. It follows that $\tau(\beta\gamma) = \tau(\beta)\gamma$, and thus $\tau(\beta)\gamma \in \mathbf{L}$. Therefore, in this case, $\tau(\beta) \in \mathbf{L}/\gamma$. Otherwise if β is terminal, then $\tau(\beta) = \beta$, and so $\tau(\beta) \in \mathbf{L}/\gamma$. In both cases it was shown that $\tau(\beta) \in \mathbf{L}/\gamma$. Therefore, bubble languages are closed under quotients. ▯

The next two lemmas prove basic prefix and suffix properties for bubble languages.

Lemma 2.2 (Prefix Property). *If $\mathbf{L}$ is a bubble language, then $\mathbf{L}/\gamma \neq \emptyset$ implies $1^s0^t\gamma \in \mathbf{L}$ for some*

$s, t \geq 0$. *In other words, if* $\mathbf{L}$ *is a bubble language that contains a string with suffix* γ*, then* $\mathbf{L}$ *contains a string of the form* $1^*0^*\gamma$.

Proof. Given a string in $\mathbf{L}$ with suffix γ, the string $1^*0^*\gamma \in \mathbf{L}$ is eventually obtained by repeatedly replacing the first 01 by 10. □

Lemma 2.3 (Suffix Property). *If* $\mathbf{L}$ *is a bubble language, then* $\mathbf{L}/01\gamma \neq \emptyset$ *implies* $\mathbf{L}/10\gamma \neq \emptyset$*. In other words, if* $\mathbf{L}$ *contains a string with suffix* 01γ*, then* $\mathbf{L}$ *contains a string with suffix* 10γ*.*

Proof. If $\mathbf{L}$ contains a string with suffix 01γ, then Lemma 2.2 implies that $\mathbf{L}$ contains a string of the form $1^s0^t01\gamma$. Then, $\tau(1^s0^t01\gamma) = 1^s0^t10\gamma$ proves that $\mathbf{L}$ contains a string with suffix 10γ. □

2.4. Generation in Co-lexicographic Order. This section gives a recursive formula for generating the strings in an arbitrary bubble language over $\mathbf{B}_d(n)$ in co-lexicographic order. *Co-lexicographic (co-lex) order* sorts strings by increasing value of their last symbol. In other words, co-lex order is lexicographic order except strings are read from right-to-left. We develop our recursive formula for bubble languages in three steps, and then describe it in terms of a computation tree. Both of these descriptions are helpful in Section 3, when we modify co-lex order to produce a Gray code.

In our recursive formulae we use three parameters: s, t, and γ. The parameter γ represents the *fixed-suffix* that is built from right-to-left, whereas s and t represent the remaining number of 1s and 0s that have not yet joined the fixed-suffix. The complete lists are obtained by taking $s = d$, $t = n - d$, and $\gamma = \epsilon$. Our first recursive formula is for generating all of the strings in $\mathbf{B}_d(n)$ in co-lex order. Typically this would be done by extending the fixed-suffix γ by a single bit, starting with 0γ and following with 1γ. Alternatively, one can extend the fixed-suffix by strings of the form 10^i for decreasing i as follows

$$\mathcal{L}(s,t,\gamma) = \begin{cases} 1^s0^t\gamma,\ \mathcal{L}(s{-}1,1,10^{t-1}\gamma),\ldots,\mathcal{L}(s{-}1,t,1\gamma) & \text{if } s > 0 \\ 0^t\gamma & \text{if } s = 0. \end{cases}$$

Notice that the string $1^s0^t\gamma$ is the special case where all of the remaining copies of 0 join the fixed-suffix. That is, $\mathcal{L}(s{-}1,0,10^t\gamma) = 1^s0^t\gamma$. Also notice that the base case occurs when every copy of 1 has been exhausted. Now let us generalize this recursive formula so that it generates an arbitrary language $\mathbf{L} \subseteq \mathbf{B}_d(n)$ in co-lex order. Since there is no guarantee that an individual string is in $\mathbf{L}$, both cases from the first recursive formula must be duplicated

$$\mathcal{L}(s,t,\gamma) = \begin{cases} 1^s0^t\gamma,\ \mathcal{L}(s{-}1,1,10^{t-1}\gamma),\ldots,\mathcal{L}(s{-}1,t,1\gamma) & \text{if } s > 0 \text{ and } 1^s0^t\gamma \in \mathbf{L} \\ \mathcal{L}(s{-}1,1,10^{t-1}\gamma),\ldots,\mathcal{L}(s{-}1,t,1\gamma) & \text{if } s > 0 \text{ and } 1^s0^t\gamma \notin \mathbf{L} \\ 0^t\gamma & \text{if } s = 0 \text{ and } 0^t\gamma \in \mathbf{L} \\ & \text{if } s = 0 \text{ and } 0^t\gamma \notin \mathbf{L}. \end{cases}$$

The final case in this recursive formula is undesirable since it is a computational dead-end that generates no strings. In our final recursive formula we simplify the previous recursive formula by assuming that $\mathbf{L}$ is a bubble language over $\mathbf{B}_d(n)$. Recall that the suffix property in Lemma 2.3 ensures that if $\mathbf{L}/10^i\gamma$ is non-empty for $i > 0$, then $\mathbf{L}/10^{i-1}\gamma$ is also non-empty. Furthermore, if $\mathbf{L}/10^i\gamma$ is non-empty, then the prefix property in Lemma 2.2 ensures that $1^{s-1}0^{t-i}10^i\gamma \in \mathbf{L}$. Combining these observations gives the following

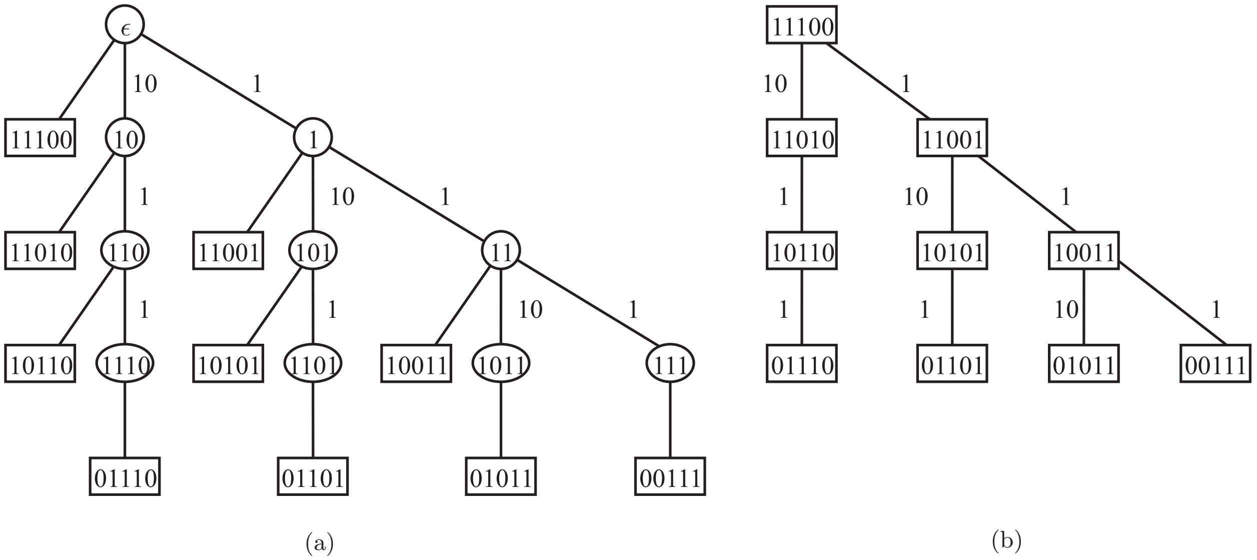


Fig. 2.2: (a) The computation tree, and (b) the compact computation tree for $\mathbf{B}_3(2)$. Co-lex order of $\mathbf{B}_3(2)$ is obtained by reading the leaves from left-to-right in (a), or by a pre-order traversal in (b).

simplified formula for generating the co-lex order of a bubble language $\mathbf{L} \subseteq \mathbf{B}_d(n)$

$$\mathcal{L}(s,t,\gamma) = \begin{cases} 1^s0^t\gamma,\ \mathcal{L}(s{-}1,1,10^{t-1}\gamma),\ldots,\mathcal{L}(s{-}1,t{-}j,10^j\gamma) & \text{if } s>0 \qquad (2.2\text{a}) \\ 0^t\gamma & \text{if } s=0 \qquad (2.2\text{b}) \end{cases}$$

where j is the minimum value such that $\mathbf{L}$ contains a string with suffix $10^j\gamma$. In each of the last two formulae, the fixed-suffix is built from right-to-left by strings of the form 10^i, so we can always assume that γ is either empty or begins with 1.

We close this section by visualizing the computation tree that results from (2.2). In the *computation tree*, each internal node is labeled with the fixed-suffix γ at that point during the computation. In particular, the root has label ϵ. The leftmost child of an internal node is the leaf labeled $1^s0^t\gamma$. The remaining children of an internal node are labeled from left-to-right as $10^i\gamma$ for decreasing values of i, and the edges of the tree are labeled with the corresponding substrings of the form 10^i. The leaves of the tree are the strings in the language $\mathbf{L}$, and the co-lex order of these strings is obtained by reading the leaves from left-to-right. The computation tree for $\mathbf{B}_3(2)$ is illustrated in Figure 2.2 (a), with internal nodes in ovals and leaves in boxes.

Since bubble languages have the prefix property found in Lemma 2.2, we can compress the computation tree by labeling each internal node with the leaf that is its leftmost child. In the *compact computation tree*, each internal node is labeled $1^s0^t\gamma$ where γ is the fixed-suffix at that point during the computation. The children of an internal node are the same as they are in the computation tree, except that the first child is removed. The compact computation tree for $\mathbf{B}_3(2)$ is illustrated in Figure 2.2 (b), with every node in a box.

Remark 2.2. *If* $\mathbf{L}$ *is a bubble language over* $\mathbf{B}_d(n)$*, then its strings can be obtained in co-lex order by a pre-order traversal of the compact computation tree.*

Although the difference between the computation tree and the compact computation tree may seem cosmetic, the distinction proves to be helpful in the next section.

3. Cool-lex Order. This section defines cool-lex order for the strings in an arbitrary bubble language over $\mathbf{B}_d(n)$, and then proves that this order provides a Gray code. The Gray code can be expressed using transpositions or shifts.

Cool-lex order is generated by making a small change to the recursive formula (2.2) we developed for generating co-lex order: the string of the form $1^s0^t\gamma$ appears last instead of first. This change is shown by the following recursive formula

$$\mathcal{C}(s,t,\gamma) = \begin{cases} \mathcal{C}(s{-}1,1,10^{t-1}\gamma),\ldots,\mathcal{C}(s{-}1,t{-}j,10^j\gamma),1^s0^t\gamma & \text{if } s>0 \quad (3.1a) \\ 0^t\gamma & \text{if } s=0 \quad (3.1b) \end{cases}$$

where j is the minimum value such that $\mathbf{L}$ contains a string with suffix $10^j\gamma$, and γ is either empty or begins with 1. Cool-lex order can also be described succinctly by its relationship to the compact computation tree discussed in Section 2.4. When compared to co-lex order, the nodes with labels of the form $1^s0^t\gamma$ are visited last instead of first in cool-lex order.

Remark 3.1. *If $\mathbf{L}$ is a bubble language over $\mathbf{B}_d(n)$, then its strings can be obtained in cool-lex order by a post-order traversal of the compact computation tree.*

To prove that cool-lex order provides a Gray code for any bubble language, we must understand the post-order traversal of the compact computation tree. We will break the post-order traversal into three basic movements: right, up, and down. Each of these movements can be accomplished by a single transposition or shift, and then we explain how these movements combine to give the post-order traversal.

Right. First suppose α is a non-root node in the compact computation tree and α is not the last child of its parent. The node to the *right* of α is its next sibling β. Notice that α must have a label of the form $1^s0^t10\gamma$ for some $t>0$, and β must have a label of the form $1^s0^t01\gamma$. The *right movement* is illustrated by Figure 3.1, and can be accomplished by swapping the bits in positions $s+t+1$ and $s+t+2$.

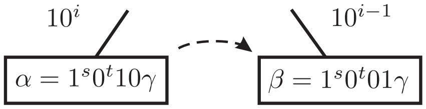


Fig. 3.1: The right movement in the compact computation tree.

Up. Next suppose α is a non-root node in the compact computation tree. The node *above* α is its parent β. Notice that α must have a label of the form $1^s0^t1\gamma$ and β must have a label of the form $1^s10^t\gamma$. The *up movement* is illustrated by Figure 3.2, and can be accomplished by swapping the bits in positions $s+1$ and $s+t+1$.

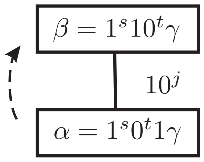


Fig. 3.2: The up movement in the compact computation tree.

Down. Finally, suppose α is a non-leaf node in the compact computation tree. The node *below* α is obtained from α by successively following the first child of each internal node in the compact computation tree until reaching a leaf β. Notice that α must have a label of the form $1^s0\gamma$ and β must have a label of the form $1^i01^{s-i}\gamma$, where i is the minimum value such that $1^i01^{s-i}\gamma$ is in the bubble language $\mathbf{L}$. The *down movement* is illustrated by Figure 3.3, and can be accomplished by swapping the bits in positions $i+1$ and $s+1$. In the figure, the edge label 10^{t-1} assumes that $\alpha = 1^s0^t\gamma'$ where γ' is empty or begins with 1.

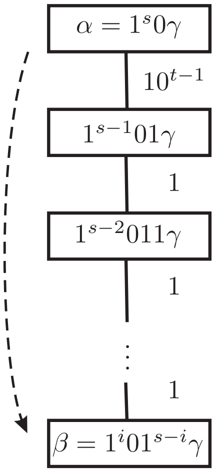


Fig. 3.3: The down movement in the compact computation tree, where i minimizes $1^i01^{s-i}\gamma \in \mathbf{L}$.

Using these three movements, we will now describe how to traverse the compact computation tree for an arbitrary bubble language $\mathbf{L} \subseteq \mathbf{B}_d(n)$ in post-order. More precisely, if $\alpha = 1^s0^t\gamma \in \mathbf{L}$ is the label of a node in the compact computation tree, then we determine the label of the node that follows α cyclically in post-order. Let β be the result of swapping the $(s+t+1)$st and $(s+t+2)$nd symbols in α, and we assume γ is empty or begins with 1. There are three cases to consider:

- If $\alpha = 1^s0^t$ is the root, then it is the last node visited in post-order. The first string in the post-order traversal is obtained from α by the down movement.
- If α is the rightmost child of its parent, then the next string in post-order is obtained by the up movement. Notice that α is the rightmost child of its parent if one of the following three conditions hold: $\gamma = 1$ or $a_{s+t+2} = 1$ or $\beta \notin \mathbf{L}$.
- If α is not the rightmost child of its parent, then the next string in post-order is obtained by the right movement followed by the down movement. Notice that α is not the rightmost child of its parent if the above cases do not apply.

Given these cases and Figures 3.1-3.3, it is easy to derive a formula for the string $\mathsf{next}(\alpha)$ that follows α in cool-lex order. If $\alpha = a_1 \cdots a_n$ is a string and $i \leq j$, then $\mathsf{swap}(i,j)$ denotes the string obtained by *transposing* (*swapping*) a_i and a_j. First we describe $\mathsf{next}(\alpha)$ in terms of transpositions. If $\alpha = 1^s0^t\gamma \in \mathbf{L}$ and γ is either empty or begins with 1, then

$$\mathsf{next}(\alpha) = \begin{cases} \mathsf{swap}(i{+}1, s{+}1) & \text{if } \gamma = \epsilon \\ \mathsf{swap}(s{+}1, s{+}t{+}1) & \text{if } \gamma = 1,\ a_{s+t+2} = 1, \text{ or } \beta \notin \mathbf{L} \\ \mathsf{swap}(s{+}t{+}1, s{+}t{+}2)\ \mathsf{swap}(i{+}1, s{+}1) & \text{otherwise} \end{cases} \tag{3.2}$$

where $\beta = \mathsf{swap}(s+t+1, s+t+2)$ (applied to α), and i is the minimum value such that $1^i01^{s-i}0^{t-1}\gamma \in \mathbf{L}$. Notice that $0 \leq i \leq s$ since the upper-bound follows from $\alpha \in \mathbf{L}$. The last line in (3.2) includes two disjoint swaps that are both applied to α, and the second has no effect when $i = s$. The above formula is also circular, since the first case describes how to change the last string in cool-lex order into the first string.

We can also express the formula for $\mathsf{next}(\alpha)$ using shifts and the same values of β and i in (3.2). If $\alpha = a_1 \cdots a_n$ and $i < j$, then let $\mathsf{shift}(j, i)$ be the result of replacing the substring $a_i \cdots a_j$ in α with $a_j a_i \cdots a_{j-1}$. In other words, $\mathsf{shift}(j, i)$ denotes the operation of *left-shifting* the jth symbol into position i. If $\alpha = 1^s 0^t \gamma \in \mathbf{L}$ and γ is either empty or begins with 1, then

$$\mathsf{next}(\alpha) = \begin{cases} \mathsf{shift}(s+t, i) & \text{if } \gamma = \epsilon \\ \mathsf{shift}(s{+}t{+}1, 1) & \text{if } \gamma = 1 \text{ or } a_{s+t+2} = 1 \text{ or } \beta \notin \mathbf{L} \\ \mathsf{shift}(s{+}t{+}2, i) & \text{otherwise.} \end{cases} \tag{3.3}$$

Again, the above formula works circularly, since the first case describes how to change the last string in cool-lex order into the first string. Also notice that the right movement followed by the down movement can be described by a single shift. This section has proven the following theorem.

Theorem 3.1. *Cool-lex order provides a cyclic Gray code for any bubble language $\mathbf{L} \subseteq \mathbf{B}_d(n)$. Furthermore, successive strings are obtained by one or two transpositions as determined by* (3.2)*, or by a single left-shift as determined by* (3.3)*.*

3.1. Layering.

We have given cool-lex Gray codes for any bubble language whose strings have a fixed-length and fixed-density. Since the Gray codes are cyclic, it is straightforward to obtain Gray codes for bubble languages whose strings have fixed-length and varying density. One way to do this is to concatenate the cool-lex Gray codes by increasing density. We can also obtain a cyclic Gray code by layering the even densities in increasing order followed by the odd densities in decreasing order. In both cases we obtain a Gray code so long as there is at least one string (the terminal string) in the language for each density.

More generally, if we partition $\mathbf{L}$ into its non-empty subsets whose strings have fixed-length and fixed-density, then each subset will contain a terminal string. Furthermore, its first and last strings in cool-lex order will differ by a constant amount. Therefore, we can obtain a (cyclic) Gray code for an arbitrary bubble language $\mathbf{L}$ so long as there exists a (cyclic) Gray code amongst its terminal strings. This leads to the following theorem.

Theorem 3.2. *If bubble language $\mathbf{L}$ has a (cyclic) Gray code for its terminal strings, then $\mathbf{L}$ has a (cyclic) Gray code.*

Although Theorem 3.2 guarantees the existence of a Gray code, it may not produce a Gray code that minimizes the difference between successive strings. In particular, the Hamming distance between successive strings can be reduced by reflecting every second fixed-density Gray code.

4. Algorithms.

In this section we provide simple recursive algorithms for generating bubble languages over $\mathbf{B}_d(n)$ in Algorithms 1. **Co-lex** generates the strings in co-lex order according to (2.2), and **Cool-lex** generates the strings in cool-lex order according to (3.1).

In these algorithms the current string is stored in an array of length n. The array is indexed from 1, and should be initialized to contain $1^d 0^{n-d}$. The array is modified by **swap**(i, j), which swaps the values stored at indices i and j. The initial call is to **Co-lex**$(d, n-d)$ or **Cool-lex**$(d, n-d)$, and **Visit**$()$ is called once for each string in $\mathbf{L}$. At the start of each recursive call the array contains $1^s 0^t \gamma$ where γ is either empty or begins with 1. The key to the algorithm is the routine **Oracle**(s, t) that returns the minimum value of j such that $1^{s-1} 0^{t-j} 1 0^j \gamma \in \mathbf{L}$. Each iteration of the for loop updates the current string from $1^s 0^t \gamma$ to $1^{s-1} 0^{t-i} 1 0^i \gamma$

```
Function Co-lex($s, t$)
  Visit()
  if $s > 0$ and $t > 0$
    $j :=$ Oracle($s, t$)
    for $i := t - 1$ to $j$
      swap($s, s + t - i$)
      Co-lex($s - 1, t - i$)
      swap($s, s + t - i$)
```

```
Function Cool-lex($s, t$)
  if $s > 0$ and $t > 0$
    $j :=$ Oracle($s, t$)
    for $i := t - 1$ to $j$
      swap($s, s + t - i$)
      Cool-lex($s - 1, t - i$)
      swap($s, s + t - i$)
  Visit()
```

Algorithms 1: **Co-lex** and **Cool-lex** generate a bubble language over $\mathbf{B}_d(n)$ in co-lex order and cool-lex order, respectively.

using a single swap. Following a recursive call, the current string is restored to $1^s0^t\gamma$ by undoing this swap. The only difference between the two routines is the location of the **Visit**().

Co-lex and **Cool-lex** reduce the complexity of generating each bubble language over $\mathbf{B}_d(n)$ to the complexity of implementing each **Oracle**(s, t). Since every recursive call of **Co-lex**(s, t) visits a string in the bubble language, we obtain the following theorem.

Theorem 4.1. *If the total amount of computation required by all calls to* **Oracle**(s, t) *for a given bubble language* $\mathbf{L} \subseteq \mathbf{B}_d(n)$ *is proportional to the number of strings in* $\mathbf{L}$*, then* ***Co-lex***$(d, n - d)$ *generates* $\mathbf{L}$ *in constant amortized time.*

In [22] and [23] we present efficient oracles for each of the bubble languages listed in Section 2, and this allows each language to be generated in constant amortized time. The first of these two articles also explains how **Cool-lex** can be augmented to output the run-length representation of each string, as well as the direct Gray code changes (as transpositions or shifts) between successive strings.

5. Examples of Bubble Languages.

This section shows that each of the combinatorial objects mentioned in Section 2.1 can be naturally represented by a bubble language. For related concepts, such as strings with forbidden 01^k or 10^k, we will focus on the one that is a first-01 bubble language.

Combinations. An (n, d)-combination (or d-subset of an n-set) can be thought of as any string of length n and density d. Clearly combinations satisfy both of the bubble properties. Combinations were the first object to be studied in the context of cool-lex order [18] [19].

Forbidden substrings: Strings avoiding the forbidden substring 01^k for any fixed value of k form a first-01 bubble language. This is because swapping the first 01 to 10 cannot introduce the forbidden substring. On the other hand, these languages do not have the property that any 01 can be replaced by 10. For example, 0101 avoids 011, but replacing its second 01 by 10 gives the string 0110, which does contain 011. Similarly, strings with forbidden substrings of the form 10^k form a first-10 bubble language.

Inversions: An inversion with respect to 1^*0^* in a string $a_1 \cdots a_n$ 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). Replacing any 01 by 10 reduces the number of inversions by one, so strings with at most k inversions form a first-01 bubble language. Similarly, strings with at most k inversions with respect to 0^*1^* form a first-10 bubble language.

Transpositions: Another way to look at a string with k inversions is that it requires k adjacent-transpositions to sort into the form 1^*0^*. If we remove the "adjacent" criteria, then we can consider a bound k on the number of transpositions required to sort a string into the form 1^*0^*. 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. Since swapping any 01 with 10 does not increase the number of transpositions required to sort it, strings with at most k transpositions to sort it into the form 1^*0^* are a first-01 bubble language. Similarly, strings with at most k transpositions to sort it into the form 0^*1^* are a first-10 bubble language.

Strings $\geq \omega$: Consider a string α that is greater than or equal to a fixed ω in the usual lexicographical sense. Clearly, swapping any 01 to 10 in a string α will only make it lexicographically larger. Thus, the language of strings that are greater than or equal to ω form a first-01 bubble language. Similarly, strings that are less than or equal to ω form a first-10 bubble language.

Reversible strings: Consider a string α that is greater than (or equal to) its reversal. If we swap the first 01 to 10 then clearly α becomes larger while its reversal gets smaller in the usual lexicographical sense. Thus, strings that are greater than (or equal to) their reversal form a first-01 bubble language. Similarly, strings that are less than (or equal to) their reversal form a first-10 bubble language. Such equivalence classes of strings have also been called neckties [21].

Complemented reversible strings: In addition to reversal, we can also consider equivalence under complements (replacing 0s by 1s and vice versa). Observe that the terminal string 1^d0^{n-d} is only greater than or equal to its complemented reversal if $d \geq \lceil n/2 \rceil$. If this condition is satisfied, then the first 01 will either occur completely in the first half of the string or it will be split over the middle. Thus, if a string α that is greater than or equal to its complemented reversal then swapping the first 01 to 10 will clearly maintain this property. Therefore, strings that are greater than or equal to their complemented reversal form a first-01 bubble language. Similarly, strings that are less than or equal to their complemented reversal form a first-10 bubble language. Furthermore, these inequalities can be replaced by strict inequalities so long as $d > n/2$.

Necklaces and Lyndon words: Necklaces are equivalence classes of strings under rotation. If we choose the lexicographically largest (or smallest) element as the representative then we will show that they form a bubble language. Using the lexicographically smallest element as representative, the aperiodic necklaces are known as Lyndon words.

Consider a necklace $\alpha = a_1 \cdots a_n$ using the lexicographically largest representative. If $\alpha \neq 1^d0^{n-d}$, then suppose that the first 01 appears in positions j and $j+1$. Note that $a_1 \cdots a_i = 1^s0^t$ for some $s, t > 0$. If we swap the first 01 to 10 then the resulting string β is larger than α. We analyze what happens to the rotations of α in four cases, where r_i denotes the string $a_i \cdots a_n a_1 \cdot a_{i-1}$:

- ▷ $r_2, \cdots, r_{j-1}$: The prefix of these rotations before the swapped bits will clearly be less than the prefix of the same length in α. Thus, these rotations will still be smaller than $\alpha < \beta$ after the swap has occurred.
- ▷ r_j: If $s > 1$, then this case is trivial since β will start 11 while r_j with the swap will start 10. If $s = 1$, then since $\alpha \geq r_j$, we must have $a_{j+2} \cdots a_{2j} = 0^t$. Thus, β will have prefix $10^{t-1}1$ while the rotation r_i with the swap will have prefix 10^{t+1}.
- ▷ r_{j+1}: This rotation will start with 0 after the swap and thus is clearly less than $\alpha < \beta$.
- ▷ $r_{j+2} \cdots r_n$: For these rotations the swap will occur later in the string than α. Thus, since α is greater than or equal to each rotation, β will be greater than each such rotation after the swap has occurred.

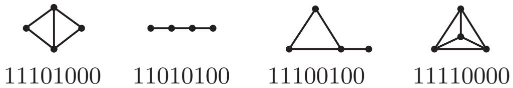
In each case β is strictly larger than each of its rotations, and so β is an aperiodic necklace. Thus, (aperiodic) necklaces using the lexicographically largest representation form a first-01 bubble language. Similarly, (aperiodic) necklaces using the lexicographically smallest representation form a first-10 bubble language.

Dyck words: A k-ary Dyck word is a binary string with d 1s and $d(k-1)$ 0s such that every prefix has $\leq k-1$ 0s for every 1. k-ary Dyck words are known to be equivalent to k-ary trees with d internal nodes. When $k = 2$, Dyck words are counted by the Catalan numbers and are equivalent to balanced parentheses strings (see Stanley [25] for 177 combinatorial objects counted by the Catalan number). Dyck words are a first-01 bubble language because swapping any 01 by 10 cannot decrease the number of 1s in a prefix.

Ordered forests: Ordered forests containing k trees are in correspondence with balanced parentheses strings with k balanced prefixes. A *balanced prefix* in a string α is any non-empty prefix of α that contains the same number of 1s as 0s. Notice that swapping a 01 by 10 can only increase the number of balanced prefixes in α when α has a prefix containing j copies of 1 and $j+1$ copies of 0. Since no such prefix exists in a balanced parentheses string, then balanced parentheses strings with at most k balanced prefixes are a first-01 bubble language.

Linear extensions of a B-poset: Consider $\alpha = a_1 \cdots a_n$ with d ones where the ith one appears within the first ℓ_i positions, for each $1 \leq i \leq d$. Such strings are in correspondence with the linear-extensions of a B-posets (see Pruesse and Ruskey [15]). Special cases include k-ary Dyck words, which are obtained by using $n = kd$ and $\ell_i = k(i-1)+1$ for $1 \leq i \leq d$. Swapping any 01 with 10 cannot decrease the density of any prefix, so the linear extensions of a B-poset form a first-01 bubble language.

Connected unit interval graphs: A unit interval graph with n vertices can be represented by a balanced parentheses string α of length $2n$. The vertex v_i corresponds to the interval between the ith one and the ith zero in α. If $i < j$, then v_i and v_j are adjacent if their intervals overlap, i.e., if the jth one appears before the ith zero. In a connected unit interval graph, its balanced parentheses string has exactly one balanced prefix. The four distinct connected unit interval graphs with four vertices appear below along with their string representations.



Observe that two balanced parentheses strings give isomorphic unit interval graphs if they are complemented reversals of one another. For example, the strings 11011000 and 11100100 give the same connected unit interval graph. Conversely, it has been shown that connected unit interval graphs can be represented uniquely by balanced parentheses strings with exactly one balanced prefix that are greater than or equal to their complemented reversals (see Saitoh et al [20]). Since balanced parentheses with one balanced prefix and binary strings greater than or equal to their complemented reverse with $d = n/2$ are both first-01 bubble languages, then the closure of first-01 bubble languages under intersections from Lemma 2.1 implies that connected unit interval graphs are also a first-01 bubble language.

0-1 knapsack: Given a knapsack with capacity C and a set of n items with non-decreaseing weights $w_1 w_2 \cdots w_n$, a feasible packing corresponds to a subset of the items whose total weight does not exceed C. Such a packing can be represented by a binary string $\alpha = a_1 \cdots a_n$ where a 1 in position i represents that the packing contains item i. Given a feasible packing α, clearly swapping any 01 with 10 will give rise to a new packing that does not increase in weight. Thus, feasible solutions to a 0-1 knapsack problem form a first-01 bubble language. Feasible solutions to 0-1 knapsack problems include many interesting special cases, as illustrated by Figure 1.1.

REFERENCES

[1] S. Bacchelli, E. Barcucci, E. Grazzini, and E. Pergola. Exhaustive generation of combinatorial objects by ECO. *Acta Informatica*, 40(8):585–602, July 2004.

[2] E. R. Canfield and S. G. Williamson. A loop-free algorithm for generating the linear extensions of a poset. *Order*, 12(1):57–75, 1995.

[3] F. Chung, P. Diaconis, and R.L. Graham. Universal cycles for combinatorial structures. *Discrete Mathematics*, 110:43–59, 1992.

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

[5] F. Gray. Pulse code communication. *U.S. Patent 2,632,058*, 1947.

[6] D. E. Knuth. *The Art of Computer Programming*, volume 4 fasicle 2: Generating All Tuples and Permutations. Addison-Wesley, errata (updated 10/02/2008) edition, 2005. ISBN 0-201-85393-0.

[7] D. E. Knuth. *The Art of Computer Programming*, volume 4 fasicle 3: Generating All Combinations and Partitions. Addison-Wesley, errata (updated 10/02/2008) edition, 2005. ISBN 0-201-85394-9.

[8] D. E. Knuth. *The Art of Computer Programming*, volume 4 fasicle 4: Generating All Trees, History of Combinatorial Generation. Addison-Wesley, errata (updated 10/02/2008) edition, 2006. ISBN 0-321-33570-8.

[9] J. F. Korsh and P. S. LaFollette. Loopless generation of linear extensions of a poset. *Order*, 18(2):115–126, 2002.

[10] J. F. Korsh and S. Lipschutz. Generating multiset permutations in constant time. *Journal of Algorithms*, 25:321–335, 1997.

[11] B. LaBounty-Lay, A. Bechel, and A. Godbole. Universal cycles of discrete functions. *Preprint (arXiv:0805.1672)*, 2008.

[12] Y. Li and J. Sawada. Gray codes for reflectable languages. *Information Processing Letters*, 109(5):296–300, 2009.

[13] E. Moreno. De Bruijn sequences and de Bruijn graphs for a general language. *Inf. Process. Lett.*, 96(6):214–219, 2005.

[14] G. Pruesse and F. Ruskey. Gray codes from antimatroids. *Order*, 10:239–252, 1993.

[15] G. Pruesse and F. Ruskey. Generating linear extensions fast. *SIAM Journal on Computing*, 23(2):373–386, April 1994.

[16] F. Ruskey. Generating linear extensions of posets by transpositions. *Journal of Combinatorial Theory (B)*, 54:77–101, 1992.

[17] F. Ruskey, J. Sawada, and A. Williams. Fixed-density de Bruijn sequences. *(submitted)*, 2010.

[18] F. Ruskey and A. Williams. Generating combinations by prefix shifts. In *COCOON '05: Computing and Combinatorics, 11th Annual International Conference*, volume 3595 of *Lecture Notes in Computer Science*, pages 570–576, Kunming, China, 2005. Springer-Verlag.

[19] F. Ruskey and A. Williams. The coolest way to generate combinations. *Discrete Mathematics*, 309(17):5305–5320, September 2009.

[20] T. Saitoh, K. Yamanaka, M. Kiyomi, and R. Uehara. Random generation and enumeration of proper interval graphs. In *WALCOM '09: Third International Workshop on Algorithms and Computation*, volume 5431 of *Lecture Notes in Computer Science*, pages 177–189. Springer Berlin / Heidelberg, 2009.

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

[22] J. Sawada and A. Williams. Efficient oracles for generating binary bubble languages. *(submitted)*, 2010.

[23] J. Sawada and A. Williams. A Gray code for fixed-density necklace and Lyndon words in constant amortized time. *(submitted)*, 2010.

[24] R. Sedgewick. Permutations generation methods. *ACM Comput. Surv.*, 9(2):137–164, 1977.

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

[26] T. Takaoka. An O(1) time algorithm for generating multiset permutations. In *ISAAC '99: Algorithms and Computation, 10th International Symposium*, volume 1741 of *Lecture Notes in Computer Science*, pages 237–246, Chennai, India, 1999. Springer.

[27] T. Ueda. Gray codes for necklaces. *Discrete Mathematics*, 219(1-3):235–248, 2000.

[28] V. Vajnovszki. A loopless algorithm for generating the permutations of a multiset. *Theoretical Computer Science*, 2(307):415–431, 2003.

[29] V. Vajnovszki. Gray code order for lyndon words. *Discrete Mathematics and Theoretical Computer Science*, 9(2):145–152, 2007.

[30] T. Walsh. Generating Gray codes in o (1) worst-case time per word. *Lecture Notes in Computer Science*, 2731:73–88, 2003.

[31] T.M.Y. Wang and C. Savage. A Gray code for necklaces of fixed density. *SIAM Journal on Discrete Mathematics*, 9(4):654–673, 1996.

[32] A. Williams. Loopless generation of multiset permutations using a constant number of variables by prefix shifts. In *SODA '09: The Twentieth Annual ACM-SIAM Symposium on Discrete Algorithms*, New York, New York, USA, 2009.

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

[34] L. Xiang, K. Cheng, and K. Ushijima. Efficient generation of Gray codes for reflectable languages. In *ICCSA 2010: Computational Science and Its Applications*, volume 6019 of *Lecture Notes in Computer Science*, pages 418–426. Springer Berlin / Heidelberg, 2010.