Ranking and Loopless Generation of k-ary Dyck Words in Cool-lex Order

Stephane Durocher[1], Pak Ching Li[1], Debajyoti Mondal[1], and Aaron Williams[2]

[1] Department of Computer Science, University of Manitoba
[2] Department of Mathematics and Statistics, Carleton University
durocher@cs.umanitoba.ca, lipakc@cs.umanitoba.ca
jyoti@cs.umanitoba.ca, haron@uvic.ca

Abstract. A binary string B of length $n = kt$ is a k-ary Dyck word if it contains t copies of 1, and the number of 0s in every prefix of B is at most $k-1$ times the number of 1s. We provide two loopless algorithms for generating k-ary Dyck words in cool-lex order: (1) The first requires two index variables and assumes k is a constant; (2) The second requires t index variables and works for any k. We also efficiently rank k-ary Dyck words in cool-lex order. Our results generalize the "coolCat" algorithm by Ruskey and Williams (*Generating balanced parentheses and binary trees by prefix shifts* in CATS 2008) and provide the first loopless and ranking applications of the general cool-lex Gray code by Ruskey, Sawada, and Williams (*Binary bubble languages and cool-lex order* under review).

1 Background

1.1 k-ary Dyck Words

Let $\mathbb{B}(n,t)$ be the set of binary strings of length n containing t copies of 1. A string $B \in \mathbb{B}(kt,t)$ is a *k-ary Dyck word* if the number of 0s in each prefix is at most $k-1$ times the number of 1s. Let $\mathbb{D}_k(t)$ be the set of k-ary Dyck words of length kt. For example, the k-ary Dyck words with $k = t = 3$ are given below

$$\mathbb{D}_3(3) = \{111000000, 110100000, 101100000, 110010000, 101010000, 100110000, 110001000, 101001000, 100101000, 110000100, 101000100, 100100100\}.$$

The k-ary Dyck words of length kt have simple bijections with a number of combinatorial objects including k-ary trees with t internal nodes [2, 3]. The 2-ary Dyck words are known as *balanced parentheses* when 1 and 0 are replaced by '(' and ')' respectively, and the cardinality of $\mathbb{D}_2(t)$ is the tth Catalan number.

A simple property of k-ary Dyck words is that they can be "separated" according to the following remark. We let $\alpha\beta$ denote the concatenation of the binary strings α and β, and we say that α and β have the same *content* if they have equal length and an equal number of 1s.

Remark 1. If $\alpha\beta, \gamma\delta \in \mathbb{D}_k(t)$ and α and γ have the same content, then $\alpha\delta, \beta\gamma \in \mathbb{D}_k(t)$. In other words, prefixes (or suffixes) of k-ary Dyck words with the same content can be separated and recombined.

1.2 Combinatorial Generation

Many computational problems require iterating through combinatorial objects of a given type and size without duplication. Generation algorithms store one object in a data structure, and create successive objects by modifying its contents. *Constant-amortized time (CAT)* and *loopless* algorithms create successive objects in amortized O(1)-time and worst-case O(1)-time, respectively. Memory for input parameters and the aforementioned data structure are fixed expenses, and the algorithm's remaining variables are *additional variables*. *Index variables* have values in $\{1, 2, \ldots, n\}$ when generating combinatorial objects of size $O(n)$.

Successive objects created by loopless algorithms differ by a constant amount (in the chosen data structure) and the resulting order of objects is a *Gray code*. If successive objects differ by operation 'x', then the order is an *'x' Gray code*; in a *2-'x' Gray code* successive objects differ by at most two applications of 'x'. In a *cyclic* Gray code the last object differs from the first object in same way.

Suppose $B = B_1B_2 \cdots B_n$ is a binary string of length n and $1 \leq i \leq j \leq n$. Informally, $\mathsf{swap}(B, i, j)$ exchanges the ith and jth bits of B, and $\mathsf{shift}(B, j, i)$ moves the jth bit of B leftwards into the ith position by moving the intermediate bits to the right. Formally, the *swap* and *shift* operations are defined as follows:

- $\mathsf{swap}(B, i, j) = B_1 \cdots B_{i-1}B_jB_{i+1} \cdots B_{j-1}B_iB_{j+1} \cdots B_n$, and
- $\mathsf{shift}(B, j, i) = B_1 \cdots B_{i-1}B_jB_iB_{i+1} \cdots B_{j-1}B_{j+1} \cdots B_n$.

When appropriate we shorten $\mathsf{swap}(B, i, j)$ to $\mathsf{swap}(i, j)$, and $\mathsf{shift}(B, j, i)$ to $\mathsf{shift}(j, i)$. Swaps are also known as *transpositions* with special cases including

- *adjacent-transpositions*: $\mathsf{swap}(i, i{+}1)$,
- *two-close-transpositions*: $\mathsf{swap}(i, i{+}1)$ or $\mathsf{swap}(i, i{+}2)$, and
- *homogeneous-transpositions*: $\mathsf{swap}(B, i, j)$ where $B_i = B_{i+1} = \cdots = B_{j-1}$.

Prefix-shifts are usually defined as operations of the form $\mathsf{shift}(j, 1)$. Swaps and prefix-shifts are efficient operations for binary strings stored in arrays and computer words, respectively.

Given an order of combinatorial objects, the *rank* of an object is its position in the order. *Ranking* determines the rank of a particular object in a given order, and *unranking* determines the object with a particular rank in a given order.

1.3 CoolCat Order

Balanced parentheses are among the most studied objects in combinatorial generation [3] but fewer results exist for k-ary Dyck words. Generation of $\mathbb{D}_k(t)$ was first discussed by Zaks [10]. A general result by Pruesse and Ruskey implies that $\mathbb{D}_k(t)$ has a 2-adjacent-transposition Gray code [4] and a result by Canfield and Williamson [1] proves that $\mathbb{D}_k(t)$ can be generated by a loopless algorithm[3]. More recently, Vajnovszki and Walsh [9] found a two-close transposition Gray

[3] Both results use that strings in $\mathbb{D}_k(t)$ correspond to linear-extensions of a poset with cover relations $a_1 \prec \cdots \prec a_t$, $b_1 \prec \cdots \prec b_{(k-1)t}$, and $a_i \prec b_{(k-1)(i-1)+1}$ for $1 \leq i \leq t$.

code and created a loopless algorithm that requires twelve if-statements and $O(n)$ additional variables stored in three additional arrays e, s, and p. Results on k-ary trees date back to Ruskey [5] and Trojanowski [8].

There are no prefix-shift Gray codes for $\mathbb{D}_k(t)$ (except when $k, t \leq 2$). However, the first bit of every k-ary Dyck word is 1, so we can instead define a *prefix-shift* as $\mathsf{shift}(i, 2)$ with the understanding that the redundant bit could be omitted from a computer word representation. Using this definition Ruskey and Williams [7] discovered an ordering of $\mathbb{D}_2(t)$ with the following properties:

- it is both a cyclic prefix-shift Gray code, and a cyclic 2-swap Gray code that uses at most one adjacent-transposition and one homogeneous -transposition,
- it can be generated by a loopless algorithm using only two if-statements and two additional index variables, and
- the ordering has an efficient ranking algorithm.

Furthermore, the Gray code can be created by the "successor rules" in Table 1. More specifically, every string in $\mathbb{D}_2(t)$ has a prefix that matches a unique rule in (1a)-(1d) which describes how the prefix is changed to obtain the next string. Table 1 uses exponentiation for symbol repetition, and the order for $\mathbb{D}_2(4)$ is:

$$10111000, 11011000, 11101000, 10110100, 11010100, 10101100, 11001100,$$
$$11100100, 10110010, 11010010, 10101010, 11001010, 11100010, 11110000.$$

For example, the matched prefix for 11001100 is 1^i0^j11 with $i = 2$ and $j = 2$. By (1a), $\mathsf{shift}(i{+}j{+}1, 2)$ (or $\mathsf{swap}(i{+}1, i{+}j{+}1)$) creates the next string 11100100. Similarly, the matched prefix for 11100100 is 1^i0^j10 with $i = 3$ and $j = 2$. By (1c), $\mathsf{shift}(i{+}j{+}2, 2)$ (or $\mathsf{swap}(2, i{+}1)$ $\mathsf{swap}(i{+}j{+}1, i{+}j{+}2)$) creates 10110010.

	Current Prefix†	Next Prefix	Shift	Swap(s)
(1a)	1^i0^j11	$1^{i+1}0^j1$	$(i{+}j{+}1, 2)$	$(i{+}1, i{+}j{+}1)$
(1b)	1^i0^j10 for $i = j$	$1^{i+1}0^{j+1}$	$(i{+}j{+}1, 2)$	$(i{+}1, i{+}j{+}1)$
(1c)	1^i0^j10 for $i > j$	$101^{i-1}0^j1$	$(i{+}j{+}2, 2)$	$(2, i{+}1)$ $(i{+}j{+}1, i{+}j{+}2)$
(1d)	1^i0^j for $i = j = t$	$101^{i-1}0^{j-1}$	$(i{+}j, 2)$	$(2, i{+}1)$

Table 1. Rules for generating balanced parentheses $\mathbb{D}_2(t)$ from [7]. Prefixes change according to (1a)-(1d) by the specified shift or the equivalent swap(s). $^\dagger j > 0$.

Rules (1a) and (1b) can be combined (see [7]) since they perform the same operation and Rule (1d) simply transforms the 'last' string in the cyclic Gray code into the 'first' string. The Gray code is also interesting because it generates $\mathbb{D}_k(t)$ according to a cyclic Gray code for $\mathbb{B}(kt, t)$ known as *cool-lex order*. That is, if $\alpha \in \mathbb{D}_k(t)$ comes before $\beta \in \mathbb{D}_k(t)$ in the cool-lex order of $\mathbb{B}(kt, t)$, then α comes before β in the Gray code defined by Table 1. The order and algorithm are named "CoolCat" after cool-lex order and the Catalan numbers.

Theorem 1 ([7]). *The balanced parentheses of length* $2t$ *in* $\mathbb{D}_2(t)$ *are generated in cool-lex order by the prefix-shift (or equivalent swap(s)) in Table 1.*

1.4 Bubble Languages and Cool-lex Order

A *bubble language*[4] is a set of binary strings $\mathbb{L} \subseteq \mathbb{B}(n,t)$ with the following property: If $B \in \mathcal{L}$ where $B = 1^i0^j01\gamma$ for some $j \geq 0$, then $1^i0^j10 \in \mathbb{L}$. In other words, they are sets of binary strings with the same content in which the leftmost 01 of any string can be replaced by 10 to give another string in the set. This definition comes from Ruskey, Sawada, and Williams who showed that bubble languages generalize many combinatorial objects including binary necklaces and solutions to knapsack problems [6]. They substantially generalized Theorem 1 by proving that cool-lex order provides a cyclic Gray code for any bubble language. In particular, the successor rules in Table 2 generate all of these Gray codes.

Lemma 1 ([6]). *The k-ary Dyck words in $\mathbb{D}_k(t)$ are a bubble language. Furthermore, the k-ary Dyck prefixes in $\mathbb{D}_k(t,s)$ (see Section 4) are a bubble language.*

Proof. Replacing 01 by 10 cannot decrease the number of 1s in a string's prefix. □

	Current String†	Next String‡	Shift	Swap(s)
(2a)	$1^i0^j11\gamma$	$1^{i+1}0^j1\gamma$	$(i{+}j{+}1, 1)$	$(i{+}1, i{+}j{+}1)$
(2b)	$1^i0^j10\gamma$ for $1^i0^{j+1}1\gamma \notin \mathbb{L}$	$1^{i+1}0^{i+1}\gamma$	$(i{+}j{+}1, 1)$	$(i{+}1, i{+}j{+}1)$
(2c)	$1^i0^j10\gamma$ for $1^i0^{j+1}1\gamma \in \mathbb{L}$	$1^h01^{i-h}0^j1\gamma$	$(i{+}j{+}2, h{+}1)$	$(h{+}1, i{+}1)$ $(i{+}j{+}1, i{+}j{+}2)$
(2d)	1^i0^j	$1^g01^{i-g}0^{j-1}$	$(i{+}j, g{+}1)$	$(g{+}1, i{+}1)$
(2e)	1^i0^j1	$1^{i+1}0^j$	$(i{+}j{+}1, 1)$	$(i{+}1, i{+}j{+}1)$

Table 2. Rules for generating a bubble language $\mathbb{L}$ from [6]. Strings change according to (2a)-(2e) by the specified shift or equivalent swap(s). $^\dagger j > 0$. $^\ddagger h$ is the minimum value such that $1^h01^{i-h}0^j1\gamma \in \mathbb{L}$ and g is the minimum value such that $1^g01^{i-g}0^{j-1} \in \mathbb{L}$.

Theorem 2 ([6]). *The strings in any bubble language are generated in cool-lex order by the shift (or equivalent swap(s)) in Table 2.*

We will examine how this result applies to k-ary Dyck words later in this article. In the meantime, observe that the rules in Table 2 refer to entire strings, and not just specific prefixes as in Table 1. This is due to the fact that bubble languages do not necessarily have the separability property mentioned in Remark 1. Also note that Table 2 produces a shift Gray code that is not necessarily a prefix-shift Gray code. On the other hand, the Gray code is still a 2-swap Gray code using at most one adjacent-transposition and one homogeneous-transposition.

1.5 New Results

We apply Theorem 2 to obtain a simple set of successor rules that generate a cyclic prefix-shift Gray code of k-ary Dyck words in Section 2. Then we use

[4] These are called "binary fixed-density bubble languages" in [6].

the Gray code as the basis for two loopless generation algorithms that store the current string in an array in Section 3. The first algorithm works for constant k and requires only two additional index variables. The second algorithm works for arbitrary k and requires four if-statements and one array of $O(n)$ additional index variables. In Section 4 we show how the Gray code can be efficiently ranked and unranked. With respect to the existing literature these results include

- the first prefix-shift Gray code for k-ary Dyck words [6],
- the first loopless algorithm for generating k-ary Dyck words that uses O(1) additional index variables (when k is constant),
- a simpler loopless algorithm for generating k-ary Dyck words using 1/3 the if-statements and additional arrays as [9] (when k is arbitrary), and
- the first order of k-ary Dyck words that has a loopless generation algorithm as well as efficient ranking and unranking algorithms.

Our results also include the first application of bubble languages to loopless generation and efficient ranking and unranking. Due to the generalization from "CoolCat" to k-ary Dyck words, we name the order and algorithms "CoolKat".

2 CoolKat Order

In this section we specialize the cool-lex Gray code for bubble languages to the special case of k-ary Dyck words of length kt. In particular, Theorem 3 will prove that k-ary Dyck words can be generated cyclically using the rules in Table 3. The resulting "CoolKat" order appears below for $\mathbb{D}_3(3)$

$$101100000, 110100000, 101010000, 100110000, 110010000, 101001000,$$
$$100101000, 110001000, 101000100, 100100100, 110000100, 111000000.$$

As in Table 1 for balanced parentheses, the rules in Table 3 refer to string prefixes and the stated shifts are prefix-shifts. Also, the rule (3d) refers only to the 'last' string $1^t0^{(k-1)t}$. In the second half of this section we optimize the swap rules in Table 3 for the array-based loopless algorithms in Section 3.

	Current Prefix†	Next Prefix	Shift	Swap(s)
(3a)	1^i0^j11	$1^{i+1}0^j1$	$(i{+}j{+}1, 2)$	$(i{+}1, i{+}j{+}1)$
(3b)	1^i0^j10 for $(k{-}1)i = j$	$1^{i+1}0^{j+1}$	$(i{+}j{+}1, 2)$	$(i{+}1, i{+}j{+}1)$
(3c)	1^i0^j10 for $(k{-}1)i > j$	$101^{i-1}0^j1$	$(i{+}j{+}2, 2)$	$(2, i{+}1)$ $(i{+}j{+}1, i{+}j{+}2)$
(3d)	1^i0^j for $i = t$, $j = (k-1)t$	$101^{i-1}0^{j-1}$	$(i{+}j, 2)$	$(2, i{+}1)$

Table 3. New rules for generating k-ary Dyck words $\mathbb{D}_k(t)$ in cool-lex order. These rules generalize those in Table 1 and specialize those in Table 2. $^\dagger j > 0$.

Theorem 3. *The k-ary Dyck words of length kt are generated in cool-lex order by the rules in Table 3.*

Proof. Since $\mathbb{L} = \mathbb{D}_k(t)$ is a bubble language [6], Theorem 2 implies that its strings are generated by Table 2. We now compare each rule in Table 2 to its proposed specialization in Table 3. In the comparison, recall that (2a)-(2e) refer to entire strings, whereas (3a)-(3d) refer to prefixes, and that $j > 0$ is always assumed in 1^i0^j1.

	Current	Next	Shift	Swap
(2a)	$1^i0^j11\gamma$	$1^{i+1}0^j1\gamma$	$(i{+}j{+}1, 1)$	$(i{+}1, i{+}j{+}1)$
(3a)	1^i0^j11	$1^{i+1}0^j1$	$(i{+}j{+}1, 2)$	$(i{+}1, i{+}j{+}1)$

If a k-ary Dyck word has prefix 1^i0^j11 and $j > 0$, then it must be that $i > 0$. Therefore, $\mathsf{shift}(i{+}j{+}1, 2)$ in (3a) is the special case of $\mathsf{shift}(i{+}j{+}1, 1)$ in (2a).

	Current	Next	Shift	Swaps
(2b)	$1^i0^j10\gamma$ for $1^i0^{j+1}1\gamma \notin \mathbb{L}$	$1^{i+1}0^{i+1}\gamma$	$(i{+}j{+}1, 1)$	$(i{+}1, i{+}j{+}1)$
(3b)	1^i0^j10 for $(k{-}1)i = j$	$1^{i+1}0^{j+1}$	$(i{+}j{+}1, 2)$	$(i{+}1, i{+}j{+}1)$

Suppose $1^i0^j10\gamma$ is a k-ary Dyck word. Remark 1 implies that $1^i0^{j+1}1\gamma$ is not k-ary Dyck word if and only if $(k{-}1)i = j$. Therefore, the condition "for $(k{-}1)i = j$" in (3b) is a special case of the condition "for $1^i0^{j+1}1\gamma \notin \mathbb{L}$" in (2b). Next observe that $i > 0$ since k-ary Dyck words must begin with the symbol 1. Therefore, $\mathsf{shift}(i{+}j{+}1, 2)$ in (3b) is the special case of $\mathsf{shift}(i{+}j{+}1, 1)$ in (2b).

	Current	Next‡	Shift	Swaps
(2c)	$1^i0^j10\gamma$ for $1^i0^{j+1}1\gamma \in \mathbb{L}$	$1^h01^{i-h}0^j1\gamma$	$(i{+}j{+}2, h{+}1)$	$(i{+}j{+}1, i{+}j{+}2)$ $(h{+}1, i{+}1)$
(3c)	1^i0^j10 for $(k{-}1)i > j$	$101^{i-1}0^j1$	$(i{+}j{+}2, 2)$	$(i{+}j{+}1, i{+}j{+}2)$ $(2, i{+}1)$

$^\ddagger h$ is the minimum value such that $1^h01^{i-h}0^j1\gamma \in \mathbb{L}$.

Suppose $1^i0^j10\gamma$ is a k-ary Dyck word. Remark 1 implies that $1^i0^{j+1}1\gamma$ is a k-ary Dyck word if and only if $(k{-}1)i > j$. Therefore, the condition "for $(k{-}1)i > j$" in (3c) is a special case of the condition "for $1^i0^{j+1}1\gamma \in \mathbb{L}$" in (2c). Next observe that Remark 1 implies that $h = 1$ is the minimum value such that $1^h01^{i-h}0^j1\gamma \in \mathbb{L}$. Therefore, the shifts and swaps in (3c) are special cases of those in (2c).

	Current	Next‡	Shift	Swaps
(2d)	1^i0^j	$1^g01^{i-g}0^{j-1}$	$(i{+}j, g{+}1)$	$(g{+}1, i{+}1)$
(3d)	1^i0^j for $i = t$, $j = (k-1)t$	$101^{i-1}0^{j-1}$	$(i{+}j, 2)$	$(2, i{+}1)$

$^\ddagger g$ is the minimum value such that $1^g01^{i-g}0^{j-1} \in \mathbb{L}$.

By similar reasoning as above, $g = 1$ is the minimum value such that $1^g01^{i-g}0^{j-1}$ is a k-ary Dyck word.

	Current	Next	Shift	Swaps
(2e)	1^i0^j1	$1^{i+1}0^j$	$(i{+}j{+}1, 1)$	$(i{+}1, i{+}j{+}1)$

This general rule for bubble languages does not apply to k-ary Dyck words because k-ary Dyck words cannot have 1 as the last symbol. □

2.1 Optimized Swap Rules

Table 4 gives swap rules that are equivalent to those in Table 3. In these rules, $\mathsf{swap}(i{+}1, i{+}j{+}1)$ is performed when creating the successor of every string (except $1^t0^{(k-1)t}$). This allows more compact array-based algorithms in Section 3.

	Current Prefix†	Next Prefix	Swap(s)
(4a)	1^i0^j11	$1^{i+1}0^j1$	$(i{+}1, i{+}j{+}1)$
(4b)	1^i0^j10 for $(k{-}1)i = j$	$1^{i+1}0^{j+1}$	$(i{+}1, i{+}j{+}1)$
(4c)	1^i0^j10 for $(k{-}1)i > j$	$101^{i-1}0^j1$	$(i{+}1, i{+}j{+}1)$ $(2, i{+}j{+}2)$
(4d)	1^i0^j for $i = t$, $j = (k-1)t$	$101^{i-1}0^{j-1}$	$(2, i{+}1)$

Table 4. Equivalent swap rules for generating k-ary Dyck words. These swap rules differ slightly from those in Table 3 and allow for a more efficient algorithm.$^\dagger j > 0$.

Corollary 1. *$\mathbb{D}_k(t)$ is generated in cool-lex order by the rules in Table 4.*

Proof. The swap(s) are identical to those in Table 3 except for (4c) below.

	Current Prefix	Next Prefix	Swap(s)
(3c)	1^i0^j10 for $(k{-}1)i > j$	$101^{i-1}0^j1$	$(i{+}j{+}1, i{+}j{+}2)$ $(2, i{+}1)$
(4c)	1^i0^j10 for $(k{-}1)i > j$	$101^{i-1}0^j1$	$(i{+}1, i{+}j{+}1)$ $(2, i{+}j{+}2)$

When $B \in \mathbb{D}_k(t)$ has prefix 1^i0^j10 with $j > 0$, then the relevant bit values are

$$B[2] = \begin{cases} 1 & \text{if } i > 1 \\ 0 & \text{if } i = 1, \end{cases} \quad B[i{+}1] = 0,\ B[i{+}j{+}1] = 1,\ \text{and } B[i{+}j{+}2] = 0.$$

If $i > 1$, then (3c) and (4c) both change the prefix to $101^{i-1}0^j1$. If $i = 1$, then (3c) and (4c) both change the prefix to $1^i0^j01 = 10^j1^{i-1}01$ via $\mathsf{swap}(i{+}j{+}1, i{+}j{+}2)$. □

3 Loopless Algorithms

In this section we provide two loopless algorithms for generating k-ary Dyck words in cool-lex order: coolkat (for "small" k) and coolKat (for "large" k).

3.1 Algorithm for Constant k

We begin with coolkat, which is a simple algorithm that uses only two additional index variables. We prove its correctness in Theorem 4 and then prove that it is loopless for constant k in Theorem 5.

Theorem 4. *Algorithm* $\mathsf{coolkat}(k, t)$ *generates each successive k-ary Dyck word of length kt in cool-lex order.*

```
Procedure coolkat(k, t)
 1:
 2: B ← array(1^t 0^{(k−1)t})
 3: x ← t
 4: y ← t
 5: visit()
 6: while x < k(t − 1) + 1
 7:    B[x] ← 0
 8:    B[y] ← 1
 9:    x ← x + 1
10:    y ← y + 1
11:    if B[x] = 0 then
12:       if x − 2 = k(y − 2) then
13:
14:          while B[x] = 0
15:             x ← x + 1
16:          end
17:
18:       else
19:          B[x] ← 1
20:          B[2] ← 0
21:          if y > 3 then
22:             x ← 3
23:          end
24:          y ← 2
25:       end
26:    end
27:    visit()
28: end
```

```
Procedure coolKat(k, t)
 1: A ← array(0^{t−2})
 2: B ← array(1^t 0^{(k−1)t})
 3: x ← t
 4: y ← t
 5: visit()
 6: while x < k(t − 1) + 1
 7:    B[x] ← 0
 8:    B[y] ← 1
 9:    x ← x + 1
10:    y ← y + 1
11:    if B[x] = 0 then
12:       if x − 2 = k(y − 2) then
13:          if B[x + 1] = 1 then
14:             A[y − 2] ← 0
15:          end
16:          A[y − 2] ← A[y − 2] + 1
17:          x ← x + A[y − 2]
18:       else
19:          B[x] ← 1
20:          B[2] ← 0
21:          if y > 3 then
22:             x ← 3
23:          end
24:          y ← 2
25:       end
26:    end
27:    visit()
28: end
```

Algorithms 1: coolkat(k, t) and coolKat(k, t) generate k-ary Dyck words of length kt in cool-lex order for any $k, t \geq 1$ (with $1^t 0^{(k-1)t}$ visited first).

Proof. We prove that the "main loop" on lines 6-28 always modifies B according to Table 4 by induction on the number of iterations. The first iteration visits $1^t 0^{(k-1)t}$ and the second iteration begins with $y = 2$, $x = 3$, and $B = 101^{t-1}0^{(k-1)t-1}$, which is correct by (4c). The second iteration provides a base case for the following main-loop invariant:

> If B has prefix $1^i 0^j 1$ for $j > 0$ on line 6, then $y = i+1$ and $x = i+j+1$.

Inductively suppose this invariant holds for the mth iteration and consider the next iteration. Lines 7-8 apply swap$(i{+}1, i{+}j{+}1)$, which is the first swap listed in each of (4a)-(4c). Lines 9-10 increment the additional variables to $y = i + 2$ and $x = i + j + 2$. Now consider the possible paths through the algorithm.

- If $B[x] = 1$ on line 11, then the mth string in cool-lex order had prefix $1^i 0^j 11$. By (4a) the successor has already been obtained by swap$(i{+}1, i{+}j{+}1)$. Furthermore, $y = i + 2$ and $x = i + j + 2$ correctly satisfy the invariant.

– If $B[x] = 0$ on line 11, then the mth string in cool-lex order had prefix 1^i0^j10.
  - If $x - 2 = k(y - 2)$ on line 12, then $j = (k - 1)i$ by simple algebra. By (4b) the successor has already been obtained by $\mathsf{swap}(i{+}1, i{+}j{+}1)$. Furthermore, $y = i + 2$ is correct. Since B now has prefix $1^{i+1}0^{j+1}$, x is greater than its current value of $i + j + 2$. The loop on line 14 scans the remainder of the B to determine the correct value of x.
  - If $x - 2 < k(y - 2)$ on line 12, then $j < (k - 1)i$. Lines 19-20 correctly apply $\mathsf{swap}(2, i + j + 2)$ by (4c) and change the prefix of B to $101^{i-1}0^j1$.
    * If $y > 3$ on line 21, then $i > 1$ and $x = 2$ is correctly set by line 22.
    * If $y = 3$ on line 21, then $i = 1$ and the current value of $x = i + j + 2$ is already correct.

    Finally, $y = 2$ is correctly set by line 24.

This induction continues until $B = 1^{t-1}0^{(k-1)(t-1)}10^{k-1}$ since this is the only string in $\mathbb{D}_k(t)$ for which $x \geq k(t-1)+1$ by the loop-invariant $x = i + j + 1$. By (4b) the successor of $1^{t-1}0^{(k-1)(t-1)}10^{k-1}$ is $1^t0^{(k-1)t}$, which was the first string visited. Therefore, $\mathsf{coolkat}(k, t)$ visits every string in $\mathbb{D}_k(t)$. □

Now we analyze coolkat. We need to show that the loop on line 14 runs a constant number of times when generating k-ary Dyck words for constant k. Towards this goal we present the following lemma.

Lemma 2. *If $1^i0^j10\gamma$ is a k-ary Dyck word and $j = (k - 1)i$, then γ does not have 0^{k-1} as a prefix.*

Proof. A k-ary Dyck word cannot have $1^i0^{(k-1)i}10^k$ as a prefix. □

Theorem 5. *Algorithm $\mathsf{coolkat}(k, t)$ uses two additional variables, and when k is a constant each successive string is created in worst-case $O(1)$-time.*

Proof. The algorithm uses the input values k and t, and stores the current k-ary Dyck word in the array B. Otherwise, the only additional variables are x and y. Therefore, the stated memory requirements are correct.

Next consider the run-time of creating each successive string in B. Notice that the only loop inside of the main loop on lines 6–28 is on 14. This loop is run when the current string stored in B at line 6 has a prefix equal to $1^i0^{(k-1)i}10$. By Lemma 2, the next k bits in B cannot all be 0. Therefore, the line 14 runs at most k times. If k is treated as a constant, then this loop can be replaced by a constant number of nested if-statements. Therefore, when k is a constant, successive strings are created in worst-case O(1)-time. □

3.2 Algorithm for Arbitrary k

To obtain a loopless algorithm for arbitrary k we perform the loop on line 14 with in O(1)-time by introducing an additional array of index variables A.

Theorem 6. $\mathsf{coolKat}(k, t)$ *is a loopless algorithm that generates each successive k-ary Dyck word of length kt in cool-lex order and uses only t index variables.*

Proof. Observe that coolkat and coolKat differ only in line 1 and lines 13-17. These lines are executed in coolKat when B begins the main-loop with a prefix of the form $1^i0^{(k-1)i}10$. By line 16, the A array is updated so that $A[i]$ contains the number of 0s that follow the prefix of the form $1^i0^{(k-1)i}1$. A formal proof of correctness requires an understanding of the recursive formulation of cool-lex order presented in Section 4 and is omitted. □

4 Ranking and Unranking

In this section we generalize k-ary Dyck words, discuss cool-lex order recursively, and then efficiently rank and unrank k-ary Dyck words in cool-lex order.

A string $B \in \mathbb{B}(s{+}t, t)$ is a *k-ary Dyck prefix* if the number of 0s in each prefix is at most $k{-}1$ times the number of 1s. Notice that k-ary Dyck prefixes with t $1s$ can have $s \leq (k{-}1)t$ $0s$, whereas k-ary Dyck words with t $1s$ must have $s = (k{-}1)t$ $0s$. Let $\mathbb{D}_k(t, s)$ be the k-ary Dyck prefixes in $\mathbb{B}(s{+}t, t)$. Thus,

$$\mathbb{D}_k(t, s) = \{B \in \mathbb{B}(s + t, t) \mid B0^{(k-1)t-s} \in \mathbb{D}_k(t)\}.$$

Let $\mathsf{N}_k(t, s)$ be the cardinality of $\mathbb{D}_k(t, s)$. Also let $v = (k{-}1)(t{-}1)$ in this section. The significance of this value is that every $B \in \mathbb{D}_k(t, s)$ has suffix 0^{s-v} if $s > v$.

Lemma 3. *$\mathsf{N}_k(t, s) = 0$ if $t = 0$, $\mathsf{N}_k(t, s) = 1$ if $t > 0$ and $s = 0$, and otherwise*

$$\mathsf{N}_k(t, s) = \begin{cases} \mathsf{N}_k(t{-}1, s) + \mathsf{N}_k(t, s{-}1) & \text{if } 1 \leq s \leq v; \\ \frac{1}{kt+1}\binom{kt+1}{t} & \text{if } v < s \leq (k-1)t. \end{cases}$$

Proof. $\mathbb{D}_k(0, s) = \emptyset$ and $\mathbb{D}_k(t, 0) = \{1^t\}$ if $t > 0$. If $1 \leq s \leq v$, then $B1 \in \mathbb{D}_k(t, s) \iff B \in \mathbb{D}_k(t{-}1, s)$ and $B0 \in \mathbb{D}_k(t, s) \iff B \in \mathbb{D}_k(t, s{-}1)$. Thus, $\mathsf{N}_k(t, s) = \mathsf{N}_k(t{-}1, s){+}\mathsf{N}_k(t, s{-}1)$. If $v < s \leq (k{-}1)t$, then all strings in $\mathbb{D}_k(t, s)$ end in 0 and $B \in \mathbb{D}_k(t, s) \iff B0^{(k-1)t-s} \in \mathbb{D}_k(t)$. Thus, $\mathsf{N}_k(t, s) = \frac{1}{kt+1}\binom{kt+1}{t}$ by the bijection between $\mathbb{D}_k(t)$ and k-ary trees with t internal nodes [3, 10]. □

Ruskey, Sawada, Williams [6] prove that the following recursive formula gives the cool-lex order of any bubble language $\mathbb{L}$. The formula is explained below.

$$\mathcal{C}(t, s, \gamma) = \begin{cases} \mathcal{C}(t{-}1, 1, 10^{s-1}\gamma), \ldots, \mathcal{C}(t{-}1, s{-}j, 10^j\gamma), 1^t0^s\gamma & \text{if } t > 0; \quad \text{(1a)} \\ 0^s\gamma & \text{if } t = 0. \quad \text{(1b)} \end{cases}$$

If $1^t0^s\gamma \in \mathbb{L}$ and γ doesn't begin with 0, then $\mathcal{C}(t, s, \gamma)$ is the cool-lex order for the strings in $\mathbb{L}$ with suffix γ. The "fixed-suffix" γ is extended in turn in (1) to $10^{s-1}\gamma, 10^{s-2}\gamma, \ldots, 10^j\gamma$ where j is the minimum value such that $10^j\gamma$ is the suffix of a string in $\mathbb{L}$. Notice that γ is extended by 10^i for decreasing i with one exception: The single string resulting from $i = s$ (namely, $1^t0^s\gamma = 1^{t-1}\underline{10^s\gamma} = \mathcal{C}(t{-}1, 0, \underline{10^s\gamma})$) is last instead of first. In fact, this is the only difference between cool-lex order and conventional "co-lex order" (see [3] for lexicographic orders). The entire cool-lex order for some $\mathbb{L}$ with $1^t0^s \in \mathbb{L}$ is $\mathcal{C}(t, s, \epsilon)$. Now we specialize cool-lex order to k-ary Dyck prefixes. Let the *coolKat order* for $\mathbb{L} = \mathbb{D}_k(t, s)$ be denoted $\mathcal{D}_k(t, s, \epsilon) = \mathcal{C}(t, s, \epsilon)$.

Lemma 4. *CoolKat order is* $\mathcal{D}_k(t,s,\gamma)=\epsilon$ *if* $t=0$*, and otherwise*

$$\mathcal{D}_k(t,s,\gamma)=\begin{cases}\mathcal{D}_k(t{-}1,1,10^{s-1}\gamma),\ldots,\mathcal{D}_k(t{-}1,s,1\gamma),1^t0^s & \text{if } s\le v;\\ \mathcal{D}_k(t{-}1,1,10^{s-1}\gamma),\ldots,\mathcal{D}_k(t{-}1,v,10^{s-v}\gamma),1^t0^s & \text{if } v<s\le(k{-}1)t.\end{cases}$$

Proof. $\mathbb{L}=\mathbb{D}_k(t,s)$ is a bubble language, so $\mathcal{D}_k(t,s,\gamma)$ follows from (1) by giving the minimum j such that 10^j is the suffix of a string in $\mathbb{L}$. If $s\le v$, then $j=0$ by $1^{t-1}0^s1\in\mathbb{L}$. If $v<s\le(k-1)t$, then $j=s-v$ by $1^{t-1}0^s10^{s-v}\in\mathbb{L}$. □

Now we efficiently rank and unrank k-ary Dyck prefixes with examples after Theorems 7 and 8. With respect to an ordered set of strings $\mathcal{L}=B_1,B_2,\ldots,B_m$, the *rank* of B_i is $\mathrm{rank}(B_i,\mathcal{L})=i{-}1$, and $\mathrm{unrank}(i{-}1,\mathcal{L})=B_i$ for $1\le i\le m$. For convenience let $\mathsf{R}(B,\mathcal{L})=\mathrm{rank}(B,\mathcal{L})+1$. Also let $\mathcal{D}_k(t,s)$ denote $\mathcal{D}_k(t,s,\epsilon)$.

Theorem 7. *If* $B=\alpha10^m\in\mathcal{D}_k(t,s)$ *for possibly empty* α *and* $m\ge0$*, then*

$$\mathsf{R}(B,\mathcal{D}_k(t,s))=\begin{cases}\mathsf{N}_k(t,s) & \text{if } B=1^t0^s;\\ \mathsf{R}(\alpha,\mathcal{D}_k(t{-}1,s{-}m))+\sum_{i=1}^{s-m-1}\mathsf{N}_k(t{-}1,i) & \text{if } B\neq1^t0^s \text{ and } s\le v;\\ \mathsf{R}(\beta,\mathcal{D}_k(t,v)) & \text{otherwise,}\end{cases}$$

where β *is the first* $t+v$ *bits of* B*.*

Proof. If $B=1^t0^s$, then $\mathsf{R}(B,\mathcal{D}_k(t,s))=\mathsf{N}_k(t,s)$ since B is last in $\mathcal{D}_k(t,s)$ by Lemma 4.

If $B\neq1^t0^s$ and $0\le s\le v$, then $\mathcal{D}_k(t{-}1,i)$ appears before B in $\mathcal{D}_k(t,s)$ for $1\le i\le s{-}m{-}1$ by Lemma 4.

If $s>v$, then by Lemma 4 each string of $\mathcal{D}_k(t,v)$ appears as a prefix of the corresponding string in $\mathcal{D}_k(t,s)$, i.e., $\mathcal{D}_k(t,s)=\mathcal{D}_k(t,v,0^{s-v})$. Therefore, $\mathsf{R}(B,\mathcal{D}_k(t,s))=\mathsf{R}(\beta,\mathcal{D}_k(t,v))$. □

With respect to an ordered set of strings $\mathcal{L}$, let $\mathsf{U}(x,\mathcal{L})=\mathrm{unrank}(x{-}1)$.

Theorem 8.

$$\mathsf{U}(x,\mathcal{D}_k(t,s))=\begin{cases}1^t0^s & \text{if } x=\mathsf{N}_k(t,s);\\ \mathsf{U}(x-\sum_{i=1}^{y}\mathsf{N}_k(t{-}1,i),\mathcal{D}_k(t{-}1,y{+}1))10^{s-y-1} & \text{if } x<\mathsf{N}_k(t,s) \text{ and } s\le v;\\ \mathsf{U}(x,\mathcal{D}_k(t,v))0^{s-v} & \text{otherwise,}\end{cases}$$

where y *is the largest integer such that* $x>\sum_{i=1}^{y}\mathsf{N}_k(t{-}1,i)$*.*

Proof. If $x=\mathsf{N}_k(t,s)$, then $\mathsf{U}(x,\mathcal{D}_k(t,s))$ is the last string in $\mathcal{D}_k(t,s)$ and by Lemma 4, $\mathsf{U}(x,\mathcal{D}_k(t,s))=1^t0^s$.

We now consider the case when $x<\mathsf{N}_k(t,s)$ and $0\le s\le v$. Let p be an integer, such that $\mathsf{U}(x,\mathcal{D}_k(t,s))$ is in $\mathcal{D}_k(t,p,10^{s-p})$. By Lemma 4, $x>\sum_{i=1}^{p-1}\mathsf{N}_k(t-1,i)$. It is now straightforward to observe that $y=p-1$. Therefore, $\mathsf{U}(x,\mathcal{D}_k(t,s))=\mathsf{U}(x-\sum_{i=1}^{y}\mathsf{N}_k(t-1,i),\mathcal{D}_k(t{-}1,y+1))10^{s-y-1}$.

The remaining case is $x<\mathsf{N}_k(t,s)$ and $s>v$. By Lemma 4, each string of $\mathcal{D}_k(t,v)$ appears as a prefix of the corresponding string in $\mathcal{D}_k(t,s)$, i.e., $\mathcal{D}_k(t,s)=\mathcal{D}_k(t,v,0^{s-v})$. Therefore, $\mathsf{U}(x,\mathcal{D}_k(t,s))=\mathsf{U}(x,\mathcal{D}_k(t,v))0^{s-v}$. □

We precompute and store the values of $\mathsf{N}_k(t,s)$ in a table so that for any value of k,t,s, we can obtain $\mathsf{N}_k(t,s)$ in $O(1)$ time. As a result we obtain $O(t+s)$-time ranking and unranking algorithms for k-ary Dyck words using Theorems 7 and 8, respectively. For example, the following table illustrates the first few values of $\mathsf{N}_k(t,s)$ for $k=5$. In the ranking and unranking process we assume that such tables for small fixed values of k are computed in advance. Thus for the corresponding precomputed values, we can obtain $\mathsf{N}_k(t,s)$ in $O(1)$ time.

$\mathsf{N}_5(t,s)$	s=0	s=1	s=2	s=3	s=4	s=5	s=6	s=7	s=8	s=9	s=10	s=11	s=12
$t=1$	1	1	1	1	1	1							
$t=2$	1	2	3	4	5	5	5	5	5				
$t=3$	1	3	6	10	15	20	25	30	35	35	35	35	35

We now compute $\mathsf{R}(100100010, \mathcal{D}_5(3,6))$ and $\mathsf{U}(16, \mathcal{D}_5(3,6))$ as follows:

$$\begin{aligned}
\mathsf{R}(100100010, \mathcal{D}_5(3,6)) &= \mathsf{R}(1001000, \mathcal{D}_5(2,5)) + \textstyle\sum_{i=1}^{6-1-1} \mathsf{N}_5(2,i) \\
&= \mathsf{R}(100, \mathcal{D}_5(1,2)) + \textstyle\sum_{i=1}^{5-3-1} \mathsf{N}_5(1,i) + \sum_{i=1}^{4} \mathsf{N}_5(2,i) \\
&= \mathsf{N}_5(1,2) + \textstyle\sum_{i=1}^{1} \mathsf{N}_5(1,i) + \sum_{i=1}^{4} \mathsf{N}_5(2,i) \\
&= 16. \\
\mathsf{U}(16, \mathcal{D}_5(3,6)) &= \mathsf{U}(16 - \textstyle\sum_{i=1}^{4} \mathsf{N}_5(2,i), \mathcal{D}_5(2,5))10^{6-4-1} \\
&= \mathsf{U}(2, \mathcal{D}_5(2,5))10 \\
&= \mathsf{U}(2, \mathcal{D}_5(2,4))0^{5-(2-1)(5-1)}10 \\
&= \mathsf{U}(2 - \textstyle\sum_{i=1}^{1} \mathsf{N}_5(1,i), \mathcal{D}_5(1,2))10^{4-1-1}010 \\
&= \mathsf{U}(1, \mathcal{D}_5(1,2))100010 \\
&= 100100010.
\end{aligned}$$

Acknowledgements. We thank Frank Ruskey for helpful conversations.

References

1. Canfield, E., Williamson, S.: A loop-free algorithm for generating the linear extensions of a poset. Order 12, 57–75 (1995)
2. Heubach, S., Li, N.Y., Mansour, T.: A garden of k-Catalan structures (2008), http://www.scientificcommons.org/43469719
3. Knuth, D.E.: The Art of Computer Programming: Generating all Trees and History of Combinatorial Generation, vol. 4. Addison-Wesley (February 2006)
4. Pruesse, G., Ruskey, F.: Generating the linear extensions of certain posets by transpositions. SIAM Journal on Discrete Mathematics 4(3), 413–422 (1991)
5. Ruskey, F.: Generating t-ary trees lexicographically. SIAM Journal on Computing 7(4), 424–439 (1978)

6. Ruskey, F., Sawada, J., Williams, A.: Binary bubble languages and cool-lex order. (under review) p. 13 pages (2010)
7. Ruskey, F., Williams, A.: Generating balanced parentheses and binary trees by prefix shifts. In: Proceedings of the 14th Computing: The Australasian Theory Symposium (CATS 2008), NSW, Australia. vol. 77, pp. 107–115 (January 22–25 2008)
8. Trojanowski, A.E.: Ranking and listing algorithms for k-ary trees. SIAM Journal on Computing 7(4), 492–509 (1978)
9. Vajnovszki, V., Walsh, T.: A loop-free two-close Gray-code algorithm for listing k-ary Dyck words. Journal of Discrete Algorithms 4(4), 633–648 (2006)
10. Zaks, S.: Generation and ranking of k-ary trees. Information Processing Letters 14(1), 44–48 (1982)