Cool-lex order and k-ary Catalan structures

Stephane Durocher [a], Pak Ching Li [a], Debajyoti Mondal [a], Frank Ruskey [b], Aaron Williams [c,*]

[a] *Department of Computer Science, University of Manitoba, Canada*
[b] *Department of Computer Science, University of Victoria, Canada*
[c] *Department of Mathematics and Statistics, McGill University, Canada*

A R T I C L E   I N F O

A B S T R A C T

For any given k, the sequence of k-ary Catalan numbers, $C_{t,k} = \frac{1}{kt+1}\binom{kt}{t}$, enumerates a number of combinatorial objects, including k-ary Dyck words of length $n = kt$ and k-ary trees with t internal nodes. We show that these objects can be efficiently ordered using the same variation of lexicographic order known as cool-lex order. In particular, we provide loopless algorithms that generate each successive object in O(1) time. The algorithms are also efficient in terms of memory, with the k-ary Dyck word algorithm using O(1) additional index variables, and the k-ary tree algorithm using O(t) additional pointers and index variables. We also show how to efficiently rank and unrank k-ary Dyck words in cool-lex order using O(kt) arithmetic operations, subject to an initial precomputation. Our results are based on the cool-lex successor rule for sets of binary strings that are bubble languages. However, we must complement and reverse $1/k$-ary Dyck words to obtain the stated efficiency.

1. Background

An important problem in discrete mathematics and theoretical computer science is the creation of efficient orders for combinatorial objects. The research area that studies this problem is *combinatorial generation*, and it is overviewed by Knuth in the most recent volume of *The Art of Computer Programming* [12]. In this article we will show that cool-lex order is an efficient order for k-ary Catalan objects. To frame our results, this section provides background on combinatorial generation, k-ary Catalan objects, and cool-lex order.
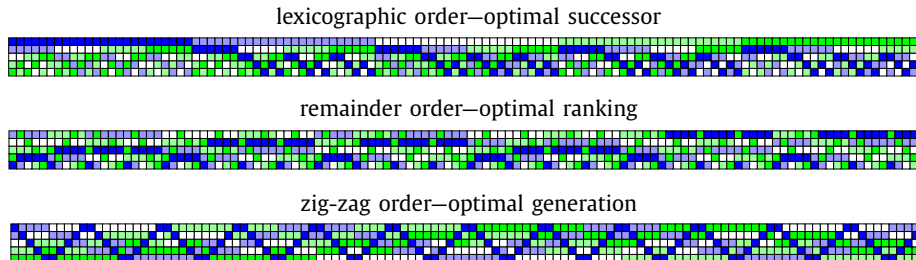
1.1. Combinatorial generation

To understand combinatorial generation, we explain each of the terms in "efficient orders for combinatorial objects" and then provide several examples. By 'combinatorial objects' we are referring to a set of objects of a particular size and type, such as binary strings of length n, permutations of $[n] = \{1, 2, \ldots, n\}$ in one-line notation, trees on n vertices, and so on. The size of the object is usually denoted by n, and the number of objects by m. An 'order' is a list in which each object in the set appears exactly once. Finally, we associate three interrelated operations with each order, for which we evaluate 'efficiency' according to the worst-case time complexity:

---

* Corresponding author.
 E-mail addresses: durocher@cs.umanitoba.ca (S. Durocher), lipakc@cs.umanitoba.ca (P.C. Li), jyoti@cs.umanitoba.ca (D. Mondal), ruskey@cs.uvic.ca
(F. Ruskey), haron@uvic.ca (A. Williams).

- A *successor algorithm* returns the object that follows a given object.
- A *ranking algorithm* returns the position, or *rank*, of a given object.
- A *generation algorithm* successively creates every object in the order.

Informally, we say that an order is 'optimal' with respect to one of these three algorithms if no other order of the same object has a known algorithm that is more efficient. To expand this discussion and to give a flavor of existing results, we consider three orders of permutations that each have one optimal algorithm. The following diagrams contain the $m = 120$ permutations of $n = 5$, where ■, ▨, □, ▤ and ▥ represent 1, 2, 3, 4 and 5, respectively.[1]

<div align="center">lexicographic order—optimal successor</div>

<div align="center">remainder order—optimal ranking</div>

<div align="center">zig-zag order—optimal generation</div>

The first order is *lexicographic order*. In this order, the permutations are ordered recursively, with lower values appearing first. For example, in the above diagram the symbol in the top row ranges from ■ to ▥, and this pattern repeats for the second row subject to the first symbol, and so on. In this order, the successor is found by identifying the shortest suffix whose first symbol is smaller than its second symbol, then by swapping this suffix's first symbol with its next largest symbol, and finally by reversing the suffix without the new first symbol. For example, in 1$3\overline{5}\overline{4}2$ we swap 3 and 4, and then reverse the resulting suffix without the new first symbol $5\overline{3}2$ to obtain the successor 1$4\overline{2}\overline{3}5$. Thus, in the diagram column ■▤▥▨□ is followed by ■▥▨□▤. When the permutation is stored in an array or doubly-linked list, this takes O(n) time and can easily be implemented using two additional variables. In general, optimal successor algorithms use $\Omega(n)$-time, with more complicated objects requiring $\omega(n)$-time. This complexity depends on the data structure used to store the combinatorial object, and preference is given to algorithms that require less additional memory. *Additional memory* refers to memory that is not used to store the combinatorial object, which is a fixed expense. In particular, an *index variable* can store integers up to O(n) and a *pointer variable* is a traditional pointer that can store an index that refers to some portion of the combinatorial object.

The second order, which we call *remainder order*, was used by Myrvold and Ruskey [15]. Informally, let (x, y) denote the swap of the xth and yth symbol of a string. For example, applying $(4, 2)$ to 456123 gives 416523. Swaps are also called transpositions. In remainder order, the ith permutation is obtained from the identity permutation by a series of $n - 1$ transpositions. The first indices of the transpositions are $n, n - 1, \ldots, 2$. The second indices are remainders when i is successively divided by $n, n - 1, \ldots, 2$, plus one. For example, here are the calculations for $i = 92$ and $n = 5$

$$92 \div 5 = 18 \qquad 18 \div 4 = 4 \qquad 4 \div 3 = 1 \qquad 1 \div 2 = 0$$
$$\text{remainder } \underline{2} \qquad \text{remainder } \underline{2} \qquad \text{remainder } \underline{1} \qquad \text{remainder } \underline{1}.$$

In this calculation, each successive quotient is used in the next division, and the divisors are in turn 5, 4, 3, 2. The underlined remainders (plus one) imply that the 92nd permutation for $n = 5$ is obtained from 12345 by successively applying the following transpositions: $(5, \underline{3}), (4, \underline{3}), (3, \underline{2}), (2, \underline{2})$. The resulting permutation is 14253, and thus the 93rd column in the diagram is ■▤▨▥□, since the first object has rank 0. Although this description is somewhat unorthodox, it directly translates into a simple *unranking algorithm*, which converts an integer i into the object of rank i. In remainder order, the unranking and ranking algorithms use O(n) arithmetic operations on values that can be as large as $n!$. These algorithms have interesting applications [15], but they do not provide an O(n)-time successor algorithm. (Although the successor can be computed by ranking, incrementing, and unranking, the arithmetic operations do not take constant time due to the size of the values.) In general, optimal ranking algorithms often use $\Omega(n)$ arithmetic operations on values as large as m, with complicated objects requiring $\omega(n)$ operations. Informally, the complexity of ranking and unranking is often stated in terms of time, with the issue of arithmetic on large values being understood. For some objects it is common to allow an initial pre-computation that is not counted against the ranking algorithm's complexity, although this is not necessary for remainder order.

The third order is *zig–zag order*, published by Johnson [10], Steinhaus [24], and Trotter [27] and known earlier to campanologists (see Knuth [12]). In this order, the smallest symbol repeatedly 'zigs' and 'zags' between the two ends of a permutation, pausing once at each end to allow the next smallest symbol that is not paused to 'zig' or 'zag'. For example, the diagram begins with columns ■▨□▤▥, ▨■□▤▥, ▨□■▤▥, ▨□▤■▥, ▨□▤▥■, □▨▤▥■, since ■ sweeps through the

---

[1] In these diagrams columns represent individual strings (i.e., permutations) when read from top-to-bottom, and the columns read from left-to-right represent the order of the strings.

initial permutation of the remaining symbols before pausing to allow □ to begin its sweep. Zig–zag order is a *Gray code* since successive objects differ by a constant amount. In particular, zig–zag order is an adjacent-transposition Gray code since successive objects differ by a transposition of adjacent symbols (i.e., $(x, x + 1)$). Moreover, it is *cyclic* since the last permutation and first permutation also differ by an adjacent-transposition. Zig–zag order can be generated by a *loopless algorithm*, meaning that successive objects are created in worst-case O(1) time (see Ehrlich's seminal paper on loopless algorithms [6]). To clarify this point, we note that successive objects do not have to be created from 'scratch' when they are generated. Instead, a single instance of the combinatorial object is created, and this instance is repeatedly modified to create the remaining instances in the order. Loopless generation algorithms require a Gray code order, but this condition is not sufficient. In particular, loopless algorithms for zig–zag order maintain extra information on the current permutation, including a 'zig–zag direction' variable and an index variable for each symbol. However, zig–zag order does not have an optimal successor algorithm since these additional variables cannot be efficiently reconstructed for an individual permutation. In general, loopless generation algorithms typically use $\omega(1)$ additional variables. For complicated objects, optimal generation algorithms often create objects in worst-case O(n) time. (Although we focus on worst-case analyses in this paper, we also mention that a common measure of efficiency for generation algorithms is amortized O(1) time, which is known as "CAT" for *constant amortized time*.)

Unfortunately, these three orders are each optimal for only one of the three algorithms in practice[2] and this type of trade-off is common in combinatorial generation. However, in some cases it is possible for a single order to be optimal for all three algorithms. An example is the eponymous *binary reflected Gray code* (*BRGC*) for binary strings of length n (see Gray [7]). The BRGC is shown below along with lexicographic order for $n = 7$ and $m = 128$, where ■ and □ are used for 0 and 1, respectively.

lexicographic order

binary reflected Gray code
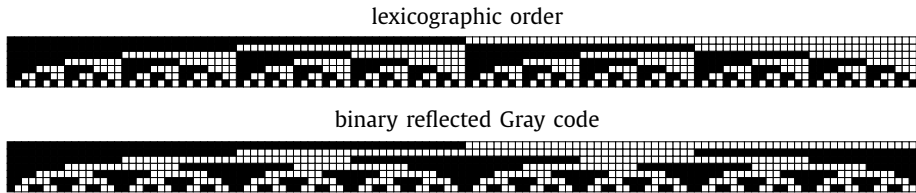
Both orders are created from two copies of their order for $n − 1$ bits, with 0 prefixed to the first copy and 1 prefixed to the second copy. However, in the BRGC the order of the strings in the second copy is reversed. The result is that successive strings in the BRGC differ in only one bit, while the overall order has a relatively simple structure.

1.2. k-ary Catalan structures

The Catalan numbers $C_t = \frac{1}{2t+1}\binom{2t}{t}$ enumerate a wide variety of combinatorial objects. Stanley updates a "Catalan Addendum" to *Enumerative Combinatorics* that lists 194 of these Catalan structures [23]. Some of these structures have natural generalizations that are enumerated by the *k-ary Catalan numbers* $C_{t,k} = \frac{1}{kt+1}\binom{kt}{t}$, where $C_t = C_{t,2}$. Heubach, Li, and Mansour maintain a 'garden' that currently includes 32 of these *k*-ary Catalan structures [8]. Two of the most well-known members of this garden are *k*-ary Dyck words and *k*-ary trees.

Let $\mathbb{B}(n, t)$ denote the set of binary strings of length n with weight (number of 1s) equal to t. 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. For example, the *k*-ary Dyck words with $k = t = 3$ are given below in lexicographic order

$$\mathbb{D}_3(3) = \{100100100, 100101000, 100110000, 101000100, 101001000, 101010000,$$

$$101100000, 110000100, 110001000, 110010000, 110100000, 111000000\},$$

where $\mathbb{D}_k(t)$ denotes the set of *k*-ary Dyck words of length kt. It is customary to store a *k*-ary Dyck word on a computer by using an array of length n. (Alternatively, a *k*-ary Dyck word can be stored in an array of length t by using the positions of 1s.) The number of strings in $\mathbb{D}_k(t)$ is $C_{t,k}$. When $k = 2$, the *k*-ary Dyck words are known as *Dyck words* or *balanced parentheses* since replacing 1 and 0 by '(' and ')', respectively, result in the strings of well-formed parentheses. For example, after this symbol substitution $\mathbb{D}_2(3)$ equals $\{()()(), ()(()), (())(), (()()), ((()))\}$.

---

[2] In theory, Mareš and Straka rank lexicographic order in O(n) time using O(1)-time bit operations on words of size $n!$ [14], and Williams gives a loopless algorithm by storing the permutation in a doubly-linked list whose nodes can have their two pointers interchanged [29].
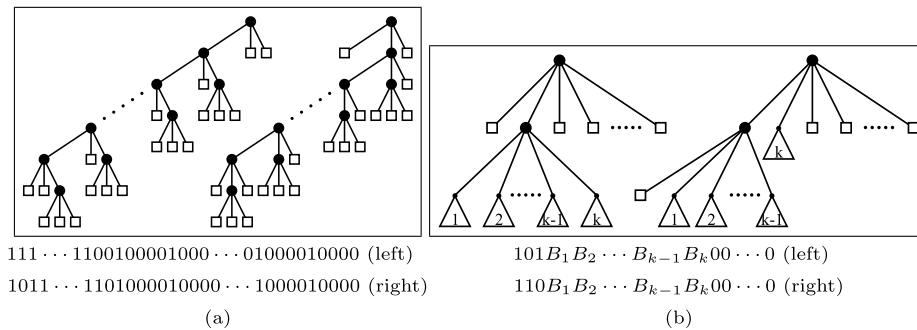
$111\cdots1100100001000\cdots01000010000$ (left)

$1011\cdots1101000010000\cdots1000010000$ (right)

(a)

$101B_1B_2\cdots B_{k-1}B_k00\cdots0$ (left)

$110B_1B_2\cdots B_{k-1}B_k00\cdots0$ (right)

(b)

Fig. 1. (a) The transposition $(2, \lceil\frac{t}{2}\rceil + 1)$ in the given 3-ary Dyck words of length $3t$ is equivalent to $\lceil\frac{t}{2}\rceil$ modifications in the corresponding 3-ary trees with t internal nodes. In particular, the children arrays for all of the nodes along the "upper path" of length $\lceil\frac{t}{2}\rceil$ are changed. Similar examples exist for any $k \geqslant 3$. (b) The adjacent-transposition $(2, 3)$ in the given k-ary Dyck word is equivalent to k modifications in the corresponding k-ary trees. In particular, every entry in the child array is changed for the second node visited in pre-order.

A *k-ary tree* is a rooted ordered tree in which every internal node has k children. The number of k-ary trees with t internal nodes is equal to $C_{t,k}$. For example, the 3-ary trees with 3 internal nodes are given below.
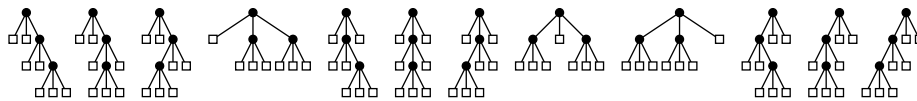
It is customary to represent a k-ary tree in a computer by having each node store an array of k pointers to its children. The trees above are ordered in lexicographic order according to a bijection to k-ary Dyck words described below.

The study of (k-ary) Catalan structures is simplified from a computational point of view by the existence of bijective correspondences between many of the structures that can be computed for each specific object in linear time. For example, a pre-order traversal of a k-ary tree with t nodes provides a k-ary Dyck word of length kt by recording a 1 for each internal node and a 0 for each leaf except the last. For this reason, an order for one structure provides a 'simultaneous' order for another structure using corresponding objects. More importantly, efficiently computable bijections allow efficient ranking and successor algorithms for one order to be converted to the other order. In particular, ranking k-ary Dyck words and ranking k-ary trees are essentially the same problem in terms of computational complexity.

Efficient generation of different (k-ary) Catalan structures provides more of a challenge than efficient ranking and successor. The main issue is that a pair of objects may differ by a constant amount using one structure, but this difference can become a non-constant amount after they are converted to another structure. For example, Fig. 1(a) shows that changing a k-ary Dyck word prefix from $111\cdots10$ to $1011\cdots1$ by a transposition can result in modifications to O(t) of the children arrays in the corresponding k-ary tree when $k \geqslant 3$. Similarly, Fig. 1(b) shows that changing a k-ary Dyck word prefix from 101 to 110 by an adjacent-transposition can result in O(k) modifications to a single children array in the corresponding k-ary tree. These examples illustrate that a Gray code for one (k-ary) Catalan structure does not necessarily provide a Gray code for another (k-ary) Catalan structure. Currently, the literature contains no example of a 'simultaneous' Gray code for k-ary Dyck words and k-ary trees. Furthermore, there is an even greater challenge in constructing 'simultaneous' loopless algorithms for k-ary Dyck words and k-ary trees.

In this article we also consider a lesser-known k-ary Catalan structure, which is essentially an alternate presentation of the seventh structure in the Catalan garden [8] involving "non-negative words". A string $B \in \mathbb{B}(kt, (k-1)t)$ is a *1/k-ary Dyck word* if the number of 1s in each prefix is at least $k-1$ times the number of 0s. For example, the $1/k$-ary Dyck words for $k = 3$ and $t = 6$ appear below

$$\mathbb{d}_3(3) = \{110110110, 111010110, 111100110, 110111010, 111011010, 111101010,$$

$$111110010, 110111100, 111011100, 111101100, 111110100, 111111000\},$$

where $\mathbb{d}_k(t)$ denotes the $1/k$-ary Dyck words of length kt. By comparing their definitions, it is clear that k-ary Dyck words and $1/k$-ary Dyck words are identical when $k = 2$. More generally, a string is in $\mathbb{d}_k(t)$ if and only if its complemented reverse is in $\mathbb{D}_k(t)$, where *complemented reverse* is the involution that complements the value and reverses the order of the bits in a string. For example, $111100110 \in \mathbb{d}_3(3)$ and $100110000 \in \mathbb{D}_3(3)$ are complemented reversals of each other. This correspondence is stated in Remark 1, and the above $1/k$-ary Dyck words are in lexicographic order according to their Dyck words.

Remark 1. $\mathbb{D}_k(t)$ and $\mathbb{d}_k(t)$ are in bijective correspondence by complementing the bits and reversing their order in each string.

From an algorithmic point of view, Remark 1 provides the option of indirectly generating k-ary Dyck words by implicitly generating $1/k$-ary Dyck words according to some efficient order. Gray codes and loopless algorithms for k-ary and $1/k$-ary Dyck words are interchangeable when the objects are stored in an array, or any data structure where modification times are unchanged by complemented reversal.

Historically, balanced parentheses are among the most studied objects in combinatorial generation [12], but fewer results exist for k-ary Dyck words. Generation and ranking of $\mathbb{D}_k(t)$ in lexicographic order was first discussed by Zaks [34]. A general result by Pruesse and Ruskey implies that $\mathbb{D}_k(t)$ has a 2-adjacent-transposition Gray code [16] and a result by Canfield and Williamson [4] proves that $\mathbb{D}_k(t)$ can be generated by a loopless algorithm.[3] More recently, Vajnovszki and Walsh [28] found a two-close-transposition Gray code and created a loopless algorithm (see Section 2 for a definition of two-close).

Results on k-ary trees in lexicographic order date back to Ruskey [17] and Trojanowski [26]. Trojanowski's result generates k-ary trees in lexicographic order of "stack permutations" giving an O(n^2)-time ranking and unranking algorithm. Baronaigien and Ruskey [3] showed how to generate k-ary trees using "A-order", achieving O(kn)-time and O$(kn \log n)$-time algorithms for ranking and unranking, respectively. Several algorithms have since been proposed with the goal of achieving efficient running times for tree generation, ranking and unranking simultaneously [1,2,17,33]. Each of these algorithms uses a precomputed table for ranking and unranking that takes O(kn^2)-time to construct. We follow the same approach in this paper, although an optimization is possible when $k = 2$. As the following table shows, this article provides the first linear-time ranking and unranking for k-ary trees (or equivalently k-ary Dyck words) for an order other than lexicographic order. To be clear, our ranking and unranking algorithms are only "linear-time" in the sense that they use O(kt) arithmetic operations on integers as large as $C_{t,k}$, and assuming O(1)-time access to a table of values which itself requires O(tn) arithmetic operations to precompute.

Order	Generation	Ranking	Unranking	References
Lexicographic order	O(n)	O(n^2)	O(n^2)	[25]
Reverse A-order	O(1)	O(kn)[*]	O$(kn \log n)$[*]	[3]
Lexicographic order	O(k)	O(n)[*]	O(n)[*]	[17]
Gray code order	O(1)	O(kn^2)[*]	O(kn^2)[*]	[1,33]
Reverse B-order	O(1)	O(kn)[*]	O(kn)[*]	[2]
Cool-lex order reverse complemented	O(1)	O(n)[*]	O(n)[*]	this article

Related work on generating k-ary trees in chronological order.
[*] Denotes that the time complexity is achieved using a precomputed table.

Several important results do not appear in this table since their assumed context is somewhat different. First, Kokosinski [13] gives a parallel algorithm to unrank k-ary trees in O(n) time that relies on a parallel architecture with kn processors and takes O(kn)-time to compute the auxiliary table. Second, Wu [30] recently proposed a new representation for trees known as an RD-sequence (for *right-distance*). Using this alternate representation, Wu gives a loopless algorithm that can rank and unrank k-ary trees in O(kn)-time without using any precomputed table. Finally, Wu, Chang, and Wang [32] and Wu, Chang, and Chang [31] give loopless algorithms and efficient ranking and unranking for non-regular trees with a prescribed branching sequence, respectively.

1.3. Cool-lex order

An interesting development in combinatorial generation was the discovery of the following successor algorithm for $\mathbb{B}(n, t)$ by Ruskey and Williams [19,21]. The ith *prefix-shift* moves the first i bits of a string once to the right (circularly).

> **Cool-lex successor for $\mathbb{B}(n, t)$:**
> Let i be the index of the first 01 substring's 1. Apply the $(i + 1)$st prefix-shift.
> (If there is no 010 or 011 substring, then perform the nth prefix-shift.)

For example, the algorithm applies the 7th prefix-shift to 110001010 since its first 01 substring ends at index $i = 6$. Thus, <u>110001010</u> is succeeded by <u>011000110</u> since the first 7 bits (underlined) move one index to the right (circularly). Equivalently, this modification complements four bits, $\overline{11}00\overline{01}010 = 011000110$, and in general the algorithm changes either two or four bits. Despite its simplicity, this simple algorithm (or 'rule') cyclically generates $\mathbb{B}(n, t)$ for all n and t. Furthermore, each application of the rule causes the first 01 substring to either move one index to the right, or to be 'reset' to the beginning of the string. Therefore, successive applications do not need to 'scan' the string to locate the first 01. These observations are the basis for an extremely simple loopless algorithm that generates the order using two additional index variables, two if-statements and zero else-statements. Knuth included cool-lex order as a last-minute addition to *The Art of*

---

[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 $1 \leqslant i \leqslant t$.

Computer Programming [12], and also provides an "incredibly efficient" implementation using computer words in his distribution of MMIX. By convention, $1^t 0^{n-t}$ is treated as the last string in cool-lex order, where exponentiation denotes symbol repetition.

Besides optimal successor and generation algorithms, cool-lex order also has an optimal ranking algorithm. This result follows from the fact that cool-lex order is very similar to the *co-lex order* of $\mathbb{B}(n, t)$, which is lexicographic order with the order of the bits in each string reversed, and from where cool-lex order gets its name. For example, the two orders appear below for $n = 9$ and $t = 4$.

co-lex order for $\mathbb{B}(9, 4)$

cool-lex order for $\mathbb{B}(9, 4)$
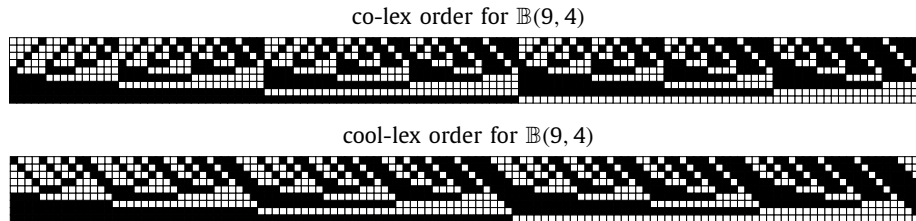
Observe that the bottom rows are identical, except that cool-lex's row is shifted by one position. More specifically, in cool-lex order the column ⬚⬚⬚⬚■■■■■ appears last instead of first. Similarly, in cool-lex order ⬚⬚⬚■■■■⬚■ is the last column with suffix ⬚■■ instead of the first. In general this $\square^* \blacksquare^*$ rearrangement happens for every suffix that is empty or begins with □, and this is the only difference between co-lex and cool-lex for $\mathbb{B}(n, t)$. Co-lex has an optimal ranking algorithm for $\mathbb{B}(n, t)$ using O(t) arithmetic operations on integers as large as $m = \binom{n}{t}$, and this same complexity can be obtained by cool-lex order.

Ruskey and Williams found a similar successor rule for balanced parentheses [20]. Since every (k-ary) Dyck word has 1 as its first symbol, in this context the ith *prefix-shift* moves bits at indices $2, 3, \ldots, i$ once to the right (circularly).

CoolCat successor for $\mathbb{D}_2(t)$**:**
Let i be the index of the first 01 substring's 1. If the $(i + 1)$st prefix-shift is valid then apply it, otherwise apply the ith prefix-shift.
(If there is no 01 substring, then perform the nth prefix-shift.)

For example, $i = 5$ in $11001010 \in \mathbb{D}_2(4)$, so the successor rule first attempts the 6th prefix-shift. However, this prefix-shift is invalid since the result is $10100110 \notin \mathbb{D}_2(4)$. Therefore, the successor is obtained from the 5th prefix-shift. Thus, $1\underline{1001}010$ is succeeded by $1\underline{1100}010$. The successor rule again complements either two or four bits, and it creates a cyclic order of $\mathbb{D}_2(t)$ whose last string is $1^t 0^{n-t}$, by convention. It also leads to an optimal ranking algorithm that uses O(t) arithmetic operations (on integers as large as C_t), as well as an optimal loopless generation algorithm that uses two additional index variables, two if-statements and one else-statement. In addition, the order also provides a Gray code and loopless generation algorithm for binary trees. This marked the first 'simultaneous' Gray code and loopless generation algorithm for balanced parentheses and binary trees. Due to the connections with cool-lex order and the Catalan numbers, the resulting order was named 'CoolCat' order.

Cool-lex and CoolCat orders motivate the following question: What other sets of strings have a similar successor algorithm? When considering this question, it is important to note that CoolCat order for $\mathbb{D}_2(t)$ is a suborder of cool-lex order for $\mathbb{B}(2t, t)$. In other words, the CoolCat order of $\mathbb{D}_2(t)$ can obtained by removing the strings that are not balanced parentheses from the cool-lex order of $\mathbb{B}(2t, t)$. For example, the cool-lex order of $\mathbb{B}(8, 4)$ appears below with ✓ beneath each string that is in $\mathbb{D}_2(4)$, and the resulting CoolCat order for $\mathbb{D}_2(4)$. In general, the suborder property can be verified by using the recursive definition (2) found in Section 4.

cool-lex for $\mathbb{B}(8, 4)$ CoolCat for $\mathbb{D}_2(4)$
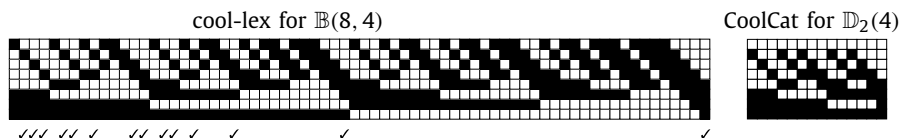
For this reason, the CoolCat order $\mathbb{D}_2(t)$ can instead be called the cool-lex order for $\mathbb{D}_2(t)$. Investigation into other cool-lex suborders resulted in a major generalization by Ruskey, Sawada, and Williams [18]. Their result proves that similar successor rules exist for any subset of $\mathbb{B}(n, t)$ that is a "bubble language" (see Section 2.1). Bubble languages represent a wide variety of combinatorial objects including proper interval graphs, feasible solutions to knapsack problems, binary necklaces (i.e., rotatable binary strings), binary neckties (i.e., reversible binary strings), and k-ary Dyck words. Although the general result by Ruskey, Sawada, and Williams provides a successor rule for a wide variety of combinatorial objects, it does not 'optimize' the rule for any specific object, nor does it address loopless generation or efficient ranking of the resulting order.

1.4. New results

In this article we utilize the general theory on cool-lex order and bubble language to k-ary Dyck words to obtain the following results:

- An O(n)-time successor algorithm.
- A loopless generation algorithm that uses two additional index variables, two if-statements and one else-statement.
- A Gray code and loopless generation algorithm for k-ary trees.
- A ranking algorithm using O(n) arithmetic operations once a table of generalized k-ary Catalan numbers is precomputed.

These results mark the first 'simultaneous' Gray codes and loopless algorithms for k-ary Dyck words and k-ary trees, and the simplest known loopless algorithm for generating k-ary Dyck words. Our article is also interesting for how it uses the general theory on cool-lex order and bubble languages. Instead of conventional cool-lex order, we instead generate k-ary Dyck words and the corresponding k-ary trees in a complemented and revered version of cool-lex order. In other words, we (implicitly) generate $1/k$-ary Dyck words in cool-lex order, and then complement and reverse these strings to obtain our order of k-ary Dyck words. This crucial difference is illustrated below for $k = 3$ and $t = 4$.

cool-lex for $\mathbb{D}_3(4)$　　　cool-lex for $\mathtt{d}_3(4)$　　　complemented and reversed
　　　　　　　　　　　　　　　　　　　　　　　　　　cool-lex for $\mathbb{D}_3(4)$
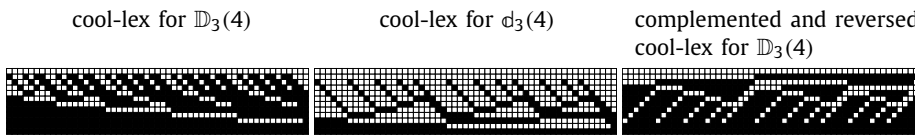
When comparing the two orders of k-ary Dyck words above, observe that the modified order's Gray code is based on suffix-shifts instead of prefix-shifts. A preliminary version of this article by Durocher, Li, Mondal, and Williams [5] used the unaltered version of cool-lex order and obtained weaker results. In particular, the preliminary loopless algorithm for $\mathbb{D}_k(t)$ required an additional array of t index variables and two more if-statements. It also did not give a 'simultaneous' Gray code or loopless generation algorithm for k-ary trees, as explained by Fig. 1(a).

Section 2 focuses on successor algorithms for generating strings in cool-lex order, and Section 3 translates these results into generation algorithms for $1/k$-ary Dyck words, k-ary Dyck words, and k-ary trees. Section 4 provides our results on ranking and unranking. Section 5 concludes with final remarks.

2. Successor algorithms

In this section we describe the general cool-lex successor algorithm for bubble languages in Section 2.1. Then we specialize this result for the special cases of k-ary and $1/k$-ary Dyck words in Section 2.2. To state our successor algorithms precisely, we present them in terms of a successor table. In a *successor table*, each string is matched by one row of the table, and this row provides both the successor and operations that create it. To illustrate this concept, we restate the two successor algorithms from Section 1, starting with Table 0 for the cool-lex successor of $\mathbb{B}(n, t)$.

For example, the string $110001010 \in \mathbb{B}(9, 4)$, matches row (0c) as $1^i 0^j 10\gamma$ for $i = 2$, $j = 3$, and $\gamma = 10$. Therefore, the successor is $01^i 0^j 1\gamma = 011000110$. As noted by the table, the successor can be created by shifting the bit from position $i + j + 2 = 7$ to position 1, or equivalently by the swapping the bits at positions $(1, i + 1) = (1, 3)$ and $(i + j + 1, i + j + 2) = (6, 7)$. Observe (0d) and (0e) handle the special cases when there is no 010 or 011 substring. In particular, the successor of $1^{t-1} 0^{n-t} 1$ is $1^t 0^{n-t}$ by (0e), and the successor of $1^t 0^{n-t}$ is $01^t 0^{n-t-1}$ by (0d).

Table 1 provides a successor table for the CoolCat successor of $\mathbb{D}_2(t)$. In all of our successor tables, any prefix written of the form $1^i 0^j 1$ is assumed to have $j > 0$ as noted by [†]. The rows in Tables 0 and 1 correspond to the general cool-lex successor table in Section 2.1, and this explains why row (0b) and (1e) are left blank. Before providing the table, we first formally define swaps and shifts to avoid potential confusion.

Table 0
Cool-lex successor table for $\mathbb{B}(n, t)$ from [21].

	String[†]	Successor	Shift	Swap(s)
(0a)	$1^i 0^j 11\gamma$	$1^{i+1} 0^j 1\gamma$	$(i + j + 1, 2)$	$(i + 1, i + j + 1)$
(0c)	$1^i 0^j 10\gamma$	$01^i 0^j 1\gamma$	$(i + j + 2, 1)$	$(1, i + 1)$ $(i + j + 1, i + j + 2)$
(0d)	$1^t 0^{n-t}$	$01^t 0^{n-t-1}$	$(n, 1)$	$(1, t + 1)$
(0e)	$1^{t-1} 0^{n-t} 1$	$1^t 0^{n-t}$	$(n, 1)$	(t, n)

[†]　$j > 0$. See figure on page 292.

Table 1
CoolCat successor table for $\mathbb{D}_2(t)$ from [20].

	String[†]	Successor	Shift	Swap(s)
(1a)	$1^i0^j11\gamma$	$1^{i+1}0^j1$	$(i+j+1,2)$	$(i+1,i+j+1)$
(1b)	$1^i0^j10\gamma$ where $i=j$	$1^{i+1}0^{j+1}\gamma$	$(i+j+1,2)$	$(i+1,i+j+1)$
(1c)	$1^i0^j10\gamma$ where $i>j$	$101^{i-1}0^j1\gamma$	$(i+j+2,2)$	$(2,i+1)$ $(i+j+1,i+j+2)$
(1d)	1^t0^t	$101^{t-1}0^{t-1}$	$(n,2)$	$(2,t+1)$

[†] $j>0$. See figure on page 292.

Suppose $B=B_1B_2\cdots B_n$ is a binary string of length n and $1\leqslant i\leqslant j\leqslant n$. The *swap* and *shift* operations are formally defined as follows:

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

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

- *adjacent*: $\mathrm{swap}(i,i+1)$,
- *two-close*: $\mathrm{swap}(i,i+1)$, or $\mathrm{swap}(i,i+2)$ when $B_{i+1}=0$, and
- *homogeneous*: $\mathrm{swap}(i,j)$ where $B_i=B_{i+1}=\cdots=B_{j-1}$.

A *prefix-shift* is an operation of the form $\mathrm{shift}(j,1)$, although in the context of Dyck words we consider $\mathrm{shift}(j,2)$ to be a prefix-shift since the first bit must be 1. Similarly, we consider $\mathrm{shift}(j,k)$ to be a prefix-shift $1/k$-ary Dyck words since the first $k-1$ bits must be 1. A *homogeneous* shift is $\mathrm{shift}(B,j,i)$ when $B_i=B_{i+1}=\cdots=B_{j-1}$.

2.1. Bubble languages

A *binary bubble language* is a set of binary strings $\mathbb{L}$ with the following property: If $B\in\mathbb{L}$ where $B=1^i0^j1\gamma$ for some suffix γ and $j>0$, then $1^i0^{j-1}10\in\mathbb{L}$. In other words, the first 01 of any string in the set can be replaced by 10 to give another string in the set. The following lemma proves that k-ary and $1/k$-ary Dyck words of length kt are binary bubble languages.

Lemma 1. *The k-ary Dyck words in $\mathbb{D}_k(t)$ are a binary bubble language, as are the $1/k$-ary Dyck words in $\mathbb{d}_k(t)$.*

Proof. Let $\mathbb{L}=\mathbb{D}_k(t)$ or $\mathbb{L}=\mathbb{d}_k(t)$. The set $\mathbb{L}$ has a stronger property: If $\alpha01\gamma\in\mathbb{L}$, then $\alpha10\gamma\in\mathbb{L}$. This is because replacing any 01 by 10 does not decrease the number of 1s in any prefix. $\square$

We only consider binary bubble languages that are subsets of some $\mathbb{B}(n,t)$; for brevity we refer to these languages simply as *bubble languages*. The main result of [18] is a general successor algorithm that cyclically generates any bubble language. As with the cool-lex successor for $\mathbb{B}(n,t)$ and the CoolCat successor for $\mathbb{D}_2(t)$, the general successor can be expressed as a single shift or as a pair of swaps. More specifically, the general successor requires at most one homogeneous-transposition and at most one adjacent-transposition. However, the shift is not necessary a prefix-shift.

Theorem 1. *(See [18].) The strings in any bubble language are generated in cool-lex order by the shift (or equivalent swap(s)) in Table 2.*

Table 2
Cool-lex successor table for a bubble language $\mathbb{L}$ from [18].

	String[†]	Successor[‡]	Shift	Swap(s)
(2a)	$1^i0^j11\gamma$	$1^{i+1}0^j1\gamma$	$(i+j+1,1)$	$(i+1,i+j+1)$
(2b)	$1^i0^j10\gamma$ where $1^i0^{j+1}1\gamma\notin\mathbb{L}$	$1^{i+1}0^{j+1}\gamma$	$(i+j+1,1)$	$(i+1,i+j+1)$
(2c)	$1^i0^j10\gamma$ where $1^i0^{j+1}1\gamma\in\mathbb{L}$	$1^h01^{i-h}0^j1\gamma$	$(i+j+2,h+1)$	$(h+1,i+1)$ $(i+j+1,i+j+2)$
(2d)	1^t0^{n-t}	$1^g01^{t-g}0^{n-t-1}$	$(n,g+1)$	$(g+1,t+1)$
(2e)	$1^{t-1}0^{n-t}1$	1^t0^{n-t}	$(n,1)$	(t,n)

[†] $j>0$.
[‡] 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 $1^g01^{t-g}0^{n-t-1}\in\mathbb{L}$.

To familiarize ourselves with this general table, we consider (2c) in detail.

	String	Successor[‡]	Shift	Swap(s)
(2c)	$1^i 0^j 10\gamma$ where $1^i 0^{j+1} 1\gamma \in \mathbb{L}$	$1^h 0 1^{i-h} 0^j 1\gamma$	$(i+j+2, h+1)$	$(h+1, i+1)$
				$(i+j+1, i+j+2)$

[‡] h is the minimum value such that $1^h 0 1^{i-h} 0^j 1\gamma \in \mathbb{L}$.

If a string matches this row, then the successor is obtained by shifting the last bit of a $1^i 0^j 10$ prefix. In particular, the definition of h implies that the shift$(i+j+2, h+1)$ operation moves this bit as far to the left as possible while ensuring the result is in $\mathbb{L}$. In this context, the condition $1^i 0^{j+1} 1\gamma \in \mathbb{L}$ ensures that this bit can be shifted at least one position. Now consider how this general rule translates into specific rules for $\mathbb{B}(n, t)$ and $\mathbb{D}_2(t)$. When $\mathbb{L} = \mathbb{B}(n, t)$, the last bit of a $1^i 0^j 10$ prefix can always be shifted all the way to the first position. Thus, $h = 0$ in this case and so shift$(i+j+2, h+1) =$ shift$(i+j+2, 1)$ as illustrated below.

	String	Successor	Shift	Swap(s)
(0c)	$1^i 0^j 10\gamma$	$01^i 0^j 1\gamma$	$(i+j+2, 1)$	$(1, i+1)$ $(i+j+1, i+j+2)$

On the other hand when $\mathbb{L} = \mathbb{D}_2(t)$, the last bit of a $1^i 0^j 10$ prefix cannot be shifted to the left at all when $i = j$. Furthermore, when $i > j$, this bit can only be shifted as far as the second position. Thus, $h = 1$ in this case and so shift$(i+j+2, h+1) =$ shift$(i+j+2, 2)$ as illustrated below, along with the condition $i > j$.

	String	Successor	Shift	Swap(s)
(1c)	$1^i 0^j 10\gamma$ for $i > j$	$101^{i-1} 0^j 1\gamma$	$(i+j+2, 2)$	$(2, i+1)$ $(i+j+1, i+j+2)$

In general, there is no guarantee that Theorem 1 will lead to an efficient successor algorithm. For example, in the case of binary necklaces, the task of computing h is non-trivial (see Sawada and Williams [22] a CAT generation algorithm using this successor algorithm). In the next subsection we specialize and optimize this general successor algorithm to the special cases of $\mathbb{L} = \mathbb{D}_k(t)$ and $\mathbb{L} = \mathfrak{d}_k(t)$. This "specialization process" is somewhat tedious and technical, but the resulting successor algorithms are very simple.

2.2. Successor algorithms for Dyck words

In Theorem 2 we prove that Tables 3 and 4 provide successor algorithms for k-ary Dyck words and $1/k$-ary Dyck words, respectively. For example, the successor algorithm in Table 3 gives the following order for $\mathbb{D}_3(3)$

101100000, 110100000, 101010000, 100110000, 110010000, 101001000,

100101000, 110001000, 101000100, 100100100, 110000100, 111000000.

In particular, the above order is the result of applying (3a), (3c), (3c), (3a), (3c), (3c), (3b), (3c), (3c), (3b), (3b), and finally (3d) to make the order cyclic. Similarly, the successor algorithm in Table 4 gives the following order for $\mathfrak{d}_3(3)$

110111100, 111011100, 111101100, 111110100, 110111010, 111011010,

111101010, 110110110, 111010110, 111100110, 111110010, 111111000.

In particular, the above order is the result of applying (4a), (4a), (4a), (4c), (4a), (4a), (4c), (4a), (4b), (4a), (4b), and finally (4d) to make the order cyclic.

Theorem 2. *The k-ary and $1/k$-ary Dyck words of length kt are generated in cool-lex order by the successor algorithms in Tables* 3 *and* 4, *respectively. Moreover, the successor algorithms take* O(n) *time and use a single prefix-shift.*

Table 3
Cool-lex successor table for the k-ary Dyck words in $\mathbb{D}_k(t)$ of length $n = kt$.

	String[†]	Successor	Shift	Swap(s)
(3a)	$1^i 0^j 11\gamma$	$1^{i+1} 0^j 1\gamma$	$(i+j+1, 2)$	$(i+1, i+j+1)$
(3b)	$1^i 0^j 10\gamma$ where $(k-1)i = j$	$1^{i+1} 0^{j+1}\gamma$	$(i+j+1, 2)$	$(i+1, i+j+1)$
(3c)	$1^i 0^j 10\gamma$ where $(k-1)i > j$	$101^{i-1} 0^j 1\gamma$	$(i+j+2, 2)$	$(2, i+1)$
				$(i+j+1, i+j+2)$
(3d)	$1^t 0^{(k-1)t}$	$101^{t-1} 0^{(k-1)t-1}$	$(n, 2)$	$(2, t+1)$

[†] $j > 0$.

Table 4

Cool-lex successor table for the $1/k$-ary Dyck words in $\mathbb{d}_k(t)$ of length $n = kt$.

	String[†]	Successor	Shift	Swap(s)
(4a)	$1^i0^j11\gamma$	$1^{i+1}0^j1\gamma$	$(i+j+1,k)$	$(i+1,i+j+1)$
(4b)	$1^i0^j10\gamma$ where $i=(k-1)(j+1)-1$	$1^{i+1}0^{j+1}\gamma$	$(i+j+1,k)$	$(i+1,i+j+1)$
(4c)	$1^i0^j10\gamma$ where $i \geqslant (k-1)(j+1)$	$1^{k-1}01^{i-k+1}0^j1\gamma$	$(i+j+2,k)$	$(k,i+1)$
				$(i+j+1,i+j+2)$
(4d)	$1^{(k-1)t}0^t$	$1^{k-1}01^{(k-1)(t-1)}0^{t-1}$	(n,k)	$(k,(k-1)t+1)$

† $j > 0$.

Proof. The two cases have very similar proofs, and the proof for k-ary Dyck words appeared in [5]; we prove the result only for $1/k$-ary Dyck words.

$\mathbb{L} = \mathbb{d}_k(t)$ is a bubble language by Lemma 1 (and [18]); Theorem 1 implies that it is generated in cool-lex order by the successor algorithm in Table 2. Now compare each rule in Table 2 to its specialization for $\mathbb{L} = \mathbb{d}_k(t)$ in Table 4.

	String	Successor	Shift	Swap(s)
(2a)	$1^i0^j11\gamma$	$1^{i+1}0^j1\gamma$	$(i+j+1,1)$	$(i+1,i+j+1)$
(4a)	$1^i0^j11\gamma$	$1^{i+1}0^j1\gamma$	$(i+j+1,k)$	$(i+1,i+j+1)$

If a $1/k$-ary Dyck word has prefix 1^i0^j11 and $j > 0$, then it must be that $i \geqslant k-1$. Therefore, shift$(i+j+1,k)$ is equivalent to shift$(i+j+1,1)$ in this context. Therefore, (4a) specializes (2a).

	String	Successor	Shift	Swap(s)
(2b)	$1^i0^j10\gamma$ where $1^i0^{j+1}1\gamma \notin \mathbb{L}$	$1^{i+1}0^{j+1}\gamma$	$(i+j+1,1)$	$(i+1,i+j+1)$
(4b)	$1^i0^j10\gamma$ where $i=(k-1)(j+1)-1$	$1^{i+1}0^{j+1}\gamma$	$(i+j+1,k)$	$(i+1,i+j+1)$

Suppose that $1^i0^j10\gamma$ is a $1/k$-ary Dyck word. Observe that $1^i0^{j+1}1\gamma$ is not a $1/k$-ary Dyck word if and only if $i < (k-1)(j+1)$. On the other hand, $1^i0^j10\gamma$ is a $1/k$-ary Dyck word. Thus, $i+1 \geqslant (k-1)(j+1)$ which implies that $i \geqslant (k-1)(j+1)-1$. Therefore, the condition in (4b) specializes the condition in (2b). Next observe that $i \geqslant k-1$ since $1/k$-ary Dyck words begin with 1^{k-1}. Therefore, shift$(i+j+1,k)$ is equivalent to shift$(i+j+1,1)$ in this context. Therefore, (4b) specializes (2b).

	String	Successor[‡]	Shift	Swap(s)
(2c)	$1^i0^j10\gamma$ where $1^i0^{j+1}1\gamma \in \mathbb{L}$	$1^h01^{i-h}0^j1\gamma$	$(i+j+2,h+1)$	$(h+1,i+1)$
				$(i+j+1,i+j+2)$
(4c)	$1^i0^j10\gamma$ where $i \geqslant (k-1)(j+1)$	$1^{k-1}01^{i-k+1}0^j1\gamma$	$(i+j+2,k)$	$(k,i+1)$
				$(i+j+1,i+j+2)$

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

Suppose $1^i0^j10\gamma$ is a $1/k$-ary Dyck word. Observe $1^i0^{j+1}1\gamma$ is a $1/k$-ary Dyck word if and only if $i \geqslant (k-1)(j+1)$. Therefore, the condition in (4c) specializes the condition in (2c). Given $i \geqslant (k-1)(j+1)$, next observe that $h = k-1$ since $1/k$-ary Dyck words begin with 1^{k-1}. Therefore, shift$(i+j+2,k)$ is equivalent to shift$(i+j+2,h+1)$ in this context, and swap$(k,i+1)$ is equivalent to swap$(h+1,i+1)$ in this context. Therefore, (4c) specializes (2c).

	String	Successor[‡]	Shift	Swap(s)
(2d)	1^t0^{n-t}	$1^g01^{t-g}0^{n-t-1}$	$(n,g+1)$	$(g+1,t+1)$
(4d)	$1^{(k-1)t}0^t$	$1^{k-1}01^{(k-1)(t-1)}0^{t-1}$	(n,k)	$(k,(k-1)t+1)$

‡ g is the minimum value such that $1^g01^{t-g}0^{n-t-1} \in \mathbb{L}$.

Since $\mathbb{L} = \mathbb{d}_k(t)$ the value of n is kt. Furthermore, if g is the minimum value such that $1^g01^{t-g}0^{n-t-1} \in \mathbb{d}_k(t)$, then $g = k-1$. Therefore, shift(n,k) is equivalent to shift$(n,g+1)$ in this context, and swap$(k,(k-1)t+1)$ is equivalent to swap$(g+1,t+1)$ in this context. Furthermore, the stated strings and successors are also equivalent. Therefore, (4d) specializes (2d).

	String	Successor	Shift	Swap(s)
(2e)	$1^{t-1}0^{n-t}1$	1^t0^{n-t}	$(n,1)$	(t,n)

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

Therefore, the successor algorithm in Table 4 specializes that successor algorithm in Table 2 when $\mathbb{L} = \mathbb{d}_k(t)$. The 'Shift' column implies that the algorithm applies a prefix-shift, and the 'Swap' column provides O(n)-time complexity. □

3. Generation algorithms

In this section we use the successor algorithms from Section 2 to develop loopless algorithms for generating k-ary and $1/k$-ary Dyck words in an array. Recall from Section 1.2 that algorithms for generating these two objects are interchangeable. For this reason we ask the following question: Which of the two successor algorithms in Section 2 will lead to a more efficient algorithm?

To answer this question, we begin with a simple observation. To apply the successor algorithm in Tables 3 and 4 to a given string, we need to first identify its prefix of the form $1^i 0^j 1$. In the case of a loopless algorithm, we will need to continually update these i and j values in worst-case O(1) time for each successive string. This becomes an issue when we apply (3c) or (4c). For example, consider the following two strings and their successors for $k = 6$:

$$\text{string:} \quad 11000000000010\gamma \in \mathbb{D}_6(t) \qquad \text{string:} \quad 11111111111110010\gamma \in \mathbb{d}_6(t)$$
$$\text{successor:} \quad 11100000000000\gamma \in \mathbb{D}_6(t) \qquad \text{successor:} \quad 11111111111111000\gamma \in \mathbb{d}_6(t)$$

In both cases the successor's prefix of the form $1^i 0^j 1$ extends into γ. Therefore, before we can generate the successor's successor, we need to determine how many leading 0s there are in γ. We consider this problem for the 6-ary Dyck word example (above left) and $1/6$-ary Dyck word example (above right).

- In the 6-ary Dyck word, the first three 1s can 'support' at most $5 \cdot 3 = 15$ copies of 0. Thus, γ contains at most four leading 0s.
- In the $1/6$-ary Dyck word, the first fifteen 1s can 'support' at most $\frac{15}{5} = 3$ copies of 0. Thus, γ contains no leading 0s.

To resolve the uncertainty in the 6-ary Dyck word, we either need to scan the first four bits of γ or use additional memory to store this information. Both of these options are explored in [5]. The scanning option leads to a "fixed-parameter loopless algorithm" that takes O(k)-time per iteration, and the storage option leads to a loopless algorithm requiring O(t) additional index variables. On the other hand, neither of these expenses will be required for $1/k$-ary Dyck words.

3.1. Generating $1/k$-ary Dyck words

Our algorithm for generating the $1/k$-ary Dyck words of length $n = kt$ is named $\text{Cool}\frac{1}{K}(k, t)$ and it appears in Algorithms 1. It stores the current $1/k$-ary Dyck word in an array of length n (with 1-based indices) and uses two additional variables x and y. The algorithm follows the swap rules of successor table for $\mathbb{d}_k(t)$ found in Table 4. We restate these swaps in Table 5 with one important change. In (4c) the stated swaps are $(k, i + 1)$ and $(i + j + 1, i + j + 2)$, while in (5c) they are $(i + 1, i + j + 1)$ and $(k, i + j + 2)$. These pairs of swaps are equivalent since the binary string to which they are applied must contain 1s in position k and $i + j + 1$, and 0s in position $i + 1$ and $i + j + 2$. The reason for adjusting these swap indices is the following: $\text{swap}(i + 1, i + j + 1)$ is now performed when creating the successor of every string (except $1^t 0^{(k-1)t}$) and so it can be applied at the start of each iteration.

As is customary, $\text{Cool}\frac{1}{K}$ calls a 'visit' statement every time the next string has been created. It is slightly more convenient for us to visit the string $1^{(k-1)t} 0^t \in \mathbb{d}_k(t)$ first instead of last. The following theorem presents a formal proof of correctness for the $\text{Cool}\frac{1}{K}$ algorithm.

Theorem 3. $\text{Cool}\frac{1}{K}(k, t)$ is a loopless algorithm that uses two additional variables and cyclically generates the $1/k$-ary Dyck words of length kt in cool-lex order.

Proof. The algorithm initializes the B array to $1^{(k-1)t} 0^t$ and then visits this string. The first iteration of the while loop completes by visiting the string $B = 1^{k-1} 0 1 1^{(k-1)(t-1)} 0^{t-1}$, which is the correct successor by (5d), and with values $x = k + 1$

Table 5
Cool-lex successor table for the $1/k$-ary Dyck words in $\mathbb{d}_k(t)$ of length $n = kt$ with modified swaps. This table is the basis of the $\text{Cool}\frac{1}{K}$ algorithm.

	String[†]	Successor	Swap(s)
(5a)	$1^i 0^j 11\gamma$	$1^{i+1} 0^j 1\gamma$	$(i + 1, i + j + 1)$
(5b)	$1^i 0^j 10\gamma$ where $i = (k-1)(j+1) - 1$	$1^{i+1} 0^{j+1}\gamma$	$(i + 1, i + j + 1)$
(5c)	$1^i 0^j 10\gamma$ where $i \geqslant (k-1)(j+1)$	$1^{k-1} 0 1 1^{i-k+1} 0^j 1\gamma$	$(i + 1, i + j + 1)$
			$(k, i + j + 2)$
(5d)	$1^{(k-1)t} 0^t$	$1^{k-1} 0 1 1^{(k-1)(t-1)} 0^{t-1}$	$(k, (k-1)t + 1)$

[†] $j > 0$.

and $y = k$. Now we state a simple loop invariant that holds at the start of the while loop for each iteration starting with the second:

$$B = 1^{y-1}0^{x-y}1z\gamma \in \mathbb{d}_k(t) \quad \text{for a bit } z \in \{0, 1\}, \text{ a suffix } \gamma, \text{ and } x - y > 0. \tag{1}$$

In other words, each loop begins with a $1/k$-ary Dyck word in which $B[y]$ stores the first 0, and $B[x]$ stores the first 1 that occurs after a 0, and z is the value of the bit following the first 01. Given this loop invariant and its correctness at the first iteration, we prove the theorem by induction on the number of iterations. In this inductive proof we also assume that $\gamma \neq \epsilon$ (the empty string) for reasons discussed at the end of the proof.

Suppose $\alpha = 1^i0^j1z\gamma \in \mathbb{d}_k(t)$ is visited at the end of the pth iteration, where $z \in \{0, 1\}$ is a single bit and $j > 0$. By induction, $B = 1^{y-1}0^{x-y}1z\gamma$ at the start of the $(p+1)$st iteration. That is, $y = i + 1$ and $x = i + j + 1$. The result of lines 7–10 is that x and y are incremented so that $y = i + 2$ and $x = i + j + 2$ and the array holds $B = 1^{y-1}0^{x-y}z\gamma$. Observe $z = B[x]$. Now consider the possible paths through the algorithm.

- If $B[x] = 1$, then $\alpha = 1^i0^j11\gamma$. In this case the if-statement on line 7 is not entered and the iteration ends by visiting

    $$B = 1^{y-1}0^{x-y}1\gamma = 1^{i+1}0^j1\gamma.$$

    This is the correct successor to α by (5a). Furthermore, the loop-invariant (1) holds by appropriately defining new values for z and γ.
- If $B[x] = 0$, then $\alpha = 1^i0^j10\gamma$ and the array holds $B = 1^{y-1}0^{x-y}0\gamma = 1^{i+1}0^j0\gamma$ since $y = i + 2$ and $x = i + j + 2$. In this case the if-statement on line 11 is entered.
  - ▷ If $(k-1)(x-y+1) = y - 1$, then $(k-1)(j+1) = i + 1$. Therefore, the prefix 1^i0^j10 in α is 'tight' in the sense that its next symbol cannot be 0. Thus, $\alpha = 1^i0^j101\gamma'$ where $\gamma = 1\gamma'$. Given the $(k-1)(x-y+1) = y - 1$ condition, the if-statement on line 12 is entered. The result is that x is incremented to $x = i + j + 3$ and the iteration ends by visiting

      $$B = 1^{y-1}0^{x-y}\gamma = 1^{y-1}0^{x-y}1\gamma' = 1^{i+1}0^{j+1}1\gamma'.$$

      This is the correct successor to α by (5b) and the fact that $i = (k-1)(j+1) - 1$. Furthermore, the loop-invariant (1) holds by appropriately defining new values for z and γ.
  - ▷ If $(k-1)(x-y+1) \neq y - 1$, then the else-statement on line 14 is entered. The result of lines 15–18 is that the iteration ends by visiting

      $$B = 1^{k-1}01^{i-k+1}0^j1\gamma = 1^{y-1}0^{x-y}1^{i-k+1}0^j\gamma = 1^{y-1}0^{x-y}\gamma',$$

      with $x = k + 1$ and $y = k$ and $\gamma' = 1^{i-k+1}0^j\gamma$. This is the correct successor to α by (5c), Furthermore, we can prove that the loop-invariant (1) holds so long as the first symbol of γ is 1. In other words, we need to prove that $i - k + 1 > 0$. This follows from the fact that $(k-1)(x-y+1) \neq y - 1$ implies that $i \geqslant (k-1)(j+1)$ (see the discussion from the previous case) and so $j > 0$ implies that $i - k \geqslant k - 2$, which is sufficient since $k \geqslant 2$.

Therefore, by induction each iteration of the algorithm correctly performs the successor algorithm, so long as $\gamma \neq \epsilon$ by our earlier assumption. Observe that $\gamma = \epsilon$ is only possible for the string $\alpha = 1^{(k-1)t-1}0^{t-1}10 \in \mathbb{d}_k(t)$. Therefore, the algorithm will eventually visit this string stored in $B = 1^{y-1}0^{x-y}1z\gamma$ where $z = 0$ and $\gamma = \epsilon$. In this case the algorithm terminates by line 6 since $x = n - 1$. To see why this is the correct behavior, observe that (5b) transforms α into $1^{(k-1)t}0^t$ which is the string we started the algorithm by visiting. Therefore every string in $\mathbb{d}_k(t)$ is visited by Theorem 2 and the restatement of Table 4 in Table 5.   □

3.2. Generating k-ary Dyck words

To generate k-ary Dyck words in $\mathbb{D}_k(t)$, we simply modify the Cool$\frac{1}{k}$ for generating $\mathbb{d}_k(t)$. More specifically, we initialize the array B to $1^t0^{(k-1)t}$ instead of $1^{(k-1)t}0^t$ (see line 1), then we reverse indices by replacing each array access of the form $B[i]$ by $B[n-i+1]$, and finally we complement each value that we get from the array and set in the array (see lines 7, 8, 11, 15, and 16). The resulting CoolK algorithm appears on the right side of Algorithm 1.

Theorem 4. *CoolK(k, t) is a loopless algorithm that uses two additional index variables, and cyclically generates k-ary Dyck words of length kt in complemented and reversed cool-lex order.*

Proof. This follows from Theorem 3 and Remark 1.   □

Although the correctness of CoolK is established by Theorem 4, its precise successor algorithm is somewhat obfuscated by string transformations being applied. For this reason, we translate the successor algorithm from Table 5 into Table 6. We will use Table 6 when generating k-ary trees.

Algorithm 1 $\text{Cool}\frac{1}{K}(k,t)$ and $\text{CoolK}(k,t)$ generate $1/k$-ary and k-ary Dyck words of length kt in cool-lex order, respectively. The algorithms are both loopless and use two additional index variables.

Procedure $\text{Cool}\frac{1}{K}(k,t)$	**Procedure** $\text{CoolK}(k,t)$
1: $B \leftarrow \text{array}(1^{(k-1)t}0^t)$	1: $B \leftarrow \text{array}(1^t0^{(k-1)t})$
2: $n \leftarrow k \cdot t$	2: $n \leftarrow k \cdot t$
3: $x \leftarrow (k-1) \cdot t$	3: $x \leftarrow (k-1) \cdot t$
4: $y \leftarrow (k-1) \cdot t$	4: $y \leftarrow (k-1) \cdot t$
5: visit()	5: visit()
6: **while** $x < n-1$	6: **while** $x < n-1$
7: $B[x] \leftarrow 0$	7: $B[n-x+1] \leftarrow 1$
8: $B[y] \leftarrow 1$	8: $B[n-y+1] \leftarrow 0$
9: $x \leftarrow x+1$	9: $x \leftarrow x+1$
10: $y \leftarrow y+1$	10: $y \leftarrow y+1$
11: **if** $B[x] = 0$ **then**	11: **if** $B[n-x+1] = 1$ **then**
12: **if** $(k-1)(x-y+1) = y-1$ **then**	12: **if** $(k-1)(x-y+1) = y-1$ **then**
13: $x \leftarrow x+1$	13: $x \leftarrow x+1$
14: **else**	14: **else**
15: $B[x] \leftarrow 1$	15: $B[n-x+1] \leftarrow 0$
16: $B[k] \leftarrow 0$	16: $B[n-k+1] \leftarrow 1$
17: $x \leftarrow k+1$	17: $x \leftarrow k+1$
18: $y \leftarrow k$	18: $y \leftarrow k$
19: **end**	19: **end**
20: **end**	20: **end**
21: visit()	21: visit()
22: **end**	22: **end**

Table 6

Successor table for the k-ary Dyck words in $\mathbb{D}_k(t)$ of length $n = kt$ in reverse complemented cool-lex order. This table is implicitly used by the $\text{Cool}\frac{1}{K}$ algorithm.

	String[†]	Successor	Swap(s)
(6a)	$\gamma 001^j0^i$	$\gamma 01^j0^{i+1}$	$(n-i, n-i-j)$
(6b)	$\gamma 101^j0^i$ where $i = (k-1)(j+1)-1$	$\gamma 1^{j+1}0^{i+1}$	$(n-i-j, n-i)$
(6c)	$\gamma 101^j0^i$ where $i \geqslant (k-1)(j+1)$	$\gamma 01^j0^{i-k+1}10^{k-1}$	$(n-i-j, n-i)$
			$(n-i-j-1, n-k+1)$
(6d)	$1^t0^{(k-1)t}$	$1^{t-1}0^{(k-1)(t-1)}10^{k-1}$	$(t, n-k+1)$

† $j > 0$.

3.3. Generating k-ary trees

Now we provide a loopless algorithm for generating k-ary trees with t internal nodes. By *k-ary tree* we mean a rooted tree, where each non-leaf node has k children ordered from 1 to k. Visually we present the parent of each node above its children, and the children ordered from left-to-right. Internal nodes and leaves are denoted by ● and □, respectively. We assume that an individual k-ary tree is stored in a typical "computer representation", where each internal node has an array of pointers to its children, as well as a pointer to its parent.

To create our loopless algorithm we first need a Gray code order. As discussed in Section 1.2, k-ary trees are 'sensitive' to changes in the prefix of their corresponding Dyck word. In particular, Fig. 1(a) provides an example where a homogeneous prefix-shift in a k-ary Dyck word results in a non-constant amount of change to the corresponding k-ary tree. However, we will see that the suffix-shifts from the successor algorithm for $\mathbb{D}_k(t)$ in Table 6 does result in a 'simultaneous' Gray code for k-ary trees. More specifically, our successor algorithm moves at most two internal nodes.

3.3.1. Additional memory

Besides the basic representation of a k-ary tree and a pointer to its root, we need $t + 1$ additional pointers and t index variables to implement our loopless algorithm. We now describe this additional memory, with Fig. 2 providing an example for a 5-ary tree. The additional index variables provide the child number of each internal non-root node. In other words, an internal node has the value ℓ associated with it when it is the ℓth child of its parent. We refer to this value as the *label* of the internal node, and note that its value is $\ell \in [k]$.

To describe the additional pointers we need to introduce a small amount of terminology. An edge is a *k-edge* if the child is labeled $\ell = k$. A *k-path* is a maximal path of k-edges that contains at least one edge. This maximality condition ensures that each node is on at most one k-path. Each k-path has two extreme nodes known as its *ends*. We associate an additional pointer with each node as follows: If a node is the end of a k-path, then its *k-pointer* points to the other end of the k-path; otherwise, its k-pointer points to itself. Observe the k-pointer points 'up' the tree for nodes that are labeled k and do not have a child labeled k, and points 'down' for nodes that have a child labeled k but are not labeled k themselves. We maintain one additional pointer a, which is discussed in the following paragraph.
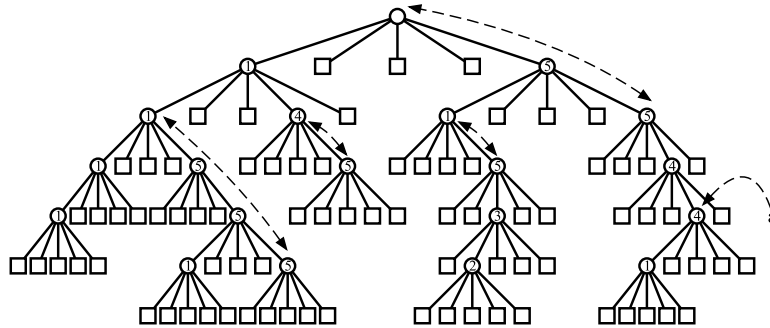
Fig. 2. The above tree illustrates (i) child number 'labels' for each non-root node, (ii) k-pointers between the ends of every k-path, and (iii) the a pointer.

When discussing the algorithm, we refer to three specific nodes: an internal node Ⓐ, a leaf Ⓑ, and an internal node Ⓒ. These nodes can be defined in terms of which bit they represent in the tree's corresponding k-ary Dyck word:

- Ⓐ represents the first 1 of a 01^j0^i suffix, and
- Ⓑ represents the first 0 of a 01^j0^i suffix, and
- Ⓒ represents the first 1 of a 101^j0^i suffix,

where $j > 0$. Observe that Ⓐ and Ⓑ are well-defined for all k-ary trees, except when the corresponding k-ary Dyck word is $1^t0^{(k-1)t}$. On the other hand, Ⓒ is only defined when the corresponding k-ary Dyck word has the specified suffix. In terms of additional memory, we maintain a pointer to Ⓐ, and references to Ⓑ and Ⓒ can be computed from Ⓐ "on-the-fly" in $O(1)$ time when necessary.

Besides these nodes, we will also refer to the "next youngest sibling" of Ⓐ. That is, if Ⓐ has label ℓ, then we will refer to the $(\ell-1)$st child of Ⓐ's parent. Since there is no word for "next youngest sibling" in English, we borrow the Japanese term *otōto* for this concept.[4] Observe that Ⓐ's otōto is always well-defined since Ⓐ has label ℓ for $\ell > 1$. In general, Ⓐ's otōto can be a leaf or an internal node, and in either case it can be easily obtained from Ⓐ in $O(1)$ time. Two small examples of Ⓐ, Ⓑ, Ⓒ, and Ⓐ's otōto appear below.
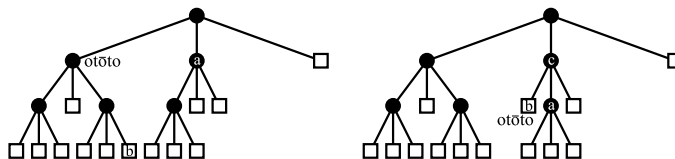
3.3.2. Successor translation: k-ary Dyck words to k-ary trees

Now we translate each row of Table 6 into corresponding $O(1)$-time operations on k-ary trees. We will see that each row is the result of moving Ⓐ and sometimes Ⓒ. We also translate the conditions of each row into $O(1)$-time tests, so that the resulting algorithm can determine which case to apply. In each case, we omit the details on updating every additional variables in $O(1)$ time. For the moment, we assume that Ⓐ is well-defined and then we discuss (6d) as a special case. First we consider (6a) below.

	String	Successor
(6a)	$\gamma 001^j0^i$	$\gamma 01^j0^{i+1}$

This case applies when the corresponding k-ary Dyck word has a suffix of the form 001^j0^i where $j > 0$. This arises in two distinct scenarios:

1. Ⓐ's label is $\ell \geqslant 3$ and Ⓐ's otōto is a leaf, or
2. Ⓐ's otōto is an internal node.

Observe that both scenarios can be tested for in $O(1)$ time. In the first scenario, Ⓑ is simply Ⓐ's otōto. In the second scenario, Ⓑ is the kth child of the otōto's k-pointer. In both scenarios, we simply swap Ⓐ and Ⓑ to change the suffix

---

[4] Otōto literally translates to "younger brother" but we use it as "next youngest sibling".

(i) 100101100001110000000 (left)
100101100011100000000 (right)

(ii) 11000010111000010001110000000 (left)
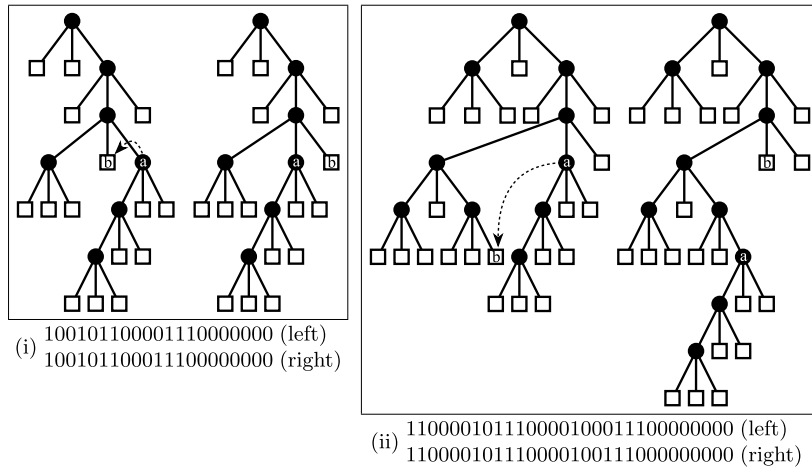11000010111000010011110000000 (right)

Fig. 3. Examples of translating (6a) when a's otōto is (i) a leaf, and (ii) a non-leaf. In both cases the successor is applied to the tree on the left, and the result is on the right.

(i) 100100110000101110000000 (left)
100100110000111100000000 (right)

(ii) 1001100001010100100101110000000 (left)
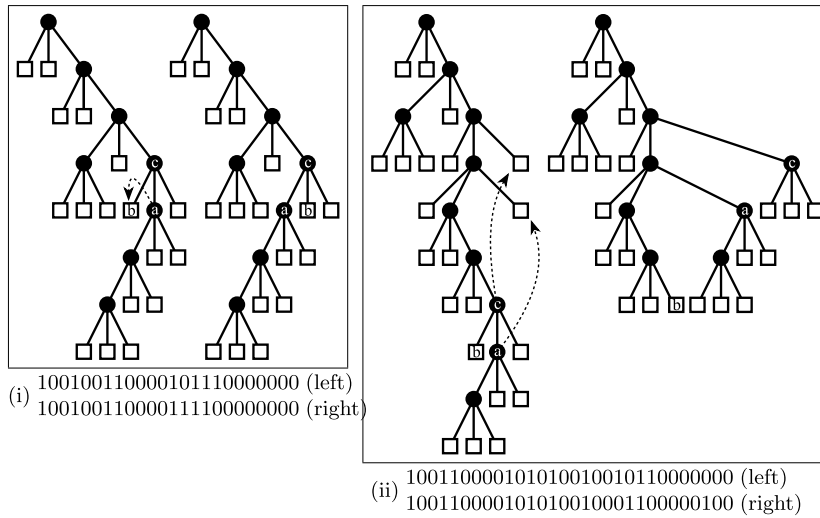1001100001010100100001100000100 (right)

Fig. 4. Examples of (i) translating (6b), and (ii) translating (6c). In both cases the successor is applied to the tree on the left, and the result is on the right.

001^j0^i into $01^j00^i = 01^j0^{i+1}$. This can be done in O(1) time, and the two scenarios are illustrated by Fig. 3(i) and (ii), respectively.

The next two cases are (6b) and (6c). In these cases the corresponding k-ary Dyck word has suffix 101^j0^i and the following two points hold:

- ⓐ's label is $\ell = 2$, and
- ⓐ's otōto is a leaf.

These two points imply that ⓒ is ⓐ's parent.
We consider (6b) below.

	String	Successor
(6b)	$\gamma 101^j0^i$ where $i = (k-1)(j+1) - 1$	$\gamma 1^{j+1}0^{i+1}$

When $i = (k-1)(j+1) - 1$, the root and ⓒ are on the same k-path. The root is one end of this k-path, and the other end of the k-path is either ⓒ (when $k > 2$) or ⓐ (when $k = 2$). Observe that both situations can be tested for in O(1) time. To change the suffix 101^j0^i into $11^j00^i = 1^{j+1}0^{i+1}$ we again simply swap ⓐ and ⓑ. This can be done in O(1) time, as illustrated by Fig. 4(i).

We consider (6c) below.

	String	Successor
(6c)	$\gamma 101^j 0^i$ where $i \geqslant (k-1)(j+1)$	$\gamma 01^j 0^{i-k+1} 10^{k-1}$

When $i \geqslant (k-1)(j+1)$, the root and ⓒ are not on the same k-path. More specifically, the root's k-pointer is not equal to ⓒ (when $k > 2$) and is not equal to ⓐ (when $k = 2$). Observe that both situations can be tested for in O(1) time. In this case we need to move both ⓐ and ⓒ. Fig. 4(ii) provides an illustration. Before describing the movements, it is helpful to point out that the subtree rooted at ⓐ represents the substring $1^i 0^{(k-1)i+1}$ in the corresponding k-ary Dyck word since the final 0 is not omitted. Also, if ⓐ was replaced by a leaf, then the subtree rooted at ⓒ would represent the substring 10^k.

We conceptualize ⓐ's movement in two steps. First we move ⓐ so that it becomes the kth child of ⓒ instead of its 2nd child. This has the effect of replacing the corresponding k-ary Dyck word suffix $101^j 0^i$ by $10^{k-1} 1^j 0^{i-k+2}$. In other words, we have moved ⓐ's substring $1^i 0^{(k-1)i+1}$ to the right $k-2$ positions. To complete the movement of ⓐ we move its substring one more position to the right. We do this by swapping ⓐ with a leaf that is obtained as follows: Take ⓒ's k-pointer to a node which must be labeled i for $i < k$, then travel up one parent, and finally down to the $i+1$st child to obtain the desired leaf. This second swap changes the corresponding k-ary Dyck word suffix from $10^{k-1} 1^j 0^{i-k+2}$ to $10^k 1^j 0^{i-k+1}$. Conceptually we have described this movement of ⓐ in two steps, but the intermediate step is unnecessary in practice. Regardless of the implementation, ⓐ's movement this can be done in O(1) time.

Next we move ⓒ so that it extends the root's k-path. More specifically, we swap ⓒ with the leaf that is the kth child of the node pointed to by the root's k-pointer. This swap has the result of swapping ⓒ's 10^k substring with the last 0 in the corresponding k-ary Dyck word suffix. Thus, the suffix is changed from $10^k 1^j 0^{i-k+1}$ to $01^j 0^{i-k+1} 10^{k-1}$. Again ⓒ's movement can be done in O(1) time.

Together, the movement of ⓐ and ⓒ cause the suffix of the corresponding k-ary Dyck word to be changed from $101^j 0^i$ to $01^j 0^{i-k+1} 10^{k-1}$, as desired.

Finally, we consider (6d) below.

	String	Successor
(6d)	$1^t 0^{(k-1)t}$	$1^{t-1} 0^{(k-1)(t-1)} 10^{k-1}$

Our algorithm begins by creating the k-ary tree corresponding to $1^{(k-1)t} 0^t$; we do not need to test for this case. This initial tree is the only tree where the value ⓐ is undefined. To apply (6d) we move the last internal node in pre-order so that it becomes the kth child of the root. The reinserted node then becomes the first value for ⓐ. This can be done in O(1) time.

Overall, the discussion in this subsection has proved the following theorem.

Theorem 5. *The k-ary trees with t internal nodes can be generated in reverse complemented cool-lex order by a loopless algorithm which uses O(t) additional variables.*

Implementations of Cool$\frac{1}{K}$, CoolK, and the k-ary tree algorithm are available from the authors. Our implementation of the k-ary tree algorithm terminates when the a pointer is set to the root, and the root pointer is never changed throughout the algorithm. We also mention that the additional memory used by the algorithm can be created "from scratch" for any tree in linear time. Thus, the successor algorithm from Table 6 takes O(n) time (without needing to translate it back and forth to k-ary Dyck words) and moreover, the loopless algorithm could be started from any tree given O(n)-time initialization. Each iteration of our k-ary tree implementation requires at most 20 variable updates (including child pointers, parent pointers, k-pointers, node labels, and the a pointer) for each $k \geqslant 3$. Given its increased generality, this compares favorably to the 16 pointer updates required by the loopless algorithm for generating binary trees in cool-lex order from [20]. (In this case, the successor algorithm found in Table 1 provides a Gray code since the issue found in Fig. 1(a) does not apply.)

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 $s \leqslant (k-1)t$ 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,

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

Let $N_k(t, s)$ 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 2. $N_k(t, s) = 0$ *if* $t = 0$, $N_k(t, s) = 1$ *if* $t > 0$ *and* $s = 0$, *and otherwise*

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

Proof. $\mathbb{D}_k(0, s) = \emptyset$ and $\mathbb{D}_k(t, 0) = \{1^t\}$ if $t > 0$. If $1 \leqslant s \leqslant v$, then $B1 \in \mathbb{D}_k(t, s)$ if and only if $B \in \mathbb{D}_k(t-1, s)$ and $B0 \in \mathbb{D}_k(t, s)$ if and only if $B \in \mathbb{D}_k(t, s-1)$. Thus, $N_k(t, s) = N_k(t-1, s) + N_k(t, s-1)$. If $v < s \leqslant (k-1)t$, then all strings in $\mathbb{D}_k(t, s)$ end in 0 and $B \in \mathbb{D}_k(t, s)$ if and only if $B0^{(k-1)t-s} \in \mathbb{D}_k(t)$. Thus, $N_k(t, s) = \frac{1}{kt+1}\binom{kt+1}{t}$ by the bijection between $\mathbb{D}_k(t)$ and k-ary trees with t internal nodes [11,34]. $\square$

Ruskey, Sawada, Williams [18] prove that the following recursive formula gives the cool-lex order of any bubble language $\mathbb{L}$. The formula is explained below.

$$\mathcal{C}(t, s, \gamma) = \begin{cases} \mathcal{C}(t-1, 1, 10^{s-1}\gamma), \ldots, \mathcal{C}(t-1, s-j, 10^j\gamma), 1^t 0^s \gamma & \text{if } t > 0; \quad \text{(a)} \\ 0^s \gamma & \text{if } t = 0. \quad \text{(b)} \end{cases} \tag{2}$$

If $1^t 0^s \gamma \in \mathbb{L}$ 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 (2a) 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, $1^t 0^s \gamma = 1^{t-1} 10^s \gamma = \mathcal{C}(t-1, 0, 10^s \gamma)$) is last instead of first. In fact, this is the only difference between cool-lex order and conventional "co-lex order" (see the comparison on page 292). The entire cool-lex order for some $\mathbb{L}$ with $1^t 0^s \in \mathbb{L}$ 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 $\mathcal{D}_k(t, s, \epsilon) = \mathcal{C}(t, s, \epsilon)$, where 'coolKat' is the k-ary Catalan generalization of 'coolCat'.

Lemma 3. *CoolKat order is* $\mathcal{D}_k(t, s, \gamma) = \epsilon$ *if* $t = 0$, *and otherwise*

$$\mathcal{D}_k(t, s, \gamma) = \begin{cases} \mathcal{D}_k(t-1, 1, 10^{s-1}\gamma), \ldots, \mathcal{D}_k(t-1, s, 1\gamma), 1^t 0^s \gamma & \text{if } s \leqslant v; \\ \mathcal{D}_k(t-1, 1, 10^{s-1}\gamma), \ldots, \mathcal{D}_k(t-1, v, 10^{s-v}\gamma), 1^t 0^s \gamma & \text{if } v < s \leqslant (k-1)t. \end{cases}$$

Proof. $\mathbb{L} = \mathbb{D}_k(t, s)$ is a bubble language, therefore $\mathcal{D}_k(t, s, \gamma)$ follows from (2a), (2b) by giving the minimum j such that 10^j is the suffix of a string in $\mathbb{L}$. If $s \leqslant v$, then $j = 0$ by $1^{t-1} 0^s 1 \in \mathbb{L}$. If $v < s \leqslant (k-1)t$, then $j = s - v$ by $1^{t-1} 0^s 10^{s-v} \in \mathbb{L}$. $\square$

Now we efficiently rank and unrank k-ary Dyck prefixes—with examples provided after Theorems 6 and 7. With respect to an ordered set of strings $\mathcal{L} = B_1, B_2, \ldots, B_m$, the *rank* of B_i is $\text{rank}(B_i, \mathcal{L}) = i - 1$, and $\text{unrank}(i-1, \mathcal{L}) = B_i$ for $1 \leqslant i \leqslant m$. For convenience define $R(B, \mathcal{L}) = \text{rank}(B, \mathcal{L}) + 1$. Also let $\mathcal{D}_k(t, s)$ denote $\mathcal{D}_k(t, s, \epsilon)$.

Theorem 6. *If* $B = \alpha 10^m \in \mathcal{D}_k(t, s)$ *for a (possibly empty) binary string* α *and* $m \geqslant 0$, *then* $R(B, \mathcal{D}_k(t, s))$ *is equal to*

$$\begin{cases} N_k(t, s) & \text{if } B = 1^t 0^s; \\ R(\alpha, \mathcal{D}_k(t-1, s-m)) + \sum_{i=1}^{s-m-1} N_k(t-1, i) & \text{if } B \neq 1^t 0^s \text{ and } s \leqslant v; \\ R(\beta, \mathcal{D}_k(t, v)) & \text{otherwise}, \end{cases}$$

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

Proof. If $B = 1^t 0^s$, then $R(B, \mathcal{D}_k(t, s)) = N_k(t, s)$ since B is last in $\mathcal{D}_k(t, s)$ by Lemma 3.

If $B \neq 1^t 0^s$ and $0 \leqslant s \leqslant v$, then $\mathcal{D}_k(t-1, i)$ appears before B in $\mathcal{D}_k(t, s)$ for $1 \leqslant i \leqslant s - m - 1$ by Lemma 3.

If $s > v$, then by Lemma 3 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, $R(B, \mathcal{D}_k(t, s)) = R(\beta, \mathcal{D}_k(t, v))$. $\square$

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

Theorem 7. *The value of* $U(x, \mathcal{D}_k(t, s))$ *is*

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

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

Table 7
The numbers $N_5(t, s)$.

$N_5(t, s)$	$s = 0$	$s = 1$	$s = 2$	$s = 3$	$s = 4$	$s = 5$	$s = 6$	$s = 7$	$s = 8$	$s = 9$	$s = 10$	$s = 11$	$s = 12$
$t = 1$	1	1	1	1	1								
$t = 2$	1	2	3	4	5	5	5	5	5				
$t = 3$	1	3	6	10	15	20	25	30	35	35	35	35	35

Proof. If $x = N_k(t, s)$, then $U(x, \mathcal{D}_k(t, s))$ is the last string in $\mathcal{D}_k(t, s)$ and by Lemma 3, $U(x, \mathcal{D}_k(t, s)) = 1^t 0^s$.

We now consider the case when $x < N_k(t, s)$ and $0 \leqslant s \leqslant v$. Let p be an integer such that $U(x, \mathcal{D}_k(t, s))$ is in $\mathcal{D}_k(t, p, 10^{s-p})$. By Lemma 3, $x > \sum_{i=1}^{p-1} N_k(t-1, i)$. It is now straightforward to observe that $y = p - 1$. Therefore, $U(x, \mathcal{D}_k(t, s)) = U(x - \sum_{i=1}^{y} N_k(t-1, i), \mathcal{D}_k(t-1, y+1))10^{s-y-1}$.

The remaining case is $x < N_k(t, s)$ and $s > v$. By Lemma 3, 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, $U(x, \mathcal{D}_k(t, s)) = U(x, \mathcal{D}_k(t, v))0^{s-v}$. □

It is straightforward to prove by induction that $\sum_{i=1}^{s} N_k(t-1, i) = N_k(t, s) - 1$. Therefore, we can achieve faster ranking and unranking by modifying the expressions for $R(B, \mathcal{D}_k(t, s))$ and $U(x, \mathcal{D}_k(t, s))$ as follows:

Corollary 1. *Assume that B, m, α, x and y have the same meaning as in Theorems 6 and 7. Then the value of $R(B, \mathcal{D}_k(t, s))$ is*

$$
\begin{cases}
N_k(t, s) & \text{if } B = 1^t 0^s; \\
R(\alpha, \mathcal{D}_k(t-1, s-m)) + N_k(t, s-m-1) - 1 & \text{if } B \neq 1^t 0^s \text{ and } s \leqslant v; \\
R(\beta, \mathcal{D}_k(t, v)) & \text{otherwise.}
\end{cases}
$$

The value of $U(x, \mathcal{D}_k(t, s))$ is

$$
\begin{cases}
1^t 0^s & \text{if } x = N_k(t, s); \\
U(x - N_k(t, y) + 1, \mathcal{D}_k(t-1, y+1))10^{s-y-1} & \text{if } x < N_k(t, s) \text{ and } s \leqslant v; \\
U(x, \mathcal{D}_k(t, v))0^{s-v} & \text{otherwise.}
\end{cases}
$$

We precompute and store the values of $N_k(t, s)$ in a table so that for any value of k, t, s, we can obtain $N_k(t, s)$ with a constant time table look-up. For example, Table 7 illustrates the first few values of $N_5(t, s)$. In the ranking and unranking processes we assume that such a table is computed in advance. The computation of the table $N_k(t, s)$ takes $O(tn)$ operations since the size of the table is $O(tn)$ and each entry can be computed with one addition by Lemma 2. Using this table we obtain $O(t + s)$-operation ranking and unranking algorithms for k-ary Dyck words using Corollary 1. Although our discussion has been in terms of cool-lex order and k-ary Dyck words, the same results hold for k-ary trees and complemented and reversed cool-lex order since converting between k-ary trees and k-ary Dyck words takes $O(n)$ time, as does complementing and reversing a binary string. Our result is summarized in Theorem 8.

Theorem 8. *The k-ary Dyck words of length $n = kt$ (and k-ary trees with t internal nodes) can be ranked and unranked using $O(kt)$ arithmetic operations on integers as large as the k-ary Catalan number $C_{t,k}$ in cool-lex order (and reverse complemented cool-lex order), so long as the table of $N_k(t, s)$ values is precomputed, which itself takes $O(tn)$ arithmetic operations.*

For example, to compute the rank of the string $100100010 \in \mathcal{D}_5(3, 6)$, we first compute $R(100100010, \mathcal{D}_5(3, 6))$ as follows:

$$
\begin{aligned}
R(100100010, \mathcal{D}_5(3, 6)) &= R(1001000, \mathcal{D}_5(2, 5)) + N_5(3, 6-1-1) - 1 \\
&= R(100, \mathcal{D}_5(1, 2)) + N_5(2, 5-3-1) - 1 + N_5(3, 4) - 1 \\
&= N_5(1, 2) + N_5(2, 1) - 1 + N_5(3, 4) - 1 \\
&= 16.
\end{aligned}
$$

Since $R(100100010, \mathcal{D}_5(3, 6)) = \text{rank}(100100010, \mathcal{D}_5(3, 6)) + 1$, therefore the rank of the string 100100010 in $\mathcal{D}_5(3, 6)$ is 15.

We now compute the string of $\mathcal{D}_5(3, 6)$ that has rank 15. Since $\text{unrank}(15) = U(16, \mathcal{D}_5(3, 6))$, we compute $U(16, \mathcal{D}_5(3, 6))$ as follows:

$$
\begin{aligned}
U(16, \mathcal{D}_5(3, 6)) &= U(16 - N_5(3, 4) + 1, \mathcal{D}_5(2, 5))10^{6-4-1} \\
&= U(2, \mathcal{D}_5(2, 5))10 \\
&= U(2, \mathcal{D}_5(2, 4))0^{5-(2-1)(5-1)}10
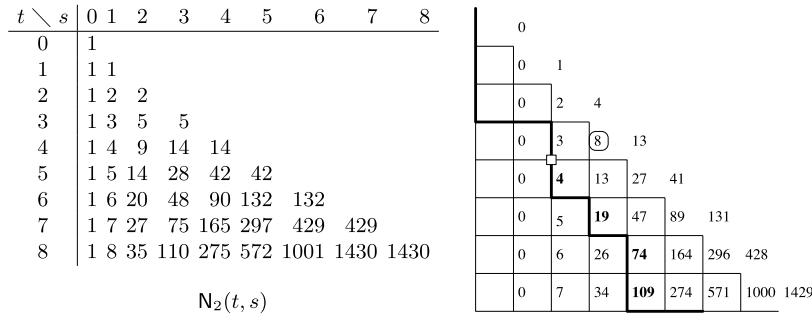\end{aligned}
$$

$t \setminus s$	0	1	2	3	4	5	6	7	8
0	1								
1	1	1							
2	1	2	2						
3	1	3	5	5					
4	1	4	9	14	14				
5	1	5	14	28	42	42			
6	1	6	20	48	90	132	132		
7	1	7	27	75	165	297	429	429	
8	1	8	35	110	275	572	1001	1430	1430

$\mathsf{N}_2(t,s)$

Lattice (right side): labels along the path —
0; 0, 1; 0, 2, 4; 0, 3, (8), 13; 0, 4, 13, 27, 41; 0, 5, 19, 47, 89, 131; 0, 6, 26, 74, 164, 296, 428; 0, 7, 34, 109, 274, 571, 1000, 1429.

Fig. 5. Ranking 11100110101100.

$$= \mathsf{U}\big(2 - \mathsf{N}_5(2,1) + 1, \mathcal{D}_5(1,2)\big)10^{4-1-1}010$$

$$= \mathsf{U}\big(1, \mathcal{D}_5(1,2)\big)100010$$

$$= 100100010.$$

Note that $\mathsf{R}(B, \mathcal{D}_k(t,s))$ ignores trailing 0s; the rank therefore depends only on the positions of the 1s. If $B = 1^t 0^s$, then the rank of B is $\mathsf{N}_k(t,s) - 1$. Otherwise, if $c_1, c_2, \ldots, c_t$ are the positions occupied by the 1s and q is the minimum value for which $c_q > q$, then $\mathsf{R}(B, \mathcal{D}_k(t,s))$, as expressed in Corollary 1, can be iterated to obtain

$$\mathsf{R}\big(B, \mathcal{D}_k(t,s)\big) = \mathsf{N}_k(q, c_q - q) - 1 + \sum_{j=q+1}^{t} \big(\mathsf{N}_k(j, c_j - j - 1) - 1\big).$$

Consequently, there is a nice way to view the ranking process as a walk on a certain integer lattice as illustrated in Fig. 5, where $k = 2$. The walk starts at the upper left; each 1 is a vertical step down and each 0 is a horizontal step to the right. The vertical edges are labeled, where the t-th row of vertical edges (counting from 1) gets labeled as follows from left-to-right: (no label), $\mathsf{N}_k(t,0) - 1, \mathsf{N}_k(t,1) - 1, \ldots, \mathsf{N}_k(t,v) - 1$. The label furthest to the right in each row is not on an edge. Fig. 5 illustrates the path for the bitstring 11100110101100. The square marks the endpoint of the part of the path that ends at the leftmost 01; i.e, the string 111001 in the example bitstring. The rank of the bitstring is obtained by summing the edge labels on the path after the square, adding the edge label on the edge to the right of the one that precedes the square (the circled label in Fig. 5), and then subtracting 1. Thus the rank of 11100110101100 is $4 + 19 + 74 + 109 + 8 - 1 = 213$.

To unrank we reverse the process. Suppose, for example, that we want to compute the string of $\mathcal{D}_2(8,6)$ that has rank 212. We start where the example path ends. We move to the left so long as the edge labels exceed the remaining rank, then move up and repeat. Arriving at the old square, we are at an impasse; the remaining rank is 7, so we have yet to encounter the square. So we move up and the rank becomes 4, which is what remains if we make the current location (one move above the old square) the new square. Thus the string of $\mathcal{D}_2(8,6)$ that has rank 212 is 11001110101100. We leave it to the reader to turn this description into an algorithm.

Observe that we can avoid computing the entire table for ranking and unranking if we can compute the values needed along the path by $O(1)$ arithmetic operations per move. This is possible for the case when $k = 2$, as proved in [20], using the property that for all $0 \leqslant s \leqslant t$, $\mathsf{N}_2(t,s) = \frac{t-s+1}{t+1}\binom{t+s}{t} = \binom{t+s}{t} - \binom{t+s}{t+1}$.

First compute $\mathsf{N}_2(t,s)$, which takes $O(n)$ arithmetic operations. Then make use of the following relations which can be checked using the closed form of $\mathsf{N}_2(t,s)$.

$$\mathsf{N}_2(t-1,s) = \frac{(t+1)(t-s)}{(t-s+1)(t+s)}\mathsf{N}_2(t,s) \quad \text{and} \quad \mathsf{N}_2(t,s-1) = \frac{s(t-s+2)}{(t-s+1)(t+s)}\mathsf{N}_2(t,s).$$

On the other hand, no nice closed form is currently known for $\mathsf{N}_k(t,s)$, where $k > 2$. Therefore, it would be interesting to examine whether the values needed along the path can be computed using $O(1)$ arithmetic operations per move even when $k > 2$. However, still we can avoid computing the entire table for ranking and unranking using Corollary 1 along with the following equation, as proved in [9].

$$\mathsf{N}_k(t,s+1) = \sum_{\lfloor \frac{t+s}{k} \rfloor + 1 \leqslant j \leqslant t} \frac{1}{j}\binom{kj}{j-1}\binom{t+s-kj}{t-j},$$

where $0 \leqslant s \leqslant v$ and $\binom{m}{n} = (-1)^n \binom{n-m-1}{n}$.

Of course, if many ranking/unranking operations are being performed then it will be better to pre-compute the $\mathsf{N}_k(t,s)$ table.

5. Final remarks

In this article we have considered cool-lex order and two of the most prominent k-ary Catalan structures—k-ary Dyck words and k-ary trees—in the context of combinatorial generation. Our results include algorithms for O(n)-time successors, loopless generation, and O(t) arithmetic operation ranking (subject to the standard table precomputation) for the reverse complemented variation of cool-lex. Furthermore, the loopless generation algorithm of k-ary Dyck words (and $1/k$-ary Dyck words) is very simple, and requires only two index variables of additional memory. As discussed in Section 1, one reason cool-lex order is able to obtain these results is because it is a Gray code that is remarkably similar to the lexicographic order known as co-lex order. In other words, it is it follows a simple pattern both locally and globally.

Several natural questions arise from this work:

1. Which other (k-ary) Catalan structures can be generated by a loopless algorithm in cool-lex order or one of its variants? Certain structures seem like natural candidates, such as the seventh structure in the "Catalan garden" [8] known as *k-ary good paths*.
2. As mentioned in Section 1, efficiently computable bijections between two structures allow for efficient successor and ranking algorithms to be translated between them. Which pairs of (k-ary) Catalan structures have bijections that can be computed in linear time?
3. Section 2.1 describes the general cool-lex successor algorithm for (fixed-weight binary) bubble languages, and Section 1.3 mentions that many combinatorial objects can be represented by bubble languages. Which of these combinatorial objects can be generated by a loopless algorithm in cool-lex order? Which of these objects can be efficiently ranked in cool-lex order?

Finally, we thank the anonymous referees who carefully corrected a number of errors and omissions.

References

[1] A. Ahmadi-Adl, A. Nowzari-Dalini, H. Ahrabian, Ranking and unranking algorithms for loopless generation of t-ary trees, Logic Journal of the IGPL 19 (1) (2011) 33–43.
[2] H. Ahrabian, A. Nowzari-Dalini, Generation of t-ary trees with ballot-sequences, International Journal of Computer Mathematics 80 (10) (2003) 1243–1249.
[3] D.R. van Baronaigien, F. Ruskey, Generating t-ary trees in A-order, Information Processing Letters 27 (4) (1988) 205–213.
[4] E. Canfield, S. Williamson, A loop-free algorithm for generating the linear extensions of a poset, Order 12 (1995) 57–75.
[5] S. Durocher, P.C. Li, D. Mondal, F. Ruskey, A. Williams, Ranking and loopless generation of k-ary Dyck words in cool-lex order, in: Proceedings of the 22nd International Workshop on Combinatorial Algorithms (IWOCA 2011), Victoria, Canada, 2011, in: LNCS, vol. 7056, Springer-Verlag, Berlin, 2011.
[6] G. Ehrlich, Loopless algorithms for generating permutations, combinations and other combinatorial configurations, Journal of the ACM 20 (3) (1973) 500–513.
[7] F. Gray, Pulse code communication, U.S. Patent 2,632,058, 1947.
[8] S. Heubach, N.Y. Li, T. Mansour, A garden of k-Catalan structures, http://www.scientificcommons.org/43469719, 2008.
[9] P. Hilton, J. Pedersen, Catalan numbers, their generalization, and their uses, The Mathematical Intelligencer 13 (2) (1991) 64–75.
[10] S.M. Johnson, Generation of permutations by adjacent transpositions, Mathematics of Computation 17 (1963) 282–285.
[11] D.E. Knuth, The Art of Computer Programming: Generating All Trees and History of Combinatorial Generation, vol. 4, Addison–Wesley, February 2006.
[12] D.E. Knuth, The Art of Computer Programming, vol. 4: Combinatorial Algorithms, Part 1, Addison–Wesley, 2010.
[13] Z. Kokosinski, A parallel dynamic programming algorithm for unranking t-ary trees, in: Proceedings of the 5th International Conference on Parallel Processing and Applied Mathematics, in: LNCS, vol. 3019, Springer, Berlin, 2004, pp. 255–260.
[14] M. Mareš, M. Straka, Linear-time ranking of permutations, in: 15th Annual European Symposium on Algorithms (ESA 2007), Eilat, Israel, October 8–10, 2007, in: LNCS, vol. 4698, Springer, Berlin, 2007, pp. 187–193.
[15] W. Myrvold, F. Ruskey, Ranking and unranking permutations in linear time, Information Processing Letters 79 (2001) 281–284.
[16] G. Pruesse, F. Ruskey, Generating the linear extensions of certain posets by transpositions, SIAM Journal on Discrete Mathematics 4 (3) (1991) 413–422.
[17] F. Ruskey, Generating t-ary trees lexicographically, SIAM Journal on Computing 7 (4) (1978) 424–439.
[18] F. Ruskey, J. Sawada, A. Williams, Binary bubble languages and cool-lex order, Journal of Combinatorial Theory, Series A 119 (2012) 155–169.
[19] F. Ruskey, A. Williams, Generating combinations by prefix shifts, in: Computing and Combinatorics, 11th Annual International Conference, COCOON '05, Kunming, China, 2005, in: LNCS, vol. 3595, Springer-Verlag, Berlin, 2005, pp. 570–576.
[20] F. Ruskey, A. Williams, Generating balanced parentheses and binary trees by prefix shifts, in: Proceedings of the 14th Computing: The Australasian Theory Symposium (CATS 2008), NSW, Australia, January 22–25, 2008, vol. 77 (CR-PIT), pp. 107–115.
[21] F. Ruskey, A. Williams, The coolest way to generate combinations, Discrete Mathematics 309 (17) (September 2009) 5305–5320.
[22] J. Sawada, A. Williams, A Gray code for fixed-density necklaces and Lyndon words in constant amortized time, Theoretical Computer Science, 2012, in press, http://dx.doi.org/10.1016/j.tcs.2012.01.013.
[23] R. Stanley, Enumerative Combinatorics, Cambridge University Press, 1997.
[24] H. Steinhaus, One Hundred Problems in Elementary Mathematics, Pergamon Press, 1963, reprinted by Dover Publications in 1979.
[25] A.E. Trojanowski, On the ordering, enumeration and ranking of k-ary trees, Department of Computer Science, University of Illinois at Urbana-Champaign, February 1977.
[26] A.E. Trojanowski, Ranking and listing algorithms for k-ary trees, SIAM Journal on Computing 7 (4) (1978) 492–509.
[27] H. Trotter, Algorithm 115: Perm, Communications of the ACM 5 (8) (August 1962) 434–435.
[28] V. Vajnovszki, T. Walsh, A loop-free two-close Gray-code algorithm for listing k-ary Dyck words, Journal of Discrete Algorithms 4 (4) (2006) 633–648.
[29] A. Williams, O(1)-time unsorting by prefix-reversals in a boustrophedon linked list, in: Fifth International Conference on Fun with Algorithms (FUN 2010), Ischia Island, Italy, June 2–4, 2010, in: LNCS, vol. 6099, Springer, Berlin, 2010, pp. 368–379.
[30] R.Y. Wu, Binary tree sequence rotations and t-ary tree enumerations, Ph.D. thesis, Department of Industrial Engineering Management, National Taiwan University of Science and Technology, Taiwan, January 2006.

[31] R.Y. Wu, J.M. Chang, C.H. Chang, Ranking and unranking of non-regular trees with a prescribed branching sequence, Mathematical and Computer Modelling 53 (5–6) (2011) 1331–1335.
[32] R.Y. Wu, J.M. Chang, Y.L. Wang, Loopless generation of non-regular trees with a prescribed branching sequence, The Computer Journal 53 (6) (2010) 661–666.
[33] L. Xiang, K. Ushijima, C. Tang, On generating k-ary trees in computer representation, Information Processing Letters 77 (5–6) (2001) 231–238.
[34] S. Zaks, Generation and ranking of k-ary trees, Information Processing Letters 14 (1) (1982) 44–48.