Greedy Gray codes for Dyck words and ballot sequences

Vincent Vajnovszki[1] and Dennis Wong[2] ✉

[1] Université de Bourgogne, Dijon, France
vvajnov@u-bourgogne.fr
[2] Macao Polytechnic University, Macao, China
cwong@uoguelph.ca

Abstract. We present a simple greedy algorithm for generating Gray codes for Dyck words and fixed-weight Dyck prefixes. Successive strings in our listings differ from each other by a transposition, that is, two bit changes. Our Gray codes are both homogeneous and suffix partitioned. Furthermore, we use our greedy algorithm to produce the first known homogeneous 2-Gray code for ballot sequences, which are Dyck prefixes of all weights. Our work extends a previous result on combinations by Williams [Conference proceedings: Workshop on Algorithms and Data Structures (WADS), LNTCS 8037:525-536, 2013].

Keywords: Dyck word · lattice path · balanced parentheses · ballot sequence · homogeneous Gray code · greedy algorithm.

1 Introduction

A *Dyck word* is a binary string with the same number of 1s and 0s such that any prefix contains at least as many 0s as 1s. Dyck words are in bijection with *balanced parentheses*, with an open bracket represented by a 0 and a close bracket represented by a 1 [4,7]. For example, all length six balanced parentheses are given by

$$((())), (()()), (())(), ()(()), ()()().$$

The Dyck words that correspond to the five balanced parentheses of length six are

000111, 001011, 001101, 010011, 010101.

Since the number of 0s and 1s of a Dyck word has to be the same, the length n of Dyck words has to be an even number. Dyck words can be used to encode lattice paths that end on their starting level and never pass below it.

A *ballot sequence* is a binary string of length n such that in any of its prefixes the number of 0s is greater than or equal to the number of 1s. As an example, the ten ballot sequences for length five are

00000, 00001, 00010, 00011, 01001, 00100, 00101, 00110, 01000, 01010.

Such a length n sequence encodes a ballot counting scenario involving two candidates in which the number of votes collected by the first candidate is always greater than or equal to those collected by the second candidate throughout the count. Ballot sequences are also known as *Dyck prefixes*, which are prefixes of Dyck words. Ballot sequences and Dyck prefixes can also be used to encode lattice paths that end on the positive region and never pass below it.

The number of Dyck words is known as the *Catalan number*, and the number of ballot sequences is known as the *ballot number*. The enumeration sequences of Dyck words and ballot sequences are A000108 and A001405 in the Online Encyclopedia of Integer Sequences respectively [23]. The enumeration formulae for the number of Dyck words and the number of ballot sequences [2] of length n are given as follows:

- Catalan number: $\dfrac{1}{\frac{n}{2}+1}\dbinom{n}{\frac{n}{2}}$;
- Ballot number: $\dbinom{n}{\lfloor \frac{n}{2} \rfloor}$.

Dyck words and ballot sequences are well studied combinatorial objects that have a wide variety of applications. For example, Dyck words have been used to encode a wide variety of combinatorial objects including binary trees, balanced parentheses, lattice paths, and stack-sortable permutations [4,7,8,11,14,19,27,31]. Ballot sequences, on the other hand, have many applications ranging from constructing more sums than differences (MSTD) sets [33], generating n-node binary trees of different shapes [1,16], and enumerating random walks with various constraints [3,6,10,12,29]. For more applications of Dyck words and ballot sequences, see [9,13,20,24].

One of the most important aspects of combinatorial generation is to list the instances of a combinatorial object so that consecutive instances differ by a specified *closeness condition* involving a constant amount of change. Lists of this type are called *Gray codes*. This terminology is due to the eponymous *binary reflected Gray code* (BRGC) by Frank Gray, which orders the 2^n binary strings of length n so that consecutive strings differ in one bit. For example, when $n = 4$ the order is

0000, 1000, 1100, 0100, 0110, 1110, 1010, 0010,
0011, 1011, 1111, 0111, 0101, 1101, 1001, 0001.

The BRGC listing is a *1-Gray code* in which consecutive strings differ by one symbol change. In this paper, we are focusing on *transposition Gray code*, where consecutive strings differ by swapping the positions of two bits. A transposition Gray code is also a *2-Gray code*, where consecutive strings differ by at most two bit changes.

Several algorithms have been proposed to generate Dyck words. Proskurowski and Ruskey [15] devised a transposition Gray code for Dyck words. Later, efficient algorithms to generate such a listing were presented in [17,28]. Bultena and Ruskey [5], and later van Baronaigien [26] and Xiang et al. [32], developed algorithms to generate homogeneous transposition Gray codes for Dyck words. For example, the algorithm by Bultena and Ruskey generates the 42 Dyck words for $n = 10$ as follows:

0101010101, 0011010101, 0010110101, 0100110101, 0001110101, 0001101101, 0100101101, 0010101101, 0011001101, 0101001101, 0100011101, 0010011101, 0001011101, 0000111101, 0000111011, 0001011011, 0010011011, 0100011011, 0101001011, 0011001011, 0010101011, 0100101011, 0001101011, 0001110011, 0100110011, 0010110011, 0011010011, 0101010011, 0101000111, 0011000111, 0010100111, 0100100111, 0001100111, 0001010111, 0010010111, 0100010111, 0000110111, 0000101111, 0001001111, 0010001111, 0100001111, 0000011111.

The Gray code is said to be *homogeneous*, where the bits between the swapped 0 and 1 are all 0s. Additionally, the Gray code is also a *suffix-partitioned* Gray code, where strings with the same suffix are contiguous. Vajnovszki and Walsh [25] discovered an even more restrictive Gray code that is *two-close*, where a 1 exchanges its position with an adjacent 0 or a 0 that is separated from it by a single 0. In contrast, Ruskey and Williams [18] provided a *shift Gray code* for Dyck words where consecutive strings differ by a prefix shift.

For ballot sequences, the problem of finding a Gray code for ballot sequences was first studied by Sabri and Vajnovszki [19]. Sabri and Vajnovszki proved that one definition of the reflected Gray code induces a 3-Gray code for k-ary ballot sequences, which is a generalization of ballot sequences that involves more than two candidates. Wong et al. [31] later provided an efficient algorithm to generate a 2-Gray code for ballot sequences. For example, the algorithm by Wong et al. generates the following cyclic 2-Gray code for ballot sequences for $n = 6$:

000111, 010011, 000011, 001011, 001001, 000001, 010001, 010101, 000101, 001101, 001100, 000100, 010100, 010000, 000000, 001000, 001010, 000010, 010010, 000110.

Another approach by Wong et al. to obtain a cyclic 2-Gray code for ballot sequences is by *filtering* the BRGC [31]. For more information about Gray codes induced by the BRGC, see [21] and [22]. However, these Gray codes for ballot sequences are not homogeneous. The greedy algorithm proposed in this paper can be used to generate the first known homogeneous 2-Gray code for ballot sequences.

2 Gray codes for Dyck words and fixed-weight Dyck prefixes

In this section, we present a greedy algorithm to generate transposition Gray codes for fixed-weight Dyck prefixes and Dyck words.

In [30], Williams proposed a greedy algorithm to generate a transposition Gray code for combinations. The greedy algorithm by Williams can be summarized as follows:

Greedy Gray code algorithm for k-combinations: Starts with 1^k0^{n-k}. Greedily swap the leftmost possible 1 with the leftmost possible 0 before the next 1 and after the previous 1 (if there are any) such that the resulting string has not appeared before.

For example, the greedy algorithm generates the following 4-combinations for $n = 7$:

1111000, 1110100, 1101100, 1011100, 0111100, 0111010, 1011010,
1101010, 1110010, 1100110, 1010110, 0110110, 0101110, 1001110,
0011110, 0011101, 1001101, 0101101, 0110101, 1010101, 1100101,
1110001, 1101001, 1011001, 0111001, 0110011, 1010011, 1100011,
1001011, 0101011, 0011011, 0010111, 1000111, 0100111, 0001111.

We generalize the idea to fixed-weight Dyck prefixes and Dyck words. The *weight* of a binary string is the number of 1s it contains. A *fixed-weight Dyck prefix* of weight k is a prefix of a Dyck word with its weight equal to k. Note that when $2k = n$, then the set of fixed-weight Dyck prefixes of weight k is equivalent to the set of Dyck words. The following simple greedy algorithm generates transposition Gray codes for fixed-weight Dyck prefixes and Dyck words of length n:

> **Greedy Gray code algorithm for fixed-weight Dyck prefixes**: Starts with $(01)^k0^{n-2k}$. Greedily swap the leftmost possible 1 with the leftmost possible 0 before the next 1 and after the previous 1 (if there are any) such that the resulting string is a Dyck prefix and has not appeared before.

Our Gray codes for fixed-weight Dyck prefixes and Dyck words are homogeneous and suffix-partitioned. Another way to understand the greedy algorithm is to greedily swap the leftmost possible 1 with the leftmost possible 0 in a homogeneous manner. As an example, the greedy algorithm generates the following Gray code for Dyck words for $n = 10$ (Dyck prefixes for $n = 10$ and $k = 5$):

0101010101, 0011010101, 0010110101, 0100110101, 0001110101, 0001101101,
0100101101, 0010101101, 0011001101, 0101001101, 0100011101, 0010011101,
0001011101, 0000111101, 0000111011, 0100011011, 0010011011, 0001011011,
0001101011, 0100101011, 0010101011, 0011001011, 0101001011, 0101010011,
0011010011, 0010110011, 0100110011, 0001110011, 0001100111, 0100100111,
0010100111, 0011000111, 0101000111, 0100010111, 0010010111, 0001010111,
0000110111, 0000101111, 0100001111, 0010001111, 0001001111, 0000011111.

Greedy Gray codes have been studied previously, with Williams [30] reinterpreting many classic Gray codes for binary strings, permutations, combinations, binary trees, and set partitions using a simple greedy algorithm. The algorithm presented in this paper can be considered as a novel addition to the family of greedy algorithms previously studied by Williams.

All strings considered in this paper are binary. Our algorithm uses a vector representation $S_1S_2 \cdots S_k$ to represent a binary string with k ones, where each integer S_i corresponds to the position of the i-th one of the binary string. For example, the string $\alpha =$ 000110100011001 can be represented by $S_1, S_2, S_3, S_4, S_5, S_6 = 4, 5, 7, 11, 12, 15$. We initialize the array $S_1, S_2, \ldots, S_k = 2, 4, \ldots, 2k$ for both Dyck words and fixed-weight Dyck prefixes. In addition, we set $S_0 = 0$ and $S_{k+1} = n + 1$. Pseudocode of the greedy algorithm to generate fixed-weight Dyck prefixes and Dyck words is given in Algorithm 1.

Algorithm 1 The greedy algorithm that generates a homogeneous transposition Gray code for fixed-weight Dyck prefixes and Dyck words.

```
1: procedure GREEDY-kDYCK-PREFIXES
2:     $S_1S_2 \cdots S_k \leftarrow 2\,4 \cdots 2k$
3:     Print($S_1S_2 \cdots S_k$)
4:     for $i$ from $1$ to $k$ do
5:         for $j$ from MAX($S_{i-1}+1, i \times 2$) to $S_{i+1}-1$ do
6:             if $S_1S_2 \cdots S_{i-1}(j)S_{i+1} \cdots S_k$ has not appeared before then
7:                 $S_i \leftarrow j$
8:                 go to 4
```

Theorem 1. *The algorithm Greedy-kDyck-Prefixes generates a homogeneous transposition Gray code for fixed-weight Dyck prefixes that is suffix-partitioned for all n and k where $2k \leq n$.*

3 Proof of Theorem 1

In this section, we prove Theorem 1 for fixed-weight Dyck prefixes. The results also apply to Dyck words as the set of Dyck words is equivalent to the set of fixed-weight Dyck prefixes when $2k = n$. To this end, we begin by proving the following lemmas for fixed-weight Dyck prefixes.

Lemma 1. *The algorithm Greedy-kDyck-Prefixes terminates after visiting the Dyck prefix $0^{n-k}1^k$.*

Proof. Assume the algorithm terminates after visiting some string $b_1b_2 \cdots b_n \neq 0^{n-k}1^k$. Since $b_1b_2 \cdots b_n \neq 0^{n-k}1^k$, it must contain the suffix 10^i1^j for some $n-k > i > 0$ and $k > j > 0$. It follows by the greedy algorithm that there exists a Dyck prefix of length n and weight k with the suffix 0^i1^{j+1} in the listing since the algorithm terminates after visiting a string with the suffix 10^i1^j. If $j+1 = k$, then clearly the only string with the suffix 0^i1^k is $0^{n-k}1^k$. However, this string has a predecessor since it is not the initial string of the greedy algorithm. Moreover, by the greedy algorithm the predecessor of $0^{n-k}1^k$ is $0^t10^{n-k-t}1^{k-1}$ for some $n-k > t > 0$, and all Dyck prefixes of length n and weight k with the suffix 01^{k-1} must have appeared before $0^{n-k}1^k$ in the listing. Therefore, the algorithm should terminate after visiting $0^{n-k}1^k$, a contradiction. Otherwise if $j+1 < k$, then let α be the last string in the listing with the suffix 10^t1^{j+1} for some $n-j-2 > t > 0$. Since α appears before $b_1b_2 \cdots b_n$ in the listing and $b_1b_2 \cdots b_n$ has the suffix 10^i1^j, the algorithm must transpose the first 1 in the suffix 1^{j+1} of α with a 0 on the left to produce a later string with the suffix 01^j. It follows by the greedy algorithm that this is only possible if a string with the suffix 0^t1^{j+2} appears before in the listing. Recursively applying the same argument implies that $0^{n-k}1^k$ exists in the listing, a contradiction since the algorithm would terminate after visiting $0^{n-k}1^k$ as discussed in the case of $j+1 = k$. Therefore by proof by contradiction, the greedy algorithm terminates after visiting $0^{n-k}1^k$. □

Lemma 2. *If $0^i1^j0^t1\gamma$ is a length n Dyck prefix with weight k for some $i > 0$, $k > j > 0$, and $t > 0$, then the non-existence of $0^i1^j0^t1\gamma$ in the greedy listing implies the non-existence of $0^i1^{j-1}010^{t-1}1\gamma$ in the greedy listing.*

Proof. We prove the lemma by contrapositive. Suppose $\alpha = 0^i1^j0^t1\gamma$ is a Dyck prefix of weight k. Clearly $\beta = 0^i1^{j-1}010^{t-1}1\gamma$ is also a Dyck prefix of weight k and now consider the possible predecessor of β in our greedy listing. If the predecessor of β is of the form $0^{i-p}10^p1^{j-2}010^{t-1}1\gamma$ for some $p > 0$, then by the greedy algorithm, all Dyck prefixes of length n and weight k with the suffix $01^{j-2}010^{t-1}1\gamma$ should have appeared previously. The next string generated by the algorithm after β is thus α if α has not appeared before, or otherwise α must have appeared previously. In either case, α exists in the listing. Otherwise if the predecessor of β shares the same prefix 0^i1^{j-1} as β, then by the greedy algorithm, this is only possible if α appears before in the listing or α is the predecessor of β. Therefore, the string α exists if β exists, which completes the proof by contrapositive. □

We now prove Theorem 1 using the lemmas we proved in this section.

Theorem 1. *The algorithm Greedy-kDyck-Prefixes generates a homogeneous transposition Gray code for fixed-weight Dyck prefixes that is suffix-partitioned for all n and k where $2k \leq n$.*

Proof. Our algorithm permits only homogeneous transposition operations, and the listing is suffix-partitioned (as shown in Lemma 2). To demonstrate the Gray code property of our algorithm, we now prove it by contradiction.

Since the greedy algorithm ensures that there is no duplicated length n string in the greedy listing, it suffices to show that each Dyck prefix of length n and weight k appears in the listing.

Assume by contradiction that there exists a Dyck prefix $b_1b_2\cdots b_n \neq 0^{n-k}1^k$ that does not appear in the listing. Since $b_1b_2\cdots b_n \neq 0^{n-k}1^k$, the string $b_1b_2\cdots b_n$ contains the substring 10. Let $b_1b_2\cdots b_n = 0^i1^j0^t1\gamma$ for some $i > 0$, $k > j > 0$, and $t > 0$. Clearly, the string $0^i1^{j-1}010^{t-1}1\gamma$ is a Dyck prefix and by Lemma 2, the string $0^i1^{j-1}010^{t-1}1\gamma$ also does not exist in the greedy Dyck prefix listing. Repeatedly applying the same argument on $0^i1^{j-1}010^{t-1}1\gamma$ implies that the strings $0^{i+1}1^j0^{t-1}1\gamma$ and eventually $0^{n-k}1^k$ also do not exist in the listing, a contradiction to Lemma 1. □

4 Gray codes for ballot sequences

In this section, we leverage Theorem 1 to construct the first known homogeneous 2-Gray code for ballot sequences. Our approach is to interleave strings from listings of homogeneous transposition Gray codes for fixed-weight Dyck prefixes, across all possible weight k, in order to create the homogeneous 2-Gray code for ballot sequences. To achieve this, we first prove the following lemma.

Lemma 3. *The string $b_1b_2\cdots b_{n-1}1$ is a Dyck prefix if and only if $b_1b_2\cdots b_{n-1}0$ is a Dyck prefix, provided that $2k < n-1$.*

Proof. The forward direction is straightforward. For the backward direction, suppose that $2k < n-1$ and that the string $b_1b_2\cdots b_{n-1}0$ is a Dyck prefix. Since $2k < n-1$, the prefix $b_1b_2\cdots b_{n-1}$ has more 0s than 1s and thus both $b_1b_2\cdots b_{n-1}1$ and $b_1b_2\cdots b_{n-1}0$ are Dyck prefixes. □

By Lemma 3, we can establish a one-to-one correspondence between Dyck prefixes $b_1b_2\cdots b_{n-1}1$ of weight $k+1$ and Dyck prefixes $b_1b_2\cdots b_{n-1}0$ of weight k when $2k < n$. This correspondence enables us to construct a homogeneous 2-Gray code for ballot sequences.

The main idea of our algorithm is to utilize the same greedy strategy used for generating fixed-weight Dyck prefixes, with the addition of generating the correspondence to the generated Dyck prefix by Lemma 3. Specifically, whenever we produce a Dyck prefix $b_1b_2\cdots b_n$ that terminates with a 1, we also generate its corresponding Dyck prefix $b_1b_2\cdots b_{n-1}0$. Conversely, when we generate a Dyck prefix $b_1b_2\cdots b_n$ that concludes with a 0 with $2k < n-1$, we also generate its corresponding Dyck prefix $b_1b_2\cdots b_{n-1}1$. Furthermore, if the application of the greedy strategy fails to produce a new string, we proceed to complement the last 1 in $b_1b_2\cdots b_{n-1}$ and then update the value of $b_n = 1$. By making two relatively minor changes to the Algorithm 1, we can generate a homogeneous 2-Gray code for ballot sequences:

1. Before applying the greedy strategy to the current string $S_1S_2\cdots S_k$, test whether $S_k = n$ or $S_k < n$ but with weight $k < \lfloor\frac{n}{2}\rfloor$. If $S_k = n$, then the algorithm generates its corresponding Dyck prefix $S_1S_2\cdots S_{k-1}$. Similarly, if $S_k < n$ but with weight $k < \lfloor\frac{n}{2}\rfloor$, then the algorithm generates its corresponding Dyck prefix $S_1S_2\cdots S_kn$;
2. After applying the greedy strategy to the current string $S_1S_2\cdots S_k$ and it does not lead to the generation of any new string. If $S_k = n$, then the next string in the sequence is $S_1S_2\cdots S_{k-2}n$. On the other hand, if $S_k < n$, then the following string in the sequence is $S_1S_2\cdots S_{k-1}n$.

The algorithm starts with the initial string $(01)^k0^{n-2k}$ with $k = \lfloor\frac{n}{2}\rfloor$. Pseudocode of the algorithm to generate the Gray code for ballot sequences is given in Algorithm 2. As an example, the algorithm generates the following homogeneous 2-Gray code for ballot sequences for $n = 7$:

0101010, 0011010, 0010110, 0100110, 0001110, 0001101, 0001100,
0100100, 0100101, 0010101, 0010100, 0011000, 0011001, 0101001,
0101000, 0100010, 0100011, 0010011, 0010010, 0001010, 0001011,
0000111, 0000110, 0000101, 0000100, 0100000, 0100001, 0010001,
0010000, 0001000, 0001001, 0000011, 0000010, 0000001, 0000000.

Let α be a prefix of a Dyck word, and $\mathcal{G}(\alpha)$ be the list of strings obtained by applying Algorithm 1 with α as initial string. Clearly, for any such string α, $\mathcal{G}(\alpha)$ contains prefixes of Dyck words of the same length and same number of 1s as α, and in $\mathcal{G}(\alpha)$ there are no repeated strings.

Algorithm 2 The greedy algorithm that generates a homogeneous 2-Gray code for ballot sequences.

```
1: procedure GREEDY-BALLOT
2:     $k = \lfloor \frac{n}{2} \rfloor$
3:     $S_1 S_2 \cdots S_k \leftarrow 2\,4 \cdots 2k$
4:     Print($S_1 S_2 \cdots S_k$)
5:     if $S_k = n$ then
6:         $S_k \leftarrow n + 1$
7:         $k \leftarrow k - 1$
8:         if $S_1 S_2 \cdots S_{i-1}(j) S_{i+1} \cdots S_k$ has not appeared before then go to 4
9:         $k \leftarrow k + 1$
10:        $S_k \leftarrow n$
11:    else if $k < \lfloor \frac{n}{2} \rfloor$ then
12:        $S_{k+1} \leftarrow n$
13:        $k \leftarrow k + 1$
14:        if $S_1 S_2 \cdots S_{i-1}(j) S_{i+1} \cdots S_k$ has not appeared before then go to 4
15:        $k \leftarrow k - 1$
16:        $S_{k+1} \leftarrow n + 1$
17:    for $i$ from $1$ to $k$ do
18:        for $j$ from MAX($S_{i-1} + 1, i \times 2$) to $S_{i+1} - 1$ do
19:            if $S_1 S_2 \cdots S_{i-1}(j) S_{i+1} \cdots S_k$ has not appeared before then
20:                $S_i \leftarrow j$
21:                go to 4
22:    if $S_k = n$ then
23:        $S_k \leftarrow n + 1$
24:        $S_{k-1} \leftarrow n$
25:        $k \leftarrow k - 1$
26:        go to 4
27:    else if $S_k = n - 1$ then
28:        $S_k \leftarrow n$
29:        go to 4
```

Theorem 2. *The algorithm Greedy-Ballot generates a homogeneous 2-Gray code for ballot sequences for all n.*

Proof. The algorithm Greedy-Ballot starts with the string $(01)^{\lfloor \frac{n}{2} \rfloor} 0^{n \bmod 2}$ with $k = \lfloor \frac{n}{2} \rfloor$. By Theorem 1, the algorithm generates all strings in $\mathcal{G}((01)^{\lfloor \frac{n}{2} \rfloor} 0^{n \bmod 2})$ which contains all Dyck prefixes of weight $k = \lfloor \frac{n}{2} \rfloor$. Furthermore, according to Lemma 3 and lines 5-16 of the algorithm, the algorithm also generates all Dyck prefixes of weight $\lfloor \frac{n}{2} \rfloor - 1$ that end with a 0.

Since $\mathcal{G}((01)^{\lfloor \frac{n}{2} \rfloor} 0^{n \bmod 2})$ ends with $0^{n - \lfloor \frac{n}{2} \rfloor} 1^{\lfloor \frac{n}{2} \rfloor}$, the algorithm generates all Dyck prefixes of weight $k = \lfloor \frac{n}{2} \rfloor$ and Dyck prefixes of weight $\lfloor \frac{n}{2} \rfloor - 1$ that end with a 0 until it reaches the string $0^{n - \lfloor \frac{n}{2} \rfloor} 1^{\lfloor \frac{n}{2} \rfloor}$ or $0^{n - \lfloor \frac{n}{2} \rfloor} 1^{\lfloor \frac{n}{2} \rfloor - 1} 0$. Then, as indicated in lines 22-29 of the algorithm, the next string generated by the algorithm is $0^{n - \lfloor \frac{n}{2} \rfloor} 1^{\lfloor \frac{n}{2} \rfloor - 2} 01$. Observe that $0^{n - \lfloor \frac{n}{2} \rfloor} 1^{\lfloor \frac{n}{2} \rfloor - 2} 01$ is generated in $\mathcal{G}((01)^{\lfloor \frac{n}{2} \rfloor - 1} 0^2 0^{n \bmod 2})$ by Algorithm 1 after exhaustively generating all Dyck prefixes of weight $\lfloor \frac{n}{2} \rfloor - 1$ that end with a 0.

Since all Dyck prefixes of weight $\lfloor \frac{n}{2} \rfloor - 1$ that end with a 0 have already been generated in our ballot sequence algorithm, the algorithm follows the same operations as $\mathcal{G}(0^{n-\lfloor \frac{n}{2} \rfloor}1^{\lfloor \frac{n}{2} \rfloor - 2}01)$ and proceeds to generate all Dyck prefixes of weight $\lfloor \frac{n}{2} \rfloor - 1$ that end with a 1. Therefore, all Dyck prefixes of weight $\lfloor \frac{n}{2} \rfloor - 1$ are in the listing generated by the algorithm.

By repeatedly applying the same argument, the algorithm generates the fixed-weight Dyck prefixes with weight ranging from k to 0, which is the set of all ballot sequences of length n.

Moreover, since each listing in $\mathcal{G}$ is a homogeneous transposition Gray code and the operations in lines 5-16 and 22-29 of the algorithm only involve removing a 1 or swapping two nearby bits, the resulting sequence generated by the algorithm is a homogeneous 2-Gray code. □

5 Final Remarks

It is worth noting that an alternative homogeneous 2-Gray code for ballot sequences can be constructed by concatenating the homogeneous transposition Gray code listings of fixed-weight Dyck prefixes ranging from weight k to 0, and reversing the listings of fixed-weight Dyck prefixes with even (or odd) weights. For instance, let $\overline{\mathcal{G}}(\alpha)$ denote the reverse of the list of strings generated by applying Algorithm 1 with α as the initial string. A homogeneous 2-Gray code for ballot sequences for $n = 7$ can be obtained by $\mathcal{G}(0101010) \cdot \overline{\mathcal{G}}(0101000) \cdot \mathcal{G}(0100000) \cdot \overline{\mathcal{G}}(0000000)$, which would result in the following listing:

0101010, 0011010, 0010110, 0100110, 0001110, 0001101, 0100101,
0010101, 0011001, 0101001, 0100011, 0010011, 0001011, 0000111,
0000011, 0001001, 0010001, 0100001, 0000101, 0000110, 0010010,
0100010, 0001010, 0001100, 0100100, 0010100, 0011000, 0101000,
0100000, 0010000, 0001000, 0000100, 0000010, 0000001, 0000000.

There is, however, no known simple algorithm to generate the reverse of the sequence generated by our algorithm for fixed-weight Dyck prefixes. This remains an open problem for future research.

Finally, efficient algorithms that generate the same Gray codes for Dyck words, fixed-weight Dyck prefixes and ballot sequences in constant amortized time per string were developed, and their details will be presented in the full version of the paper.

Acknowledgements

The research is supported by the Macao Polytechnic University research grant (Project code: RP/FCA-02/2022) and the National Research Foundation (NRF) grant funded by the Ministry of Science and ICT (MSIT), Korea (No. 2020R1F1A1A01070666).

A part of this work was done while the second author was visiting Kanazawa University in Japan. The second author would like to thank Hiroshi Fujisaki for his hospitality during his stay in Kanazawa.

References

1. H. Ahrabian and A. Nowzari-Dalini. Generation of t-ary trees with ballot-sequences. *Int. J. Comput. Math.*, 80(10):1243–1249, 2003.
2. M. Aigner. Enumeration via ballot numbers. *Discrete Math.*, 308(12):2544–2563, 2008.
3. D. Barton and C. Mallows. Some aspects of the random sequence. *Ann. Math. Stat.*, 36(1):236 – 260, 1965.
4. S. Benchekroun and P. Moszkowski. A new bijection between ordered trees and legal bracketings. *European J. Combin.*, 17(7):605–611, 1996.
5. B. Bultena and F. Ruskey. An Eades-McKay algorithm for well-formed parentheses strings. *Inf. Process. Lett.*, 68(5):255–259, 1998.
6. L. Carlitz. Sequences, paths, ballot numbers. *Fibonacci Quart*, 10(5):531–549, 1972.
7. E. Deutsch. A bijection on Dyck paths and its consequences. *Discrete Math.*, 179(1):253–256, 1998.
8. E. Deutsch and L. Shapiro. A bijection between ordered trees and 2-Motzkin paths and its many consequences. *Discrete Math.*, 256(3):655–670, 2002.
9. I. Goulden and D. Jackson. *Combinatorial Enumeration*. A Wiley-Interscience Publication. John Wiley & Sons Inc., New York, 1983.
10. B. Hackl, C. Heuberger, H. Prodinger, and S. Wagner. Analysis of bidirectional ballot sequences and random walks ending in their maximum. *Ann. Comb.*, 20(4):775–797, 2016.
11. J. Labelle and Y. N. Yeh. Generalized Dyck paths. *Discrete Math.*, 82(1):1–6, 1990.
12. T. Lengyel. Direct consequences of the basic ballot theorem. *Stat. Probab. Lett.*, 81(10):1476–1481, 2011.
13. T. Mütze. Combinatorial Gray codes - an updated survey. arXiv Preprint, Feb. 2022. arXiv:2202.01280
14. A. Panayotopoulos and A. Sapounakis. On binary trees and Dyck paths. *Mathématiques et Sciences Humaines*, 131:39–51, 1995.
15. A. Proskurowski and F. Ruskey. Binary tree Gray codes. *J. Algorithms*, 6(2):225–238, 1985.
16. D. Rotem and Y. Varol. Generation of binary trees from ballot sequences. *J. ACM*, 25(3):396–404, 1978.
17. F. Ruskey and A. Proskurowski. Generating binary trees by transpositions. *J. Algorithms*, 11(1):68–84, 1990.
18. F. Ruskey and A. Williams. Generating balanced parentheses and binary trees by prefix shifts. In *Proceedings of the Fourteenth Symposium on Computing: The Australasian Theory - Volume 77*, CATS '08, page 107–115, AUS, 2008.
19. A. Sabri and V. Vajnovszki. On the exhaustive generation of generalized ballot sequences in lexicographic and Gray code order. *Pure Math. Appl.*, 28(1):109–119, 2019.
20. C. Savage. A survey of combinatorial Gray codes. *SIAM Review*, (4):605–629, 1997.
21. J. Sawada, A. Williams, and D. Wong. Inside the binary reflected Gray code: Flip-swap languages in 2-Gray code order. In T. Lecroq and S. Puzynina, editors, *Combinatorics on Words*, pages 172–184, Cham, 2021. Springer International Publishing.

22. J. Sawada, A. Williams, and D. Wong. Flip-swap languages in binary reflected Gray code order. *Theor. Comput. Sci.*, 933:138–148, 2022.
23. N. Sloane. The on-line encyclopedia of integer sequences, `http://oeis.org`. Sequence A000108 and A001405.
24. D. Stanton and D. White. *Constructive Combinatorics*. Springer Science & Business Media, 2012.
25. V. Vajnovszki and T. Walsh. A loop-free two-close Gray-code algorithm for listing k-ary Dyck words. *J. Discrete Algorithms*, 4(4):633–648, 2006.
26. D. van Baronaigien. A loopless Gray-code algorithm for listing k-ary trees. *J. Algorithms*, 35(1):100–107, 2000.
27. G. Viennot. Theorié combinatoire des nombres d'Euler et de Genocchi. *Séminaire de théorie des nombres, Publications Univ. Bordeaux I*, 1980.
28. T. Walsh. Generation of well-formed parenthesis strings in constant worst-case time. *J. Algorithms*, 29(1):165–173, 1998.
29. M. Wildon. Knights, spies, games and ballot sequences. *Discrete Math.*, 310(21):2974–2983, 2010.
30. A. Williams. The greedy Gray code algorithm. In F. Dehne, R. Solis-Oba, and J.-R. Sack, editors, *Algorithms and Data Structures*, pages 525–536, Berlin, Heidelberg, 2013.
31. D. Wong, F. Calero, and K. Sedhai. Generating 2-Gray codes for ballot sequences in constant amortized time. *Discrete Math.*, 346(1):113168, 2023.
32. L. Xiang, K. Ushijima, and C. Tang. Efficient loopless generation of Gray codes for k-ary trees. *Inf. Process. Lett.*, 76(4):169–174, 2000.
33. Y. Zhao. Constructing MSTD sets using bidirectional ballot sequences. *J. Number Theory*, 130(5):1212–1220, 2010.

Appendix: C code to generate homogeneous 2-Gray codes for k-combinations, Dyck words, fixed-weight Dyck prefixes, and ballot sequences

```c
#include <stdio.h>
#include <stdlib.h>
#define INF 99999
#define MAX(a,b) (((a)>(b))?(a):(b))

int n, k, type, total = 0, s[INF], p[INF];

//-----------------------------------------------
int binToDec() {
   int i, j = 1, t = 0;
   for(i=1; i<=n; i++) if (s[j]==i) {t = t+(1<<(n-i)); j++;}
   return t;
}

//-----------------------------------------------
int greedy() {
   int i, j, t, r;

   if (type==4) {
      if (s[k]==n) {
         s[k] = n+1; k--;
         if (!p[binToDec()]) {p[binToDec()] = 1; return 1;}
         k++; s[k] = n;
      }
      else if (k<n/2) {
         s[k+1] = n; k++;
         if (!p[binToDec()]) {p[binToDec()] = 1; return 1;}
         k--; s[k+1] = n+1;
      }
   }

   for (i=1; i<=k; i++) {
      if (type==1) r = s[i-1]+1;
      else r = MAX(s[i-1]+1, i*2);

      for (j=r; j<s[i+1]; j++) {
         t = s[i]; s[i] = j;
         if (!p[binToDec()]) {p[binToDec()] = 1; return 1;}
         s[i] = t;
      }
   }

   if (type==4) {
      if (s[k]==n) {
         s[k] = n+1; s[k-1] = n; k--;
         p[binToDec()] = 1; return 1;
      }
      else if (s[k]==n-1) {
         s[k] = n;
         p[binToDec()] = 1; return 1;
      }
   }
   return 0;
}

//-----------------------------------------------
int main() {
   int i, j;

   printf(" ========================================\n");
   printf(" 1. Combinations\n");
   printf(" 2. Dyck words\n");
   printf(" 3. Prefix of Dyck words of weight k\n");
   printf(" 4. Ballot sequences\n");
   printf(" ========================================\n");

   printf(" Enter selection #: "); scanf("%d", &type);

   printf(" ENTER n: "); scanf("%d", &n);
   if (type!=2 && type!=4) {printf(" ENTER k: "); scanf("%d", &k);}
   else k = n/2;
   if (type==2 && n%2>0) {printf("n must be an even number. \n"); exit(0);}
   if (type==3 && k>n/2) {printf("k must be less than or equal to n/2. \n"); exit
        (0);}

   for (i=0; i<INF; i++) p[i] = 0;
   for (i=0; i<=k; i++) {if (type!=1) s[i] = i*2; else s[i] = i;}

   s[0] = 0; s[k+1] = n+1;
   p[binToDec()] = 1;

   do {
      j = 1;
      for (i=1; i<=n; i++) if (s[j]!=i) printf("0"); else {printf("1"); j++;}
      printf("\n"); total++;
   } while(greedy());
   printf("Total = %d\n", total);
}
```