Stamp Foldings, Semi-meanders, and Open Meanders: Fast Generation Algorithms

Joe Sawada*
School of Computer Science
University of Guelph
CANADA
jsawada@uoguelph.ca

Roy Li
School of Computer Science
University of Guelph
CANADA
royli0106@hotmail.com

Submitted: Nov 12, 2010; Accepted: Jun 2, 2012; Published: Jun 13, 2012
Mathematics Subject Classifications: 68R05, 05A99

Abstract

By considering a permutation representation for stamp-foldings and semi-meanders we construct tree-like data structures that will allow us to generate these objects in constant amortized time. Additionally, by maintaining the wind-factor and applying an additional optimization, the algorithm for semi-meanders can be modified to produce the fastest known algorithm to generate open meanders.

Keywords: Stamp folding, semi-meander, meander, CAT algorithm, permutation

1 Introduction

An *open meander* can be described by a geographic analogy of a river starting from the northwest and meandering back and forth across an infinite horizontal road. The river never intersects itself and it can flow freely to the east. The *order* of an open meander is the number of times the river intersects or crosses the road. For example, an open meander of order 6 is shown in Figure 1(a). A *semi-meander* is a slight generalization of an open meander where one of the end points is allowed to be *wound* inside the river. An example of a semi-meander, that is not a meander is shown in Figure 1(b). If we generalize a step further, and allow both ends of the river to be wound up inside itself, then we obtain a *stamp folding*. An analogy is to consider a folding of a linear strip of n stamps into a single pile, where the perforations between the

*Supported by NSERC.

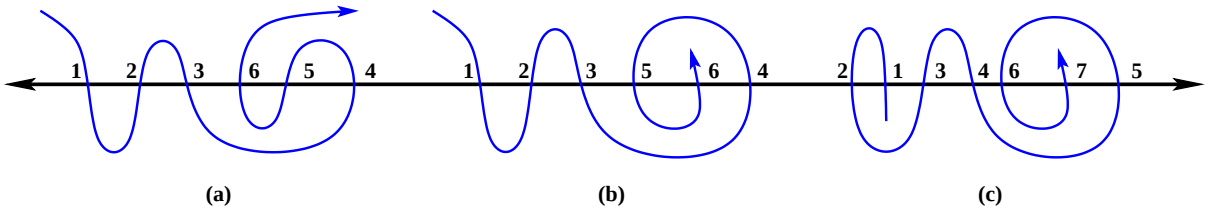
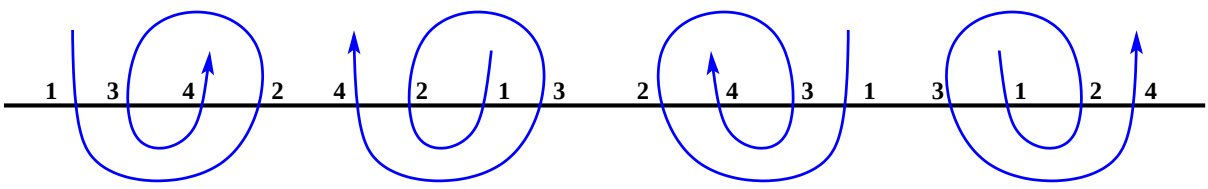


Figure 1: (a) An open meander of order 6. (b) A semi-meander of order 6. (c) A stamp folding of order 7.

stamps are assumed to be infinitely elastic. An example of a stamp folding that is not a semi-meander is shown in Figure 1(c). Observe that each labeled crossing represents a stamp. As shown in the Figure 1, each of these three objects can be represented by a permutation; however not all permutations represent even the most general of these objects - the stamp foldings. For example, the permutation 1423, would require the strip of stamps (or river) to intersect itself.

The focus of this paper is to develop efficient algorithms to exhaustively list stamp foldings, semi-meanders and open meanders of order n. For a history on combinatorial generation algorithms consult Knuth's recent addition to his series *The Art of Computer Programming* [5]. Stamp foldings were first discussed in [7, 11], and algorithms for generating stamp foldings were considered in Scott Lausch's Master's thesis [9]. An implementation of the latter algorithm is used by the "Combinatorial Object Server" at `http://www.theory.csc.uvic.ca/` in the permutation section. A FORTRAN algorithm to generate all semi-meanders is outlined in the appendix of [2], but no analysis is provided. An explicit algorithm for generating meanders has been given by Di Francesco et al [2] with complexity proportional to the Catalan numbers ($c_n \approx 4^n$). Franz and Earnshaw's [3] algorithm also appears to have an asymptotic running time that is greater than the number of meanders being generated (no implementation details or analysis is provided). The fastest known algorithm is given by Bobier and Sawada [1]. Although a rigorous analysis is not provided, the implementation of the algorithm is very simple.

For each of these three objects we can consider equivalences under various actions. For semi-meanders, if one end is uniquely determined to be allowed to wind inside the curve, we can consider equivalence under reversal to obtain *symmetric semi-meanders*. For stamp foldings, if we consider the stamps to be unlabeled without regard for the orientation of the stamps, then we obtain *unlabeled stamp foldings*. For example the permutation 1342 is equivalent to 4213 by relabeling the stamps. If we consider the reversal of each folding we may also obtain two different permutations; in this case we also obtain 2431 and 3124 as illustrated below.



Note that each equivalence class will consist of either 2 or 4 permutations. An example of a class that contains only two permutations is $\{1234, 4321\}$. Similarly we can obtain *symmetric meanders* by considering the same actions. Enumeration sequences for each of these 6 objects

are given in the table below for n up to 25. Each sequence is labeled with its corresponding sequence number in Sloane's Encyclopedia of Integer Sequences [10].

n	A000136 Stamp Foldings	A001011 Unlabled Stamps	A000682 Semi-meanders	A000560 Symmetric Semis	A005316 Open Meanders	A077055 Symmetric Meanders
1	1	1	1	1	1	1
2	2	1	2	1	1	1
3	6	2	4	2	2	1
4	16	5	10	5	3	2
5	50	14	24	12	8	3
6	144	38	66	33	14	8
7	462	120	174	87	42	13
8	1392	353	504	252	81	42
9	4536	1148	1406	703	262	72
10	14060	3527	4210	2105	538	273
11	46310	11622	12198	6099	1828	475
12	146376	36627	37378	18689	3926	1970
13	485914	121622	111278	55639	13820	3506
14	1557892	389560	346846	173423	30694	15368
15	5202690	1301140	1053874	526937	110954	27888
16	16861984	4215748	3328188	1664094	252939	126510
17	56579196	13976335	10274466	5137233	933458	233809
18	184940388	46235800	32786630	16393315	2172830	1086546
19	622945970	155741571	102511418	51255709	8152860	2039564
20	2050228360	512559185	329903058	164951529	19304190	9652364
21	6927964218	1732007938	1042277722	521138861	73424650	18360296
22	22930109884	5732533570	3377919260	1688959630	176343390	88172609
23	77692142980	19423092113	10765024432	5382512216	678390116	169610371
24	258360586368	64590165281	35095839848	17547919924	1649008456	824506191
25	877395996200	219349187968	112670468128	56335234064	6405031050	1601297937

The main results of this paper are as follows:

- ▷ A constant amortized time algorithm to generate stamp foldings,
- ▷ A constant amortized time algorithm to generate semi-meanders,
- ▷ The fastest known algorithm to generate open meanders.

Additionally, these algorithms can be modified to obtain:

- ▷ A $O(n)$ amortized time algorithm to generate unlabeled stamp foldings,
- ▷ A constant amortized time algorithm to generate symmetric semi-meanders,
- ▷ A $O(n)$ amortized time algorithm to generate symmetric open meanders.

For each algorithm, we use a permutation to represent each object as illustrated in Figure 1. An alternate permutation representation has been considered in [4]. The key to each algorithm is a special tree-like data structure whose nodes contain a pair of doubly linked lists. By focussing on a specific current node, we can determine all the valid intervals to extend the order of a given object in constant time. Once the order is extended, the data structure can also be updated in constant time. In Section 2 we begin by outlining this data structure for semi-meanders. Stamp foldings are slightly more complicated and are detailed in Section 2.2. Then in Section 3.2, by maintaining the *wind-factor* for semi-meanders we obtain an efficient algorithm to generate open-meanders. This algorithm is analyzed and compared experimentally with the previously fastest known algorithm to exhaustively list open-meanders [1]. The paper concludes with a summary in Section 4.

2 Generating semi-meanders and stamp foldings

In this section we begin by describing an algorithm to exhaustively list all semi-meanders and symmetric semi-meanders of order n, since they are the easiest to handle using the permutation representation. The key to making the algorithm run in constant amortized time is the maintenance of a tree of special nodes. Then, by applying a subtle tweak to this data structure, we outline a constant amortized time algorithm for stamp foldings.

2.1 Semi-meanders

The basic idea behind our algorithm is to extend a semi-meander of order t represented by a permutation P to a semi-meander of order $t+1$ by considering all *valid* intervals to extend the semi-meander curve. For example, Figure 2(a) below illustrates a semi-meander of order 9 along with its corresponding permutation representation. The valid intervals to extend the semi-meander through are (3,2), (1,9), (9,8) and (7,4) respectively. We consider the permutation to be prefixed with a '0' and suffixed with a '$-$', so that every interval has a clearly defined start and end value. From our example, this means that $(0, 3)$ would be the leftmost interval and $(4, -)$ would be the rightmost interval.

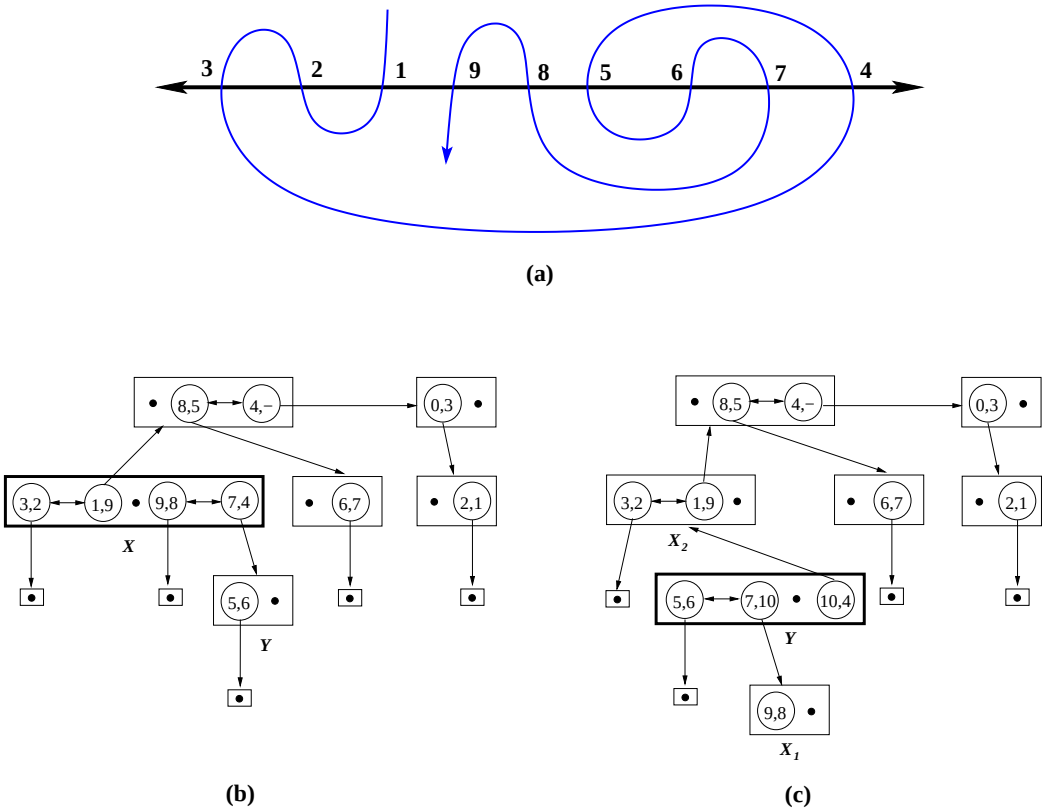


Figure 2: (a) A semi-meander of order 9 and its permutation representation. (b) The node tree for the semi-meander in (a). (c) The node tree obtained by extending the semi-meander in (a) to cross the interval (7,4).

```
procedure Gen (t)
    if (t > n) then Process(P)
    else
        LIST := list of valid intervals to extend the semi-meander
        for each interval I ∈ LIST do
            insert t into permutation P depending on I
            Gen(t+1)
            remove t from P
end
```

Figure 3: Algorithm Gen(t), to list semi-meanders of order n.

Given a permutation representing a semi-meander of order t, our goal is to efficiently determine which of the $t+1$ intervals can be used for the next crossing. If these intervals are available in a list, then we can use the simple algorithm Gen(t) in Figure 3 to generate all semi-meanders of order n. The permutation P is initialized to have one crossing and the initial call is Gen(2). For this first recursive call, the $LIST$ will consist of the two intervals $(0,1)$ and $(1,-)$. The permutation itself can be updated in constant time if it is represented as a doubly linked list with pointers to each element. The function Process(P) is a generic function that may perform some action on the current semi-meander P.

In order to efficiently obtain and update such a list of intervals, it is useful to split the list into two doubly linked lists L and R such that L (respectively R) contains all valid intervals to the left (right) of the current crossing. For example, if the current permutation P is 321985674 as illustrated in Figure 2(a), then $L = \langle(3,2),(1,9)\rangle$ and $R = \langle(9,8),(7,4)\rangle$. The lists are doubly linked so that the addition or deletion of an interval can be done in constant time. We call the data structure containing these two lists a *node*. If X is a node, then we let L_X denote its left list and let R_X denote its right list.

If we consider what happens when we extend a semi-meander by crossing through an interval I, then it becomes apparent that we need to know which intervals become valid in addition to the new intervals that have just been created. For example, if we extend the semi-meander in Figure 2(a) by crossing the interval (7,4) then in addition to the new intervals $(7,10)$ and $(10,4)$, the next crossing would also be able to cross interval (5,6) to the left but nothing else to the right. This leads to building a *tree of nodes* where each interval points to a unique node in the tree and where each interval appears in exactly one node. As an example, the tree of nodes correponding to the semi-meander in Figure 2(a) is shown in part (b) where the *current node* labeled X is in bold.

To incorporate the tree of nodes data structure to the basic algorithm Gen(t):

- pass the current node X as a parameter to each recusive call,
- set $LIST$ to the concatenation of L_X and R_X,
- let Y represent the node pointed to by the current interval $I = (i,j)$, and
- add a function Update(X,Y,I) to update the tree of nodes data structure and the permutation P.

The challenge that remains is how to efficiently implement the function Update(X, Y, I). Observe that as the interval I is crossed by the semi-meander, it will be replaced with 2 new intervals in Y: $I_1 = (i, t)$ and $I_2 = (t, j)$. It is not difficult to see that I_1 should be added to the end of L_Y and that I_2 will be inserted to the front of R_Y. Once I is removed from X, the remaining intervals of X get split into 2 nodes X_1 and X_2 such that X_1 contains the intervals accessible by crossing I_1 and X_2 contains the intervals accessible by crossing I_2. Once these nodes are created, we set I_1 to point to X_1 and I_2 to point to X_2. Precisely how the node X is split into X_1 and X_2 depends on whether I belongs to L_X or R_X. If the intervals in the list containing I are $i_1, i_2, \ldots, i_k$, where i_c denotes the interval I being crossed, then the following table describes how to construct X_1 and X_2:

$I \in L_X$	$I \in R_X$
$L_{X_1} = i_1, \ldots, i_{c-1}$ $R_{X_1} = R_X$	$L_{X_1} = i_1, \ldots, i_{c-1}$ $R_{X_1} =$ null
$L_{X_2} =$ null $R_{X_2} = i_{c+1}, \ldots, i_k$	$L_{X_2} = L_X$ $R_{X_2} = i_{c+1}, \ldots, i_k$

Each of these assignments can be implemented in constant time by maintaining pointers to the start and end of each interval list. To summarize, the function Update(X, Y, I) does the following:

- insert the new interval $I_1 = (i, t)$ to the end of L_Y,
- insert the new interval $I_2 = (t, j)$ to the front of R_Y,
- remove interval I from X,
- split X into two new nodes X_1 and X_2
- set I_1 to point to X_1 and set I_2 to point to X_2.

As an example of the steps involved in an update, Figure 2(c) shows the result of how the node tree from Figure 2(b) gets updated when the interval (7,4) (from R_X) is crossed. Observe in the figure that the new intervals added to Y point to nodes labeled X_1 and X_2.

By applying the tree of nodes data structure, the resulting algorithm GenSemi(t, X) is shown in Figure 4. The procedure Restore() undoes the changes made in Update(X, Y, I). Both functions can be implemented to run in constant time. To initialize the algorithm an initial node X is created with $L_X = \langle(0, 1)\rangle$ and $R_x = \langle(1, -)\rangle$. The initial intervals point to nodes with empty interval lists. The initial call is GenSemi$(2, X)$.

To analyze this algorithm, observe that each recursive call is the result of a constant amount of work. Thus, the total amount of work done by the algorithm is proportional to the number of recursive calls in the computation tree. Since the number of semi-meanders generated is equal to the number of leaves in the computation tree, an amortized analysis can be performed by considering the ratio of the total number of nodes in the computation tree to the number of leaves. If this ratio is bounded by a constant then the algorithm will run in constant amortized time, i.e., the total work done divided by the number of objects generated is bounded by a constant. Since there are no dead ends in this algorithm, every recursive call will lead to an

```
procedure Update(X, Y, I)
    insert new intervals I_1 := (i, t) and I_2 := (t, j) into Y
    remove I = (i, j) from X
    split X into X_1 and X_2
    point I_1 to X_1
    point I_2 to X_2
    update P
end

procedure GenSemi (t, X)
    if (t > n) then Process(P)
    else
        for each interval I = (i, j) ∈ L_X, R_X do
            Y := node pointed to by I
            Update(X, Y, I)
            GenSemi(t+1, Y)
            Restore()
end
```

Figure 4: Algorithm GenSemi(t, X), to list semi-meanders of order n.

semi-meander being generated. Also, there are always at least two possible ways to extend a semi-meander of order i to one of order $i + 1$: i.e., each internal node has at least 2 children. Thus, the number of leaves will be greater than the number of internal nodes which implies that the ratio is constant.

Theorem 1. *Semi-meanders of order n can be generated in constant amortized time.*

2.1.1 Symmetric semi-meanders

By considering reflective symmetry about the first crossing we obtain symmetric semi-meanders. To generate symmetric semi-meanders using the algorithm in GenSemi(t, X), we force the second crossing to be to the left (or equivalently to the right) of the first crossing. This can be implemented in constant time by skipping the intervals in the right list when $t = 2$. Alternatively, the semi-meander can be initialized to have two crossings.

Corollary 2. *Symmetric semi-meanders of order n can be generated in constant amortized time.*

2.2 Stamp Foldings

Recall that a stamp folding is a generalization of a semi-meander where both ends are allowed to wind inside the curve (river, strip of stamps). To generate stamp foldings, we can apply the semi-meander algorithm GenSemi(t, X) with a slight change to the data structures. In

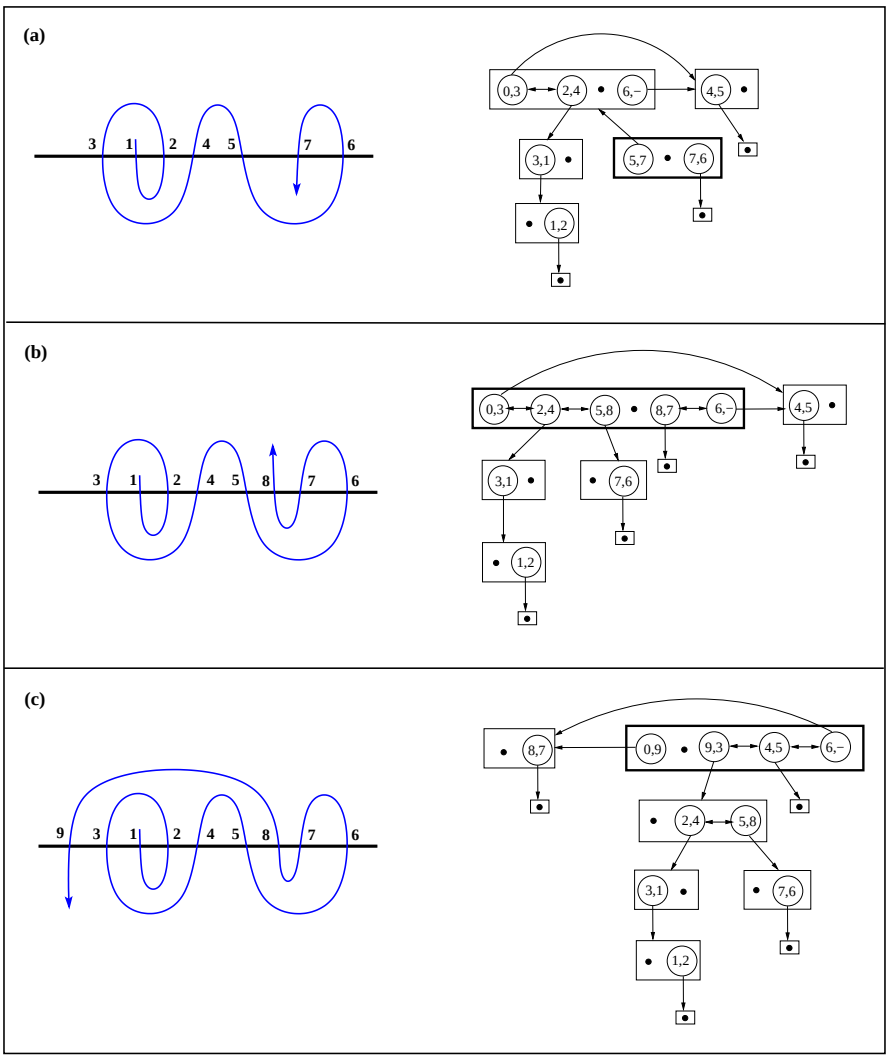


Figure 5: (a) A stamp folding of order 7 and its corresponding data structure representation. (b) The changes after extending the stamp folding in (a) by crossing (5,7). (c) The changes after extending the stamp folding in (b) by crossing (0,3).

particular, for stamp foldings the two intervals $(0, x)$ and $(y, -)$ will always belong to the same node X and will both point to the same node Y. This means we no longer have a *tree* of nodes, which makes the algorithm slightly more complicated when one of the intervals $(0, x)$ or $(y, -)$ is crossed. As an illustration, the node structures for a series of stamp foldings in Figure 5.

Since an interval $I = (0, x)$ is special, consider what happens just before such an interval is crossed (a similar analysis applies to $(y, -)$). Assume that the interval I points to the node Y. In the case of semi-meanders, all valid intervals will be in the right list R_Y and the left list L_Y will be empty. However, for stamp-foldings, if Y was created by crossing through an interval of the form $(y, -)$ then all available intervals will be in the left list L_Y and the right list R_Y will be empty. Since there are pointers to the front of each list, this can be checked in constant time. Thus after we cross I, if the left list L_Y is non-empty then we set $R_Y = L_Y$ and set L_Y to be

empty. The only remaining modification is to move the interval of the form $(y, -)$ from X to the end of R_Y. This can also be done in constant time since we maintain pointers to the end of each list.

In order to convert the algorithm GenSemi(t, X) into one that generates all stamp foldings, the following operations must be added to the function Update(X, Y, I):

- If crossing an interval I in node X of the form $(0, x)$ then
  - ▷ if L_Y is not empty assign $R_Y = L_Y$ and set L_Y to be empty,
  - ▷ move the interval $(y, -)$ which is the last interval in R_X to the end of R_Y and point it to node X_1.
- If crossing an interval I in node X of the form $(y, -)$ then
  - ▷ if R_Y is not empty assign $L_Y = R_Y$ and set R_Y to be empty,
  - ▷ move the interval $(0, x)$ which is the first interval in L_X to the front of L_Y and point it to node X_2.

The function Restore() must undo these operations.

The analysis for stamp foldings is the same as for semi-meanders since at each recursive call there are at least two ways to extend the current permutation.

Theorem 3. *Stamp foldings of order n can be generated in constant amortized time.*

2.2.1 Unlabeled stamp foldings

Recall from Section 1 that if we consider the stamps to be unlabeled and disregard the orientation of each stamp folding, we obtain an unlabeled stamp folding. Each equivalence class has at most 4 permutations and we let the lexicographically smallest permutation be the canonical representative. Unfortunately, to determine whether a given permutation is in canonical form is not a trivial matter like it was for symmetric semi-meanders. Thus it remains an open problem to generate unlabeled stamp foldings in constant amortized time. By performing a simple linear time check on each permutation against the 4 possible symmetries we obtain the following theorem.

Theorem 4. *Unlabeled stamp foldings of order n can be generated in $O(n)$ amortized time.*

3 The wind-factor and open meanders

The *wind-factor* [2] of a semi-meander is the smallest number of additional crossings required to extend the semi-meander into an open meander. Thus, meanders are precisely the semi-meanders with wind-factor 0. For example, the wind-factor of the semi-meander in Figure 2(a) is 1 and it also corresponds to the depth of the current node in its tree of nodes. As another example, the wind-facor of the semi-meander in Figure 1(b) is 2. We consider the wind-factor

for two reasons. First it may be of interest to list all semi-meanders with a given wind-factor. Second, it is important to maintain if we want to efficiently modify the semi-meander algorithm to generate all open meanders. We begin this section by outlining how to modify the semi-meander algorithm GenSemi(t, X) so it maintains the wind-factor. Then we discuss how it can be applied to efficiently generate meanders.

3.1 Maintaining the wind-factor

In order to generate all semi-meanders with a given wind-factor w, we must maintain the current wind-factor at each step of the algorithm GenSemi(t, X). In order to efficiently maintain this information, we need to know the unique interval that can be crossed to reduce the wind-factor if $w > 0$ or to maintain the wind-factor if $w = 0$. We call such an interval the *unwinding interval* and for a node X we denote its unwinding interval by U_X. If $w > 0$, then every node in the path up the tree from the current node will have an unwinding interval associated with it. For example, in Figure 2(a-b), the unwinding interval for the current node labeled X is (1,9). The node pointed to by this interval will have wind-factor $w = 0$ and if it becomes the current node, the unique interval that maintains the wind-factor is $(4, -)$.

If we add a pointer to the unwinding interval in the node data structure, then we can determine if a given interval corresponds to the unwinding interval in constant time. To efficiently maintain the unwinding intervals it is important to know which list it is in: the Left or the Right. For a given node X let $S_X \in \{\text{L,R}\}$ denote the list that the unwinding interval belongs to. If we add the current wind-factor w as a parameter to each recursive call, then we can maintain the wind-factor in constant time as follows: if the current interval corresponds to the unwinding interval then we decrement the wind-factor if $w > 0$ and leave it unchanged if $w = 0$; otherwise we increment the wind-factor.

The only challenge that remains is to update the unwinding intervals when a new crossing is added. To simplify the discussion, let the function SetUnwind(X, I, s) set the unwinding interval $U_X = I$ and its corresponding list $S_X = s$. There are several cases to consider; however, by maintaining a boolean to remember if the unwinding interval has been visited when iterating through the interval lists, each case can be performed in constant time and added to the function Update(X, Y, I) as follows:

- If $I = U_X$ when $w > 0$: no update required.
- If $I = U_X$ when $w = 0$: call SetUnwind(Y, I_2, R).
- If $I \neq U_X$ and $I \in L_X$:
  - ▷ If $U_X \in L_X$ and comes after I call SetUnwind(Y, I_2, R) & SetUnwind(X_2, U_X, R).
  - ▷ Otherwise call SetUnwind(Y, I_1, L) & SetUnwind(X_1, U_X, S_X).
- If $I \neq U_X$ and $I \in R_X$:
  - ▷ If $U_X \in R_X$ and comes after I call SetUnwind(Y, I_2, R) & SetUnwind(X_2, U_X, S_X).
  - ▷ Otherwise call SetUnwind(Y, I_1, L) & SetUnwind(X_1, U_X, L).

The correctness of these updates can easily be observed by considering a few sample semi-meanders like the one in Figure 2(a). Applying these extra operations allows us to generate all semi-meanders of order n with a given wind-factor w. As a summary, to maintain the wind-factor efficiently the node data structure is as follows:

- L_X: a doubly linked list of valid intervals to the left of the current crossing ordered from left to right,
- R_X: a doubly linked list of valid intervals to the right of the current crossing ordered from left to right,
- U_X: a pointer to the unwinding interval if it exists,
- $S_X \in \{\text{L,R}\}$: a character indicating which list the unwinding interval is in if it exists.

3.2 Generating Open Meanders

To generate open meanders, we can simply apply the semi-meander algorithm that maintains the wind-factor and then output only those semi-meanders with wind-factor 0. Such an algorithm would be far from efficient since it effectively generates all semi-meanders. However by applying the following optimization we obtain a much more efficient algorithm. The basic idea is to consider a semi-meander with wind-factor w and order $n - w$. For such a semi-meander there there is no point in winding any further, since it will never produce an open meander of order n. Thus, in this situation, we only produce a recursive call for the interval corresponding to the unwinding interval. Specifically, the optimization is as follows:

Observation 5. *If w is the wind-factor of a semi-meander of order $t-1$ and $n-t \leqslant w$, then the only way the semi-meander can be extended into an open meander of order n is by unwinding.*

Pseudocode that applies this optimization to generate open meanders is given by GenMeander(t, X, w) shown in Figure 6. The initialization is the same as with semi-meanders with the wind-factor w initially set to 0. It is assumed that the function Update(X, Y, I) updates the unwinding intervals as outlined in the previous subsection and that the function Restore undoes this action.

The analysis for algorithm GenMeander(t, X, w) is a challenge because of the introduction of degree one nodes in the computation tree when applying the optimization. Each such degree one node will correspond to a semi-meander of order $n - i$ with a wind-factor of i for some $i > 0$. Let $Comp(n)$ denote the number of nodes in the computation tree to generate open meanders of order n. We partition the computation tree into sets of nodes based on the order of the node and the wind-factor. Let $S(i, j)$ denote the number of semi-meanders or order i with a wind-factor of j. Since the wind-factor of a node with order i will never exceed $n - i$ (from the

```
procedure GenMeander (t, X, w)
    if (t > n) then Process(P)
    else
        if (n − t ⩽ w) then
            Y := node pointed to by U_X
            Update(X, Y, U_X)
            GenMeander(t+1, Y, w − 1)
            Restore()
        else
            for each interval I ∈ L_X, R_X do
                Y := node pointed to by I
                Update(X, Y, I)
                if (I = U_X) then GenMeander(t+1, Y, max(0, w − 1))
                else GenMeander(t+1, Y, w + 1)
                Restore()
end
```

Figure 6: Algorithm GenMeadner(t, X, w), to list open meanders of order n.

algorithm's optimization), we obtain the following expression for $Comp(n)$:

$$Comp(n) = \sum_{i=1}^{n} \sum_{j=0}^{min(i-1,n-i)} S(i,j).$$

As an illustration, we consider $Comp(7)$:

$$\begin{aligned} Comp(7) \;=\; & S(1,0) + \\ & S(2,0) + S(2,1) + \\ & S(3,0) + S(3,1) + S(3,2) + \\ & S(4,0) + S(4,1) + S(4,2) + S(4,3) + \\ & S(5,0) + S(5,1) + S(5,2) + \\ & S(6,0) + S(6,1) + \\ & S(7,0). \end{aligned}$$

Note that $S(i,0)$ counts the number of open meanders of order i. To prove that the generation algorithm for open meanders runs in constant amortized time we must show that there exists some constant c such that

$$\frac{Comp(n)}{S(n,0)} \leqslant c.$$

Empirically, for n up to 27 this ratio for the algorithm GenMeander(t, X, w) is given in the following table:

n	$\frac{Comp(n)}{S(n,0)}$	n	$\frac{Comp(n)}{S(n,0)}$
4	3.00000	5	2.87500
6	3.14286	7	2.92857
8	3.13580	9	2.94275
10	3.13197	11	2.96007
12	3.13831	13	2.97923
14	3.14882	15	2.99745
16	3.16008	17	3.01381
18	3.17084	19	3.02824
20	3.18072	21	3.04092
22	3.18965	23	3.05207
24	3.19769	25	3.06194
26	3.20492	27	3.07070

Even though the ratio is growing, it does not rule out the possibility that it is bounded by a constant. What is required is the ability to bound $S(i, j)$ recursively.

Lemma 6. *For $i > 1$:*

(a) $S(i, 0) = S(i - 1, 0) + S(i - 1, 1)$,
(b) $S(i, j) \geqslant S(i - 1, j + 1) + S(i - 1, j - 1)$ *for* $j > 0$.

PROOF: For (a) consider the first $i-1$ crossings for any semi-meander of order i and wind-factor 0. Either the first $i-1$ crossings will have wind factor 1 or 0. In either case, there is exactly one way to extend such semi-meanders into ones with wind-factor 0. For part (b), observe that each semi-meander of order $i-1$ and wind factor $j+1$ can be extended uniquely into a semi-meander of order i and wind factor j (via the unwinding interval). For a semi-meander of order $i-1$ and wind factor $j-1$, there may be many ways to extend it into one with wind factor j by adding one more crossing, thus giving the simple bound. □

The second bound for $j > 0$ can actually be improved to

$$S(i, j) \geqslant S(i - 1, j + 1) + \sum_{k=0}^{\lfloor i/2 \rfloor - 1} S(i - 1 - 2k, j - 1)$$

by considering unique extensions of semi-meanders of order $i - 1 - 2k$ with wind factor $j - 1$ into semi-meanders of order i and wind factor j. Unfortunately, even tighter bounds seem to be required to prove the conjecture that the generation algorithm for open meanders runs in constant amortized time. Since we do not have a proof of such a claim, we prove the very loose upper bound of a $O(n)$ amortized time algorithm.

By considering the diagonals of $Comp(n)$ moving from the bottom left to the top right, we can re-express $Comp(n)$ as follows:

$$Comp(n) = \sum_{i=1}^{n} \sum_{j=0}^{\lceil \frac{i}{2} \rceil - 1} S(i-j, j).$$

Since $S(i,j) > S(i-1, j+1)$ we get the bound:

$$\begin{aligned} Comp(n) &\leqslant \sum_{i=1}^{n} i \cdot S(i,0) \\ &\leqslant n \cdot \sum_{i=1}^{n} S(i,0) \\ &\leqslant cn \cdot S(n,0), \end{aligned}$$

where c is a constant since open meanders grow exponentially.

Theorem 7. *Open meanders of order n can be generated in $O(n)$ amortized time.*

Using a 2.2 GHz Opteron processor, Table 1 compares the running time of our algorithm GenMeander(t, X, w) for open meanders with the fastest previously known algorithm from [1]. Observe that for $n = 29$ that our new algorithm finds all open meander in about 31.8 hours compared to 46.1 hours for the algorithm in [1].

n	GenMeander(t, X, w)	Algorithm from [1]
20	4	5
21	14	20
22	35	43
23	127	186
24	325	430
25	1214	1877
26	3139	4022
27	11700	16575
28	30480	38358
29	114414	165987

Table 1: Comparison of running time in seconds for two open meander generation algorithms.

3.2.1 Symmetric Open Meanders

Recall from Section 1 that if we consider the equivalence classes of open meanders under the operations of relabeling and reversal, we obtain symmetric meanders. If we let the lexicographically smallest element be the canonical representative of each equivalence class then the following tests can be performed on each generated open meander to determine if it corresponds to its canonical representative:

- make sure that 1 appears before n in the permutation, and
- test that the permutation is lexicographically smaller than its relabeled reversal: each value i from the original permutation is replaced with $n-i+1$ and the result is considered in reverse.

These tests can easily be performed in $O(n)$ time after each open meander has been generated. Thus from Theorem 7 we obtain the following result.

Corollary 8. *Symmetric open meanders of order n can be generated in $O(n)$ amortized time.*

4 Summary

In this paper we have constructed a new data structure representation for semi-meanders, meanders and stamp foldings and applied the data structure to develop efficient algorithms to exhaustively list:

- semi-meanders and symmetric semi-meanders in $O(1)$ amortized time,
- stamp foldings in $O(1)$ amortized time,
- unlabeled stamp foldings in $O(n)$ amortized time.
- open meanders in $O(n)$ amortized time, and
- symmetric open meanders in $O(n)$ amortized time,

The algorithms have been implemented in C and are available for download at:
http://www.socs.uoguelph.ca/~sawada/programs.html.

It remains an open problem to determine whether or not the meander algorithm provided in this paper runs in constant amortized time. This can be answered if the right recursive bounds can placed on semi-meanders with a given wind-factor. Another open problem is to improve the running time for unlabeled stamp foldings. Finally, does there exist a Gray code for any of these objects?

References

[1] B. Bobier, J. Sawada, *A fast algorithm to generate open meandric sequences and meanders*, Transactions on Algorithms, Vol. 6 No. 2 (2010) 12 pages.

[2] P. Di Francesco, O. Golinelli and E. Guitter, *Meanders: a direct enumeration approach*, Nuc. Phys. B 482, (1996), pp. 497-535.

[3] R. Franz and B. Earnshaw, *A constructive enumeration of meanders*, Annals of Combinatorics 6:(1) (2002), pp. 7-17.

[4] K. Hoffmann, K. Mehlhorn, P. Rosenstiehl, and R. Tarjan, *Sorting Jordan sequences in linear time using level-linked search trees*, Information and Control, 68 (1986), pp. 170-184.

[5] D. E. Knuth, The Art of Computer Programming, Volume 4: Generating All Trees; History of Combinationatorial Generation,, Fascicle 4, Addison-Wesley, February 2006, 150 pages.

[6] K.H. Ko and L. Smolisky, *A combinatorial matrix in 3-manifold theory*, Pacific J. Math. 149 (1991) pp. 3190336.

[7] J. E. Koehler, *Folding a strip of stamps*, Journal of Combinatorial Theory, 5 (1968), pp. 135-152.

[8] S. K. Lando and A. K. Zvonkin, *Plane and projective meanders*, Theoretical Computer Science 11:(2) (1993), pp. 117-144.

[9] S. Lausch, *Generating Some Restricted Classes of Permutations*, Master's Thesis, University of Victoria, Canada, 1999.

[10] N. Sloane, *The on-line encyclopedia of integer sequences,* http://oeis.org/, IDs: A000136, A001011, A000682, A000560, A005316, A077055, (2009).

[11] J. Touchard, *Contributions à l'étude du problème des timbres postes*, Canad. J. Math., 2 (1950), pp. 385-398.