A Fast Algorithm To Generate Open Meandric Systems and Meanders

BRUCE BOBIER
University of Waterloo
and
JOE SAWADA
University of Guelph

An open meandric system is a planar configuration of acyclic curves crossing an infinite horizontal line in the plane such that the curves may extend in both horizontal directions. We present a fast, recursive algorithm to exhaustively generate open meandric systems with n crossings. We then illustrate how to modify the algorithm to generate unidirectional open meandric systems (the curves extend only to the right) and non-isomorphic open meandric systems where equivalence is taken under horizontal reflection. Each algorithm can be modified to generate systems with exactly k curves. In the unidirectional case when k=1, we can apply a minor modification along with some additional optimization steps to yield the first fast and simple algorithm to generate open meanders.

Categories and Subject Descriptors: G.2.1 [**Combinatorics**]: Combinatorial algorithms

General Terms: Algorithms

Additional Key Words and Phrases: CAT algorithm, meander, open meandric system

1. INTRODUCTION

An open *meander* can be thought of as the system formed by an infinite river running from northwest to east which passes beneath n bridges of an infinite straight road going from west to east. It is from this geographical analogy, shown in Fig. 1, that the name"meander" is derived.

Meanders have been studied in various contexts, including the map-folding problem [13], the stamp-folding problem [4; 9; 18], polymer chains [5] and in relation to Jordan curves [3; 16], with literature dating back to Poincaré's work on differential geometry [15]. Meanders are also related to simple alternating transit mazes of depth n [14], and to ovals of planar algebraic curves (Hilberts 16^{th} problem) [2]. The problem of enumerating meanders is known to be a difficult problem and a significant body of work has been dedicated to it [1; 4; 7; 8; 11; 12]. However, until now, no algorithms have been developed that focus on the efficient generation of

Author's email: Bruce Bobier (bbobier@engmail.waterloo.ca), Joe Sawada (jsawada@uoguelph.ca)
Research supported by NSERC.

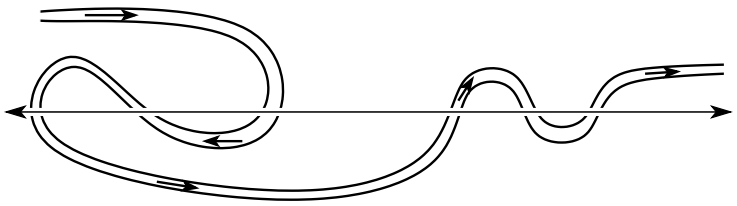


Fig. 1. Geographic analogy of a road with 6 bridges and a meandering river.

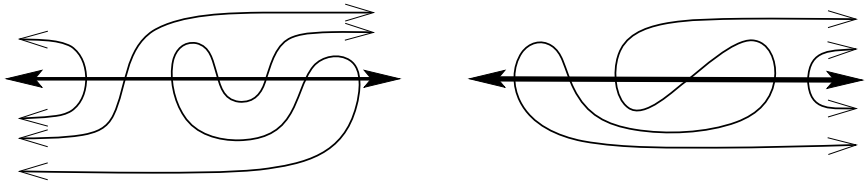


Fig. 2. a) Example of an open meandric system with 7 crossings. b) Example of a unidirectional open meandric system with 6 crossings.

meanders.

Open meandric systems are a generalization of open meanders obtained by allowing *multiple* unclosed curves extending in one or both horizontal directions. They should not be confused with "systems of meanders" which refer to systems of non-intersecting *closed* curves. Formally, an *open meandric system* is a planar configuration of acyclic curves crossing an infinite horizontal line in the plane such that the curves may extend infinitely in both horizontal directions. An example of an open meandric system with 7 crossings (and 3 curves) is shown in Fig. 2a. A *unidirectional open meandric system* is similar to an open meandric system, except that all curves must extend to the right (Fig. 2b). The study of (unidirectional) open meandric systems was originally motivated by the discussion of non-crossing matchings and sphere cut branch decompositions in [6]. It turns out that these systems, and in particular, their enumeration sequences, have also been studied rather extensively by Bacher [3] as they relate to a graded algebra of meander slices.

In this paper we develop Constant Amortized Time (CAT) algorithms (i.e., the amount of computation time is proportional to the number of generated objects) which generate:

—all open meandric systems with n crossings,

—all unidirectional open meandric systems with n crossings, and

—all (unidirectional) open meanderic systems with n crossings and equivalence under reflection.

Additionally, each algorithm can easily be modified to list all such systems with n crossings having exactly k curves. In the unidirectional case when k=1, we can apply a minor modification along with simple optimizations to yield the first fast

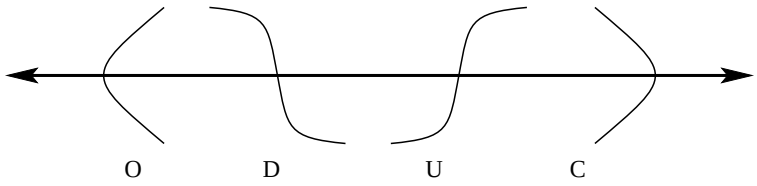


Fig. 3. The alphabet for crossings.

and simple algorithm for generating open meanders.

2. GENERATING OPEN MEANDRIC SYSTEMS

In this section, we develop a fast algorithm for generating (unidirectional) open meandric systems with n crossings. Before we do this, however, we first introduce a simple string representation for these systems.

Let each point where a curve intersects the line be called a *crossing*. Then each crossing can be represented by one of four characters, as adopted from [10]: {O,D,U,C} (O=Open, D=Down, U=Up, C=Close).

Fig. 3 illustrates how the characters correspond to pieces of each curve. As reference, some other notations that have been used to describe similar systems include:

—{D,R,L,U} to describe meanders situated on a vertical line [1],

—$\{(, \backslash, /,)\}$ in [3], and

—{a, b, c, d} in [12].

For the remainder of this paper we will represent each open meandric system with n crossings as a word (string) of length n over the alphabet {O,D,U,C}.

2.1 A Simple Algorithm

Using the string representation we can generate all open meandric systems by applying a standard recursive approach that builds up the strings one character at a time. The only constraint when adding a new character to the end of a string is that it cannot create a cycle. Observe that this is only possible only when adding the character C. For example the words OC and OUDC do not correspond to valid open meandric systems since they form cycles (closed curves).

Pseudocode for such an algorithm to exhaustively list all open meandric systems with n crossings is given in Algorithm 1, where the global array *word* is used to maintain the current string representation. The function `Cycle` can be implemented to run in $O(n)$ time by tracing back through the string. It returns true only if the addition of a C to the current string creates a cycle. The function `Print` prints out the contents of *word*. The initial call is `SimpleGen`(1).

Algorithm 1: `SimpleGen(`t`)`

```
procedure SimpleGen( t : integer)
if t > n then Print();
else
    word[t] := 'O';    SimpleGen(t + 1);
    word[t] := 'D';    SimpleGen(t + 1);
    word[t] := 'U';    SimpleGen(t + 1);
    word[t] := 'C';    if not Cycle(word) then SimpleGen(t + 1);
```

Since every internal node in the computation tree of `SimpleGen(`t`)` has at least 2 children, the overall running time of this algorithm will be proportional to the number of leaves (the open meandric systems) times the amount of time required for each recursive call. Thus, the algorithm will run in $O(n)$ amortized time. In the next section we introduce some data structures that allow us to perform the cycle check in constant time.

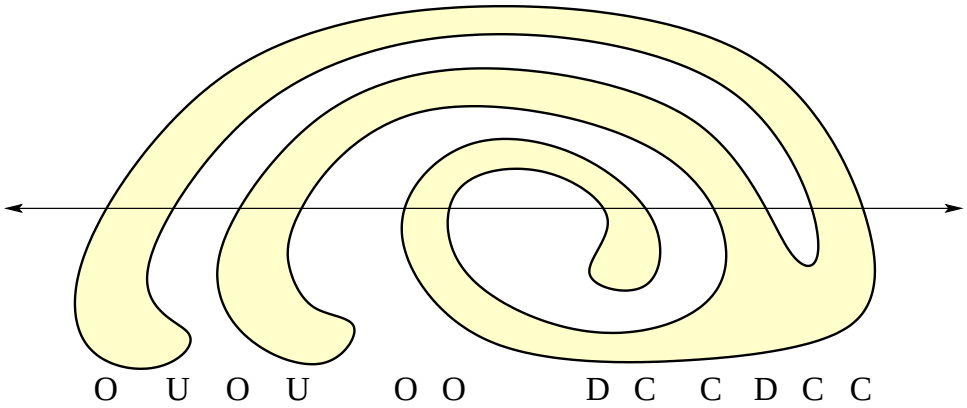
2.2 A Fast Algorithm

In this subsection, we introduce some extra data structures that allow us to perform the cycle test in constant time. The main idea is to keep track of the endpoints of each curve in the open meandric system so that when we add a new character, we can determine which curves, if any, it will attach to. To do this, we use two stacks *top* and *bot* to store the endpoints of available curves that extend "to the right": one for the top of the line, and one for the bottom. If a character creates a new curve, we initially label the curve with its position in the sequence.

Observe that anytime we add an O, we will always push its position onto both stacks. When adding a D (or symmetrically a U) and the top stack is non-empty, we push the new position only onto the bottom stack since the other endpoint extends infinitely to the left. If the top stack is non-empty, the new crossing will attach itself to an existing curve for which we pop an endpoint off the top stack and push it onto the bottom stack. The difficult case is when we attempt to add a C. If both stacks are empty, we do nothing. If only one stack is non-empty, we simply pop an endpoint off the non-empty stack. If both stacks are non-empty, then after popping endpoints of curves off of both stacks, one of two things can happen. If the popped endpoints belong to the same curve, then adding a C will form a cycle. If they belong to different curves, then we end up joining two curves together. The tricky part is how to join the curves together since the other endpoints of each curve may still be in the stack.

To handle this case, we first considered using the position of the first crossing to represent each curve. However, for some cases this strategy still requires a linear amount of work. Instead, we introduce a mapping f of the endpoints for each curve. Initially, the mapping for each curve t is itself: $f(t) = t$. Then, when two curves get joined together, the unjoined endpoints get mapped to each other.

In Fig. 4 we illustrate the data structures for the word OUOUOODCCDCC. Initially the stacks are empty and $f[t] = t$ for $t = 1$ to n. The stacks are ordered so that the top of each stack is the left of the list.



t	$word$	top	bot	f
0	-	-	-	$f[t] := t$
1	O	1	1	-
2	U	1,1	-	-
3	O	3,1,1	3	-
4	U	3,3,1,1	-	-
5	O	5,3,3,1,1	5	-
6	O	7,5,3,3,1,1	7,5	-
7	D	5,3,3,1,1	7,7,5	-
8	C	3,3,1,1	7,5	$f[5] := 7$, $f[7] := 5$
9	C	3,1,1	5	$f[5] := 3$, $f[3] := 5$
10	D	1,1	3,5	-
11	C	1	5	$f[1] := 5$, $f[5] := 1$
12	C	-	-	**cycle**

Fig. 4. Illustrating the data structures for the word OUOUOODCCDCC .

Observe when we add the U at t=2 that the endpoint 1 gets popped off of the bottom stack and added to the top stack. When we add the C at $t = 8$, we are joining the two curves 5 and 7 together. Since $f[5] = 5$ and $f[7] = 7$, we match up their other endpoints by setting $f[5] := 7$ and $f[7] := 5$. Then at $t = 9$ (the tricky part of the example), we are joining the endpoints 3 and 7 together. Since $f[3] = 3$ and $f[7] = 5$ we must map their corresponding endpoints 3 and 5 together: $f[3] := 5$ and $f[5] := 3$. When we add the C at $t = 11$ we pop 1 and 3 from the stacks. Since $f[1] = 1$ and $f[3] = 5$ we map the endpoints 1 and 5 together: $f[1] := 5$ and $f[5] := 1$. Finally, at $t = 12$, we pop 1 and 5 off of the stacks. Now since $f[1] = 5$, we know that we are joining the endpoints of the same curve together and are creating a cycle. Thus, the cycle detection can be performed in constant time.

Pseudocode which applies these data structures to generate open meandric systems is shown in Algorithm 2. Since the cycle detection can be performed in constant time, the running time of the algorithm will be proportional to the size of the computation tree (the number of recursive calls). Further, since each node in this tree has at least 2 children and each leaf in the tree corresponds to a unique open meandric system, we arrive at the following theorem:

Algorithm 2: `Gen(`t`)`

```
procedure Gen( t : integer)
local x, y, a, b, a2, b2: integer;
  if t > n then Print();
  else
      word[t] := 'O';
      push(top, t);      push(bot, t);
      Gen(t + 1);
      pop(top);      pop(bot);

      word[t] := 'D';
      if notEmpty(top) then
          x : = pop(top);      push(bot, x);
          Gen(t + 1);
          x : = pop(bot);      push(top, x);
      else
          push(bot, t);
          Gen(t + 1);
          pop(bot);

      word[t] := 'U';
      if notEmpty(bot) then
          x : = pop(bot);      push(top, x);
          Gen(t + 1);
          x : = pop(top);      push(bot, x);
      else
          push(top, t);
          Gen(t + 1);
          pop(top);

      word[t] := 'C';
      if notEmpty(top) and notEmpty(bot) then
          x := pop(top);      y := pop(bot);
          a := f[x];      b := f[y];
          a2 := f[a];      b2 := f[b];
          if a ≠ y then
              f[b] := a;      f[a] := b;
              Gen(t + 1);
              f[b] := b2;      f[a] := a2;
          push(top, x);      push(bot, y);
      else if notEmpty(top) then
          x := pop(top);
          Gen(t + 1);
          push(top, x);
      else if notEmpty(bot) then
          x := pop(bot);
          Gen(t + 1);
          push(bot, x);
      else
          Gen(t + 1);
```

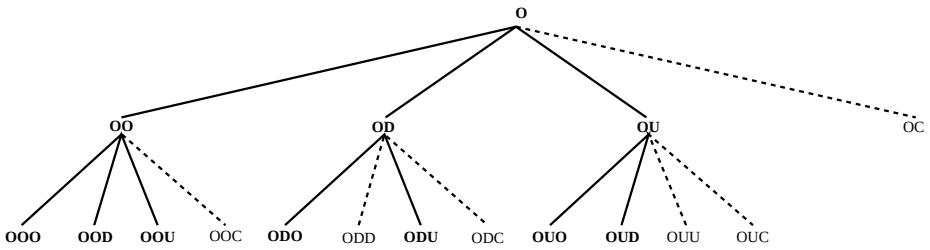


Fig. 5. Computation tree for unidirectional open meandric systems.

THEOREM 2.1. *The algorithm* `Gen(t)` *for exhaustively listing all open meandric systems with n crossings is CAT.*

To generate all open meandric systems with n crossings and exactly k curves, we can add a simple constant time test to the `Print` function that checks if the number of elements in the two stacks *top* and *bot* sums to $2k$.

2.3 Unidirectional Open Meandric Systems

In this subsection, we describe the modifications required to convert `Gen(`t`)` so that it only generates unidirectional open meandric systems.

From our definition of unidirectional open meandric systems, we know that all curves must extend to the right, meaning that the words first character can only be an O. Also, we can never add a D or C if the top stack is empty since it would result in a curve extending to the left. Similarly we cannot add a U or a C if the bottom stack is empty. To enforce these constraints, we can simply remove the **else** statements marked by a vertical bar in `Gen(`t`)` of Algorithm 2 and make the same initial call of `Gen(1)` .

Fig. 5 illustrates the computation tree for $n = 3$ that results from applying these modifications to `Gen(`t`)`. The dashed lines indicate nodes that will not be generated since they either create a cycle or have a curve that extends to the left. Notice that at each successive level in this tree we are generating all words for n at depth t, and that the leaf nodes in bold represent the words produced. Observe that at every node we can always add an O and at least one of D or U. Thus each internal node in the computation tree will still have at least two children.

THEOREM 2.2. *Unidirectional open meandric systems with n crossings can be generated in constant amortized time with a modified version of* `Gen(t)`*.*

2.4 Equivalence Under Horizontal Reflection

In this section we consider equivalence under reflection about the horizontal line. When we reflect a string representing a (unidirectional) open meandric system the O and the C remain invariant while each U becomes a D and vise-versa. For example, the string CODDUC is isomorphic to COUUDC under reflection (see Fig. 6).

It is easy to see that the strings that are self-equivalent under reflection are those that do not contain any Ds or Us. These are precisely the strings composed of i Cs followed by $n - i$ Os for $i = 0$ to n. For unidirectional open meandric systems,

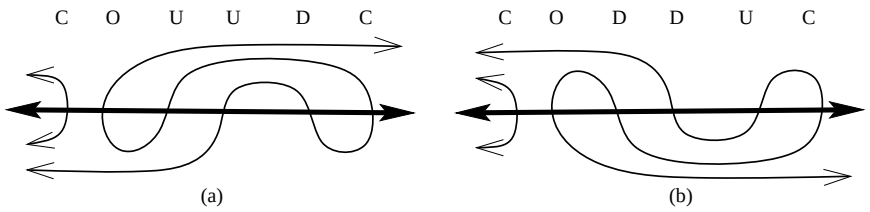


Fig. 6. Two open meandric systems that are equivalent under reflection.

only the sequence of all Os is equivalent to itself under reflection.

Since we only want to generate one string from each equivalence class, we arbitrarily let the string that has a D preceding all Us represent each class that contains two strings. The algorithm `Gen(t)` can easily be adapted to handle this by maintaining an additional parameter that indicates whether or not a D has been added to the current string. If it has not been added, then we do not consider the case where we add a U.

THEOREM 2.3. *(Unidirectional) open meandric systems with n crossings and equivalence under horizontal reflection can be generated in constant amortized time with a modified version of* `Gen(t)`.

We have also considered equivalence under reversal and 180 degree rotation (reflection and reversal). These equivalence classes can also be generated in constant amortized time in a fairly straightforward manner, but we omit the details.

2.5 Enumeration Sequences

Let S_n denote the number of open meandric systems with n crossings and let U_n denote the number of unidirectional systems with n crossings. Similarly, let S'_n and U'_n denote the corresponding systems with equivalence under reflection. In earlier discussion we demonstrated that the number of strings in S_n and U_n that are invariant under reflection are $n+1$ and 1 respectively. Thus, we get the following theorem:

THEOREM 2.4. *For $n \geq 0$,*

$$\begin{aligned} S'_n &= (S_n + n + 1)/2, \\ U'_n &= (U_n + 1)/2. \end{aligned}$$

In [3], the enumeration sequences for S_n and U_n are given up to n=26 and n=19 respectively. We extended these sequences by distributing our algorithm using SHARCNET[1]. The computation of S_{28} took less than a month using 25 processors while taking advantage of the equivalence under reflection. The results are in Table I. The sequences for S_n and U_n correspond to sequences A060111 and A060089 respectively in Sloans Encyclopedia of Integer Sequences [17].

[1]SHARCNET is a consortium of colleges and universities in a cluster of clusters of high performance computers, linked by advanced fibre optics (`http://www.sharcnet.ca`).

n	S_n	U_n
0	1	1
1	4	1
2	15	3
3	56	7
4	207	23
5	764	63
6	2805	213
7	10288	627
8	37609	2149
9	137380	6597
10	500655	22787
11	1823440	71883
12	6629423	249523
13	24090332	802291
14	87418221	2794365
15	317085352	9111917
16	1148825185	31814061
17	4160744164	104862813
18	15054719697	366796437
19	54454345624	1219313185
20	196805925995	4271041447
21	711077858188	14295561451
22	2567375653681	50131159253
23	9267176552040	168742700865
24	33430012251123	592279599483
25	120565130387572	2003050663889
26	434578910451203	7035894016347
27	1566103257814584	23890177457535
28	5641039781305999	83968962295531

Table I. The first terms of the sequences S_n and U_n.

3. GENERATING MEANDERS

Earlier we described an *open meander* as a river that starts in the northwest and flows east passing beneath n bridges of an infinite road going from west to east. Typically, a (closed) *meander* of order n refers to a self-avoiding closed curve that crosses a line in the plane at exactly $2n$ locations. There is a 1-1 correspondence between open meanders of order $2n-1$ and (closed) meanders of order n [12]. Table II shows the sequence of meandric numbers (open meanders) of order n. It corresponds to sequence A005316 in Sloanes Encyclopedia of Integer Sequences [17]. The odd numbered terms correspond to sequence A005315: the closed meandric numbers.

Open meanders are closely related to unidirectional open meandric systems with exactly one curve ($k = 1$). The only extra restriction required is that one of the endpoints of the single curve starts above the horizontal line, which represents the endpoint of the open meander that must start in the northwest. Thus, to generate open meanders, we can simply modify the `Print` function in the unidirectional pseudocode to test that $k = 1$ and that the stack top is not empty. Both tests can be performed in constant time. (*If desired, to change the direction of the first endpoint in the stack top from east to west, the index of the character for each*

n	Open meanders	n	Open meanders
0	1	13	13820
1	1	14	30694
2	1	15	110954
3	2	16	252939
4	3	17	933458
5	8	18	2172830
6	14	19	8152860
7	42	20	19304190
8	81	21	73424650
9	262	22	176343390
10	538	23	678390116
11	1828	24	1649008456
12	3926	25	6405031050

Table II. The meandric numbers.

endpoint should also be maintained in the stack. Thus, this character can be found in constant time. An O gets changed to a D and a U gets changed to a C.)

Notice that the amount of work (in the asymptotic sense) to generate open meanders of order n is the same as the amount of work to generate all unidirectional open meandric systems with n crossings. In the next subsection we consider how to optimize the algorithm for open meanders.

3.1 Optimizing the Generation of Meanders

To optimize the generation of meanders of length n, we can prune the computation tree by identifying sequences of length t that can never be extended into a meander of length n. For a sequence σ of length t to be extendable into a meander, the $n-t$ remaining characters must be sufficient so that it is possible for the unidirectional open meandric sequence being generated to end up with exactly one curve where the top stack is non-empty. This means that for each sequence the top stack will end up with 1 or 2 endpoints while the bottom stack will end up with 1 or 0 endpoints.

To determine the number of curves k currently in σ, we must maintain (in constant time) the size of the two stacks *top* and *bot*. The number of curves k can thus be computed: $\frac{|top|+|bot|}{2}$. Now, if σ is currently composed of $k > 1$ curves, then the only way we can reduce the number of curves is to append a C, which will reduce the number by 1 if it does not create a cycle. This means that σ can be extended into a meander of length n only if $k - 1 \leq n - t$.

We can improve on this optimization by focusing on how the endpoints are distributed among the two stacks. Since the bottom stack must end up with at most one endpoint, σ will require at least $|bot| - 1$ appended characters to satisfy this constraint. This is because each appended character will reduce the stack size by at most 1. Similarly, σ will require at least $|top| - 2$ appended characters to ensure that the stack top ends up with at most 2 endpoints. Thus σ can be extended to a meander of length n only if $|bot| - 1 \leq n - t$ and $|top| - 2 \leq n - t$. Observe that these two restrictions together imply the earlier restriction that $k - 1 \leq n - t$.

As one final optimization, observe that if $|top| > 2$ where the first two endpoints in the stack *top* correspond to the same curve (which can be tested in constant time if the stacks are implemented as arrays), then appending $|top| - 2$ characters will

n	Open meanders	Comp tree un-optimized	Ratio	Comp tree optimized	Ratio
10	538	10222	19.0	2347	4.4
11	1828	34299	18.8	9093	5.0
12	3926	108280	27.6	17800	4.5
13	13820	367697	26.6	71650	5.2
14	30694	1186862	38.7	143597	4.7
15	110954	4061487	36.6	594629	5.4
16	252939	13315389	52.6	1214315	4.8
17	933458	45809969	49.1	5140082	5.5
18	2172830	151912154	69.9	10659244	4.9
19	8152860	524688621	64.4	45919155	5.6
20	19304190	1755153136	90.9	96451719	5.0

Table III. Comparison of the running times of two algorithms to generate meanders.

not be sufficient to reduce to one curve. To see this notice that after appending $|top| - 2$ characters, the curve corresponding to the first two endpoints in the stack *top* never get affected. Thus any such extended sequence will still consist of at least two curves. Thus, in this case we can add the restriction that $|top| - 1 \leq n - t$.

To apply these optimizations to generate meanders of order n, we can simply add the following pseudocode before the first **if** statement in the algorithm `Gen(`t`)` (for generating unidirectional open meandric systems):

if $|bot| - 1 > N - t + 1$ **or** $|top| - 2 > N - t + 1$ **then return**;
if $|top| - 1 > N - t + 1$ **and** $|top| > 2$ **and** $top[1] = f[top[2]]$ **then return**;
if $t > N$ **and** ($|top| = 0$ **or** $|top| + |bot| > 2$) **then return**;

The first two lines are the optimization steps and can easily be implemented in constant time, where $top[i]$ represents the i-th element in the stack *top*. The third line is the restriction that ensures that the final sequence is an open meander.

Table III shows the amount of computation required to generate meanders comparing the unoptimized algorithm with the optimized version just described. Notice that for $n = 20$ the number of nodes in the optimized computation tree is about 18 times less than the un-optimized version. The ratios in the table represent the number of nodes in the computation tree divided by the number of meanders generated. If the ratio is bounded by a constant, then the algorithm would achieve the optimal time bound of running in constant amortized time. Even though it does not appear that the optimized version of the algorithm attains this result, it is nonetheless the fastest and simplest algorithm currently known to generate meanders.

4. SUMMARY

In this paper we have presented a CAT algorithm to generate (unidirectional) open meandric systems. The algorithm can be easily modified to yield a fast and simple algorithm to generate open and closed meanders. It can also be easily modified to list those systems with exactly k curves, however the algorithm will not run in constant amortized time. Thus, an interesting problem is to find a CAT algorithm to generate open meandric systems with n crossings and k curves.

Another interesting open problem is whether or not there exists a Gray code for (unidirectional) open meandric systems.

REFERENCES

M.H. Albert, M.S. Paterson, *Bounds for the growth rate of meander numbers*, Journal of Combinatorial Theory, Series A 112 (2005), pp. 250-262.

V.I. Arnold, *The branched covering of CP2 → S4, hyperbolicity and projective topology*, Siberian Math. J. 29, (1988), pp. 717-726 (translated from Sibirskii Matematicheskii Zhurnal 29:(36) (1988)).

R. Bacher, *Meander Algebras*, prépublication de l'Institut Fourier 478, (1999).

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

P. Di Francesco, O. Golinelli and E. Guitter, *Meander, Folding, and Arch Statistics*, Mathl. Comput. Modelling 26:(9), (1997), pp. 97-147.

F. Dorn, E. Penninkx, H. Bodlaender, and F.V. Fomin, *Efficient exact algorithms on planar graphs: Exploiting sphere cut branch decompositions*, in Proceedings of the 13th Annual European Symposium on Algorithms (ESA 2005), vol. 3669 of LNCS, Springer, Berlin, (2005), pp. 95-106.

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

I. Jensen, *A transfer matrix approach to the enumeration of plane meanders*, J. Phys. A 33:(24) (2000), pp. 5953-5963.

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

M.A. La Croix, *Approaches to the Enumerative Theory of Meanders*, Master's Essay, University of Waterloo, Canada, (2003).

S.K. Lando and A.K. Zvonkin, *Meanders*, Selecta Mathematica Sovietica, 11 no. 2 (1992), pp. 117-144.

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

W. Lunnon, *A map folding problem*, Math. of Computation 22 (1968), pp. 193-199.

A. Phillips, Simple alternating transit mazes, preprint. Abridged version appeared as "La topologia dei labirinti", in M. Emmer, editor, L'Occio di Horus: Itinerari nell'Imaginario Matematico. Istituto della Enciclopeida Italia, Rome, 1989, pp. 57-67.

H. Poincaré, *Sur un théorème de géométrie*, Rend. Circ. Mat. Palermo 33 (1912).

J. Reeds and L. Shepp, *An upper bound on the meander constant*, preprint (1999).

N. Sloane, *The on-line encyclopedia of integer sequences:* ID A000108, A001011, A060089, A060111, `www.research.att.com/~njas/sequences/index.html` (2007).

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