Exhaustive generation of positive lattice paths

Elena Barcucci
elena.barcucci@unifi.it

Antonio Bernini
antonio.bernini@unifi.it

Renzo Pinzani
renzo.pinzani@unifi.it

Dipartimento di Matematica e Informatica "U. Dini",
Università degli Studi di Firenze, Viale G.B. Morgagni 65, 50134 Firenze, Italy.

Abstract

We refer to lattice positive paths as to paths in the discrete plane constituted by different kinds of steps (north-east, east and south-east), starting from the origin and never going under the x-axis. They have been deeply studied both from a combinatorial and an algorithmic point of view. We propose some algorithms for the exhaustive generation of positive paths which are prefixes of Dyck, Motzkin and q-coloured Motzkin paths, according to their length. For each kind of path we define a recursive version as well an iterative one, specifying which path follows a given one in the lexicographic order. Furthermore we study the complexity of these algorithms by using the relations between the number of paths of given size and the number of north-east steps appearing in the final rise.

1 Introduction

Random and exhaustive generation has a fundamental role in the study of parameters characterizing different classes of combinatorial objects as well as for testing the algorithms operating on them. Many classes of paths in the plane and other combinatorial objects have been studied from the point of view of random generation [AS95, BBHT17, BBJ17, BPS95, DFLS04]. In their books [Rus03, Knu08], F. Ruskey and D. Knuth propose many algorithms for the exhaustive generation of several combinatorial objects according to their size. These algorithms can be classified into two main types: the recursive algorithms and those based on the use of a (next) procedure able to generate the successor of any object according to a fixed (usually lexicographic) order. In this paper we present some algorithms of both the above mentioned kinds for the exhaustive generation of three classes of lattice paths in the discrete plane: prefixes of Dyck, Motzkin and q-coloured Motzkin paths, also called Dyck, Motzkin and q-coloured Motzkin positive paths. Furthermore we show that the proposed algorithms have the CAT (Constant Amortized Time) property [Rus03], that is the average number of operations needed to generate any object of a given size does not depend on the size itself but it is limited by a constant.

We consider three kinds of steps $U = (1, 1)$, $D = (1, -1)$, $F = (1, 0)$, called rise, fall and flat steps. An n-length Dyck (Motzkin) path is a path in $\mathbb{Z}^2$ made up of U and D steps (U, D and F steps) starting from $(0, 0)$, ending in $(n, 0)$ and never going under the x-axis. If the flat steps can be coloured in $q \geq 2$ different ways (that is we have q different flat steps $F_1, F_2, ..., F_q$) we obtain q-coloured Motzkin paths (called bicoloured when $q = 2$). Positive Dyck (Motzkin, q-coloured Motzkin) paths are prefixes of Dyck (Motzkin, q-coloured Motzkin) paths, that is paths having the above mentioned properties but ending at a point (n, h), where $h \geq 0$ is called the height of the path.

In: L. Ferrari, M. Vamvakari (eds.): Proceedings of the GASCom 2018 Workshop, Athens, Greece, 18–20 June 2018, published at http://ceur-ws.org

We can establish a total order among the paths in the same class in a very natural way. If $p = p_1p_2...p_n$ and $r = r_1r_2...r_n$ are two paths we say that p precedes r if there is an index $k > 0$ such that $p_i = r_i$ for $i = 1, ..., k-1$ and the step p_k lies under the step r_k (for the flat steps of different colours we can choose an order whatever, so we have $D < F_1 < F_2 < ... < F_q < U$). We will define some algorithms able to exhaustively generate the paths having the same length according to the above order.

2 Dyck prefixes

Dyck paths have been widely studied, both from a combinatorial and an algorithmic point of view, directly or in connection with the numerous combinatorial structures in bijection with them and enumerated by Catalan numbers. Well-formed parenthesis, binary trees, parallelogram polyominoes, triangulations of a polygon, chords on a circle, pattern avoiding permutations are only a few examples. The book of Stanley [Sta15] presents 214 different kinds of objects counted by Catalan numbers and many others enumerated by sequences related to them. In the literature there exist also many papers dealing with exhaustive generation and Gray codes for structures enumerated by Catalan numbers (see for instance [BR98, Rus03, RW08, VW06, Wal98]).

Dyck prefixes correspond to ballot sequences and the number of n-length Dyck prefixes is

$$d_n = \binom{n}{\lfloor \frac{n}{2} \rfloor}$$

(sequence A001405 in [S]). From this we can easily obtain that

$$d_{2n} = 2d_{2n-1}$$

and

$$d_{2n+1} = \frac{2n+1}{n+1} d_{2n} < 2d_{2n}$$

These relations have a very simple interpretation. Any odd length path has a strictly positive height, so we can add at its end either a rise or a fall step. As a result from any $(2n-1)$-length path we obtain two $2n$-length different paths. The even length paths include also 0-height paths, that is Dyck paths enumerated by Catalan numbers (C_n), and only a rise step can be added at the end of these paths. As a results from the $2n$-length paths we obtain $2d_{2n} - C_n$ different $(2n+1)$-length paths.

In order to exhaustively generate the n-length paths according to the natural order above defined, we represent them by means of n-length words on the alphabet $\{0, 1\}$, by associating 1 to each rise step and 0 to each fall step, and then generate the words in lexicographic order.

The first n-length word is $first(n) = (10)^{\frac{n}{2}}$ if n is even and $first(n) = (10)^{\lfloor \frac{n}{2} \rfloor}1$ if n is odd. The last one is always 1^n. Furthermore we define $first(n) = \varepsilon$ when $n \leq 0$. We denote by $h(w)$ the difference between the number of 1's and the number of 0's in the word w, that is the height of the corresponding path. Let $w = v0$, then the following word w' is $v1$ and $h(w') = h(w) + 2$. If $w = v01^p$ then $w' = v10^kz$ where k is the maximum number of 0's that can be added after $v1$, that is $k = \min\{p, h(w) - p + 2\}$ because $h(v1) = h(w) - p + 2$, and $z = first(p - k)$. Furthermore, $h(w') = h(w) - 2p + 2$ if $p < h(w) - p + 2$ and $h(w') = h(z)$ otherwise.

As a result, in order to obtain the successor of a word w it is necessary to scan w from right to left till the rightmost 0 is detected and then modify w from this position to the end as described above. This can be achieved by the procedure *next*() (described in Algorithm 1 in a Java style). We suppose that a is a global array containing w in the entries $1..n$, $a[0] = 0$ is a flag for recognizing the last word, h is a global variable containing the height of the path represented in a and $finish$ a global boolean variable whose value becomes *true* when the procedure recognizes the last sequence. After initializing a with $first(n)$, h with $h(first(n))$ and $finish$ with $false$, the procedure is used into a do-while cycle which stops when $finish$ becomes *true*.

In order to evaluate the complexity of the algorithm it is easy to see that the number of elements tested and modified for generating the next sequence from $w = v01^p$ is equal to $p + 1$. Let t_n be the total number of final 1's in all the n-length words (that is the number of 1's which take part in the 1^p suffixes). The following relation

Algorithm 1 procedure next for Dyck prefixes

```
void next(){
  int j, p=0;
  while (a[n-p]==1) p=p+1;
  if (p==n) finish=true;
  else {
    a[n-p]=1;
    h=h-p+2;
    j=n-p+1;
    while (h>0 && j<=n){a[j]=0; j=j+1; h=h-1;}
    while (j<n) {a[j]=1; a[j+1]=0; j=j+2;}
    if (j==n) {a[n]=1; h=h+1;}
  }
}
```

holds $t_n = t_{n-1} + d_{n-1}$ because an element 1 can be added at the end of any $(n-1)$-length word and the final 1's are still final 1's. So the total number of operations necessary to generate all the n-length words is $t_n + d_n$ and the average number for each word is

$$avg_n = \frac{t_n + d_n}{d_n} = \frac{t_n}{d_n} + 1$$

So the quantity $r_n = \frac{t_n}{d_n}$ must be evaluated.

$$r_{2n} = \frac{t_{2n}}{d_{2n}} = \frac{t_{2n-1} + d_{2n-1}}{2d_{2n-1}} = \frac{1}{2}\left(r_{2n-1} + 1\right)$$

$$r_{2n+1} = \frac{t_{2n+1}}{d_{2n+1}} = \frac{t_{2n} + d_{2n}}{\frac{2n+1}{n+1}d_{2n}} = \frac{t_{2n-1} + d_{2n-1} + 2d_{2n-1}}{2\frac{2n+1}{n+1}d_{2n-1}} = \frac{n+1}{2(2n+1)}\left(r_{2n-1} + 3\right)$$

If $r_{2n-1} < 2$ then

$$r_{2n+1} < \frac{n+1}{2(2n+1)}5 = \frac{5n+5}{4n+2} < 2$$

for $n > 1$.

Since $r_1 < 2$ we can conclude that $r_n < 2$ for $n \geq 1$ and $avg_n < 3$, so the algorithm has the CAT property.

A recursive version can also be defined. The array a and h have the same meaning and do not necessitate any initialization. The array a is filled from left to right: the first parameter of the procedure *dyckR* is the next position j to be defined and the second parameter indicates the height of the Dyck prefix represented in the first $j-1$ entries of a. If the value of the first parameter in a call is greater than n, a new prefix is contained in a and it can be displayed (or elaborated). Otherwise the prefix is not yet completed: a fall step is added ($a[j] = 0$) and a recursive call with parameters $j+1$ and $h-1$ is generated only if $h > 0$, while a rise step can always be added ($a[j] = 1$) followed by a recursive call with parameters $j+1$ and $h+1$. The initial call is $dyckR(1, 0)$.

In order to evaluate the complexity we can consider the total number of recursive calls made for generating all the n-length words, because any call makes a constant number of operations. We can slightly modify the procedure in such a way that any call outputs a new word or generate exactly two recursive calls. In this way the recursion tree is a binary tree with d_n leafs and $d_n - 1$ internal nodes and, as a result, we have that the average number of calls for a word is less than 2 and the algorithm is CAT. When we add a fall step, the height of the prefix decreases by 1. So if the height was 1 it becomes 0 and at a successive call only a rise step can be added. So when the height is equal to 1 and we add a fall step in position j, if $j < n$, we add also a rise step in position $j+1$ and make a call with parameters $j+2$ and 1. As a result, the only calls with $h = 0$ are those with $j = n$ and in the recursion tree any internal node has exactly two sons. In this case the initial call should

Algorithm 2 recursive procedure for Dyck prefixes

```
void dyckR(int j, int h){
  if (j>n) {print(a);}
  else {
    if (h>0) {a[j]=0; dyckR(j+1, h-1);}
    a[j]=1; dyckR(j+1, h+1);
  }
}
```

Algorithm 3 improved recursive procedure for Dyck prefixes

```
void dyckRC(int j, int h){
  if (j>n) {print(a);}
  else {
    a[j]=0;
    if (h==1 && j<n) {a[j+1]=1; dyckRC(j+2, 1);}
    else dyckRC(j+1, h-1);
    a[j]=1; dyckRC(j+1, h+1);
  }
}
```

be $\mathit{dyckRC}(2,1)$, after initializing $a[1] = 1$.

3 Motzkin prefixes

Motzkin numbers [Aig98, DS77] enumerate many combinatorial structures which are very close to those enumerated by Catalan numbers [Sta15]. Motzkin prefixes have been deeply studied mainly due to their relation with directed animals and percolation [BPS94, GV88]. They are enumerated by sequence A005773 in [S]. There is no closed formula for the number m_n of n-length Motzkin prefixes, but for our sakes the recurrence relation (18) [BPS91]:

$$m_n = 2m_{n-1} + 3\frac{n-1}{n+1}m_{n-2},$$

with the conditions $m_0 = 1$ and $m_1 = 2$, is sufficient to state that

$$m_n > 2m_{n-1}.$$

In order to exhaustively generate the n-length Motzkin prefixes we represent them by means of n-length words on the alphabet $\{0, 1, 2\}$, by associating 2 to each rise step, 1 to each flat step and 0 to each fall step. In this way the natural order for the paths defined in the introduction corresponds to the lexicographic order on the words.

The first n-length word is 1^n and the last one is 2^n. We denote by $h(w)$ the height of the path coded by w, that is the difference between the number of 2's and the number of 0's in the word w.

Let $w = v0$ ($w = v1$), then the following word w' is $v1$ ($w' = v2$) and $h(w') = h(w) + 1$. If $w = v02^p$ then $w' = v10^k1^{p-k}$ where k is the maximum number of 0's that can be added after $v1$, that is $k = \min\{p, h(w)-p+1\}$ because $h(v1) = h(w) - p + 1$. Furthermore, $h(w') = h(w) - p + 1 - k$. In an analogous way, if $w = v12^p$ then $w' = v20^k1^{p-k}$ where k is the maximum number of 0's that can be added after $v2$, that is $k = \min\{p, h(w)-p+1\}$ because $h(v2) = h(w) - p + 1$. In this case too, $h(w') = h(w) - p + 1 - k$.

As a result, it is necessary to scan a word w from right to left till the rightmost element different from 2 is detected and then modify w from this position to the end as described above, in order to obtain its successor. This can be achieved by the procedure *next*() (described in Algorithm 4 in a Java style). We suppose that a is a global array containing w in the entries $1..n$, $a[0] = 0$ is useful for recognizing the last word, h is a global variable containing the height of the path represented in a and $finish$ a global boolean variable whose value becomes $true$ when the procedure recognizes the last sequence. After initializing a with 1^n, h with 0 and $finish$ with $false$, the procedure is used into a do-while cycle which stops when $finish$ becomes $true$.

Algorithm 4 procedure next for Motzkin prefixes

```
void next(){
   int j, p=0;
   while (a[n-p]==2) p=p+1;
   if (p==n) finish=true;
   else {
      a[n-p]=a[n-p]+1;
      h=h-p+1;
      j=n-p+1;
      while (h>0 && j<=n) {a[j]=0; h=h-1; j=j+1;}
      while (j<=n) {a[j]=1; j=j+1;}
   }
}
```

In order to evaluate the complexity of the algorithm we remark that the number of elements tested and modified for generating the next sequence from $w = v02^p$ or $w = v12^p$ is equal to $p + 1$. Let t_n be the total number of final 2's in all the n-length words (that is the number of 2's which take part in the 2^p suffixes). The following relation holds $t_n = t_{n-1} + m_{n-1}$ because an element 2 can be added at the end of any $(n-1)$-length word and the final 2's are still final 2's. So the total number of operations necessary to generate all the n-length words is $t_n + m_n$ and the average number for each word is

$$avg_n = \frac{t_n + m_n}{m_n}$$

It is easy to verify that $t_n < m_n$. This is true for $n = 1$, and if we assume that $t_k < m_k$ for $k \leq n-1$, we obtain

$$t_n = t_{n-1} + m_{n-1} < 2m_{n-1} < m_n.$$

As a result

$$avg_n = \frac{t_n + m_n}{m_n} < \frac{m_n + m_n}{m_n} = 2.$$

So this algorithm too has the CAT property.

A recursive version can be defined in this case too. The array a and h have the same meaning and do not necessitate any initialization. The array a is filled from left to right: the first parameter of the procedure *motzR* is the next position j to be defined and the second parameter indicates the height of the Motzkin prefix represented in the first $j-1$ entries of a. If the value of the first parameter in a call is greater than n, a new prefix is contained in a and it can be displayed (or elaborated). Otherwise the prefix is not yet completed: a fall step is added ($a[j] = 0$) and a recursive call with parameters $j+1$ and $h-1$ is generated only if $h > 0$, while a flat and a rise step can always be added ($a[j] = 1$ and $a[j] = 2$) followed by a recursive call with parameters $j+1$ and h for the flat step and parameters $j+1$ and $h+1$ for the rise step, respectively. The initial call is $motzR(1, 0)$.

In order to evaluate the complexity we can consider the total number of recursive calls made for generating all the n-length words, because any call makes a constant number of operations. We remark that any call outputs a new word or generate at least two recursive calls. The recursion tree is a tree with m_n leafs and a number of internal nodes less than $m_n - 1$ and, as a result, the average number of calls for a word is less than 2 and the algorithm is CAT.

Algorithm 5 recursive procedure for Motzkin prefixes

```
void motzR(int j, int h){
  if (j>n) {print(a);}
  else {
        if (h>0) {a[j]=0; motzR(j+1, h-1);}
        a[j]=1; motzR(j+1, h);
        a[j]=2; motzR(j+1, h+1);
  }
}
```

4 q-coloured Motzkin prefixes

q-coloured Motzkin paths [BDPP95] are a natural generalization of Motzkin paths obtained by allowing q different colours be used for flat steps. When $q = 2$, bicoloured Motzkin paths are enumerated by Catalan numbers, while their prefixes are counted by the binomial coefficients $d_n = \binom{2n+1}{n+1}$ (sequence A001700 in [S]).

The discussion relative to Motzkin prefixes can be easily extended to the prefixes of q-coloured Motzkin paths. They can be represented by words on the alphabet $\Sigma_q = \{0, 1, ..., q, q+1\}$ where $q+1$ corresponds to a rise step, 0 to a fall step and 1, 2 ,..., q to the different flat steps. The first n-length word is 1^n and the last one $(q+1)^n$. If $w = vx(q+1)^p$, where $x \in \Sigma_q$ and $x < q+1$, then the successive word is $w' = vx'0^k1^{p-k}$, where $x' = x+1$ and k is the maximum number of 0's that can be added after vx'. We have that $h(vx') = h(vx) = h(w) - p$ if $0 < x < q+1$, because we only change the colour of a flat step, and $h(vx') = h(vx)+1 = h(w)-p+1$ otherwise, because we transform a fall step in a flat step or a flat step in a rise step. As a result, $k = \min\{p, h(vx')\}$.

In order to obtain w' it is again necessary to scan w from right to left till the rightmost element different from $q+1$ is detected and then modify w starting from this position as described above. This is achieved by the procedure *next*() (Algorithm 6). The variables, their meaning and initialization and the use context of the procedure are the same as for Motzkin prefixes.

Algorithm 6 procedure next for q-coloured Motzkin prefixes

```
void next(){
  int j, p=0;
  while (a[n-p]==q+1) p=p+1;
  if (p==n) finish=true;
  else {
    a[n-p]=a[n-p]+1;
    h=h-p;
    if(a[n-p]==1||a[n-p]==q+1) h=h+1;
    j=n-p+1;
    while (h>0 && j<=n) {a[j]=0; h=h-1; j=j+1;}
    while (j<=n) {a[j]=1; j=j+1;}
  }
}
```

As far as the complexity of the algorithm is concerned, we remark that the number of elements tested and modified for generating the next sequence from $w = vx2^p$, where $x < q+1$, is equal to $p+1$. Let $c_{n,q}$ be the number of n-length q-coloured Motzkin prefixes. We have that $c_{n,q} > (q+1) \cdot c_{n-1,q}$ because a rise or any flat step can always be added to an $(n-1)$-length q-coloured prefix.

The total number $t_{n,q}$ of final $(q+1)$'s in all the n-length words again satisfies the relation $t_{n,q} = t_{n-1,q} + c_{n-1,q}$ because an element $(q+1)$ can be added at the end of any $(n-1)$-length word and the final $(q+1)$'s are still final $(q+1)$'s. So the total number of operations necessary to generate all the n-length words is $t_{n,q} + c_{n,q}$.

In this case too, we can verify that $t_{n,q} < c_{n,q}$. This is true for $n = 1$, as a matter of fact $t_{1,q} = 1$ and $c_{1,q} = q+1$. By assuming that $t_{k,q} < c_{k,q}$ for $k \leq n-1$, we obtain

$$t_{n,q} = t_{n-1,q} + c_{n-1,q} < 2c_{n-1,q} < c_{n,q}.$$

As a result the average number of elements tested and modified in to generate each word is

$$avg_n = \frac{t_{n,q} + c_{n,q}}{c_{n,q}} < \frac{c_{n,q} + c_{n,q}}{c_{n,q}} = 2$$

and this algorithm too has the CAT property.

We can also define a recursive procedure with the same parameters used for Motzkin prefixes. This procedure (Algorithm 7) adds in position j, corresponding to the first parameter, a fall step (if the height h of the prefix so long defined is positive), a flat step of any colour, and a rise step. Obviously the recursion stops when $j > n$.

Algorithm 7 recursive procedure for q-coloured Motzkin prefixes

```
void qmotzR(int j, int h){
  int i;
  if (j>n) {print(a);}
  else {
    if (h>0) {a[j]=0; qmotzR(j+1, h-1);}
    for (i=1; i<=q; i=i+1) {a[j]=i; qmotzR(j+1, h);}
    a[j]=q+1; qmotzR(j+1, h+1);
  }
}
```

Any call of such procedure produces a new sequence or generate at least $q + 1$ recursive calls. As a result this version too has the CAT property.

5 Further developments

Recently binary words avoiding one or more patterns [BMPP12] have been employed in order to construct sets of words constituting a non-overlapping code [Bla15], in particular Dyck words and Motzkin words were used in [BPP12] and [BBPPS16]. Their exhaustive generation and Gray code were tackled in [BBPSV14, BBPSV15, BBPV15, BBPV17]. It would be interesting to establish similar procedures for prefixes of words avoiding those patterns.

Acknowledgements

This work was partially supported by the GNCS 2018 project "Proprietà combinatorie e rilevamento di pattern in strutture discrete lineari e bidimensionali".

References

[Aig98] M. Aigner. Motzkin numbers. *European Journal of Combinatorics*, 19:663-675, 1998.

[AS95] L. Alonso and R. Schott. *Random generation of trees - random generators in computer science.* Kluwer, 1995.

[BBHT17] A. Bacher, O. Bodini, H.-K. Hwang and T.-H. Tsai. Generating Random Permutations by Coin Tossing: Classical Algorithms, New Analysis, and Modern Implementation. *ACM Transactions on Algorithms*, 13:Art. 24, 43 pp., 2017.

[BBJ17] A. Bacher, O. Bodini and A. Jacquot. Efficient random sampling of binary and unary-binary trees via holonomic equations. *Theoretical Computer Science*, 695:42–53, 2017.

[BBPPS16] E. Barcucci, S. Bilotta, E. Pergola, R. Pinzani and J. Succi. Cross-bifix-free sets generation via Motzkin paths. *RAIRO - Theoretical Informatics and Applications*, 50:81–91, 2016.

[BPS91] E. Barcucci, R. Pinzani and R. Sprugnoli. The Motzkin family. *Pure Mathematics and Applications*, 2:249–279, 1991.

[BPS94] E. Barcucci, R. Pinzani and R. Sprugnoli. The random generation of directed animals. *Theoretetical Computer Science*, 127:333–350, 1994.

[BPS95] E. Barcucci, R. Pinzani and R. Sprugnoli. The random generation of underdiagonal walks. *Discrete Mathematics*, 139:3–18, 1995.

[BDPP95] E. Barcucci, A. Del Lungo, E. Pergola and R. Pinzani. A Construction for Enumerating k-coloured Motzkin Paths. *Lecture Notes in Computer Science*, 959:254–263, 1995.

[BBPSV14] A. Bernini, S. Bilotta, R. Pinzani, A. Sabri and V. Vajnovszki. Prefix partitioned Gray code for particular cross-bifix-free sets. *Cryptography and Communications*, 6:359–369, 2014.

[BBPSV15] A. Bernini, S. Bilotta, R. Pinzani, A. Sabri and V. Vajnovszki. Gray code orders for q-ary words avoiding a given factor. *Acta Informatica*, 52:573–592, 2015.

[BBPV15] A. Bernini, S. Bilotta, R. Pinzani and V. Vajnovszki. A trace partitioned Gray code for q-ary generalized Fibonacci strings. *Journal of Discrete Mathematical Sciences and Cryptography*, 18:751–761, 2015.

[BBPV17] A. Bernini, S. Bilotta, R. Pinzani and V. Vajnovszki. A Gray code for cross-bifix-free sets. *Mathematical Structures in Computer Science*, 27:184–196, 2017.

[BMPP12] S. Bilotta, D. Merlini, E. Pergola and R. Pinzani. Pattern $1^{j+1}0^j$ avoiding binary words. *Fundamenta Informaticae*, 117:35–55, 2012.

[BPP12] S. Bilotta, E. Pergola and R. Pinzani. A new approach to cross-bifix-free sets. *IEEE Transactions on Information Theory*, 58:4058–4063, 2012.

[Bla15] S. Blackburn. Non-overlapping codes. *IEEE Transactions on Information Theory*, 61:4890–4894, 2015.

[BR98] B. Bultena and F. Ruskey. An Eades-McKay algorithm for well formed parantheses strings. *Information Processing Letters*, 68:255-259, 1998.

[DS77] R. Donaghey and L. Shapiro. Motzkin numbers. *Journal of Combinatorial Theory Series A*, 23:291-301, 1977.

[DFLS04] P. Duchon, P. Flajolet, G. Louchard and G. Schaeffer. Boltzmann samplers for the random generation of combinatorial structures. *Combinatorics Probability and Computing*, 13:577–625, 2004.

[GV88] D. Gouyou-Beauchamps and G. Viennot. Equivalence of the two-dimensional directed animal problem to a one-dimensional path problem. *Advances in Applied Mathematics*, 9:334–357, 1988.

[Knu08] D. E. Knuth. *The Art of Computer Programming, vol. 4.* Addison-Wesley, 2008.

[Rus03] F. Ruskey. *Combinatorial Generation.* At http://www.1stworks.com/ref/RuskeyCombGen.pdf, 2003.

[RW08] F. Ruskey and A. Williams. Generating Balanced Parentheses and Binary Trees by Prefix Shifts. In: *Proc. Fourteenth Computing: The Australasian Theory Symposium, CRPIT*, 77:107–115, 2008.

[S] N. J. A. Sloane. *The on-line encyclopedia of integer sequences.* At http://oeis.org.

[Sta15] R. P. Stanley. *Catalan numbers.* Cambridge University Press, 2015.

[VW06] V. Vajnovszki and T. Walsh. A loop-free two-close algorithm for listing k-ary Dyck words. *Journal of Discrete Algorithms*, 4:633–648, 2006.

[Wal98] T. Walsh. Generation of well-formed parenthesis strings in constant worst-case time. *Journal of Algorithms*, 29:165–173, 1998.