PU.M.A. Vol. **17** (2006), No. 1–2, pp. 39–53

An exhaustive generation algorithm for Catalan objects and others

Antonio Bernini
Dipartimento Sistemi e Informatica, viale G.B. Morgagni 65, 50134 Firenze, Italy
e-mail: bernini@dsi.unifi.it

and
Irene Fanti
Dipartimento Sistemi e Informatica, viale G.B. Morgagni 65, 50134 Firenze, Italy
e-mail: fanti@dsi.unifi.it

and
Elisabetta Grazzini
Dipartimento Sistemi e Informatica, viale G.B. Morgagni 65, 50134 Firenze, Italy
e-mail: ely@dsi.unifi.it

(Received: October 31, 2006)

Abstract. The paper presents a CAT generation algorithm for Dyck paths with fixed length n. It is the formalization of a method for the exhaustive generation of these paths which can be described by two equivalent strategies. The former uses a rooted tree, the latter lists the paths and provides a visit of the nodes of the tree, basing on three simple operations. These constructions are strictly connected with ECO method and can be encoded by a rule, very similar to the *succession rule* in ECO, with a finite number of labels for each n. Moreover, with a slight variation, this method can be generalized to other combinatorial classes like Grand Dyck or Motzkin paths.

Mathematics Subject Classifications (2000). 68R05

1 Introduction

One of the most important aims in combinatorics has always been the generation of objects of a particular class according to a fixed parameter. Actually, many practical questions require, for their solution, an exhaustive search throughout all the objects in the class. In general, the idea is to find methods listing in a particular order combinatorial objects, without either repetitions or omissions so that it is possible to deduce a recursive construction of the studied class. We are talking about exhaustive generation algorithms [22] which can be seen as an enumerating technique where each object is counted and recorded once it is generated [23]. Often, these algorithms are useful in several areas such as hardware and software testing, thermodynamic, biology and biochemistry ([3, 6, 7]) where it could be helpful to have a particular order of the objects.

In literature one of the common approach has been the generation of the combinatorial elements in such a way that two successive items differ only slightly; in this sense a well known example is the classical binary reflected Gray code scheme for listing n-bit binary numbers so that successive items differ by exactly

one bit. Gray codes find a lot of applications in many different areas (for more details and examples see [4, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 20, 21, 24]).

In this paper we present a method for the generation of all objects of a combinatorial class with fixed size. We focus our attention on Dyck paths and we introduce two strategies for their generation. The former uses an operator which can be described by a rooted tree, the latter lists the objects using three operations and it corresponds to visit the nodes of the tree. Both of them use only a constant amount of computations per object in amortized sense, so they have CAT property [1].

Our argument is based on ECO method. This link suggests to look for a rule, similar to the *succession rule* in ECO, for encoding the construction of the list (for more details about ECO method and *succession rule* see [1, 2, 5, 25, 26]). Typically, in the framework of generation, ECO method is used for the construction of the objects with any size n: starting from the minimal size, an ECO construction generates all objects of size $n+1$ from the objects of size n. In our work, ECO method is used to construct only objects with a fixed sized n. A similar aim is pursued in [1], where the authors use strings of integers for encoding the objects. Then, they have to use a different algorithm, with a proportional amount of computations, to come back to the objects from their encodes. The main difference in our approach is that we uses directly the combinatorial objects, providing the generation of Dyck paths of size n by some operations with a constant cost.

In Section 2 we give some preliminaries and notations, then our main idea is presented. Section 3 introduces an operator for the construction of a tree where the nodes are all the Dyck paths of length $2n$. The tree can be described by a succession rule (Section 3.1). Section 3.2 shows how the tree can be visited by means of three simple operations leading to an efficient algorithm using binary strings (Section 3.3).

2 Preliminaries and notations

We define *path* as a sequence of points in $\mathbb{N} \times \mathbb{N}$ (negative coordinates are not allowed) and *step* as a pair of two consecutive points in the path. A *Dyck path* is a path $\mathcal{D} := \{s_0, s_1, \ldots, s_{2n}\}$ such that $s_0 = (0,0)$, $s_{2n} = (2n, 0)$ with only northeast ($s_i = (x, y)$, $s_{i+1} = (x+1, y+1)$) or southeast ($s_i = (x, y)$, $s_{i+1} = (x+1, y-1)$) steps. The number of northeast steps is equal to the number of southeast steps and the path's length is the number of its steps. In particular $\mathcal{D}_n$ is the set of Dyck paths with length $2n$. Hereafter we say that $\mathcal{D} \in \mathcal{D}_n$ has *size* n.

A *peak* (resp. *valley*) is a point s_i such that step (s_{i-1}, s_i) is a northeast (southeast) and the step (s_i, s_{i+1}) is a southeast (northeast); moreover, we call *pyramid* p_h , $\forall\ h \in \mathbb{N}$, a sequence of h northeast steps followed by h southeast steps such that if (s_i, s_{i+1}) is the first northeast step and (s_{i+2h-1}, s_{i+2h}) is the last southeast step of this sequence, then $s_i = (x, 0)$ and $s_{i+2h} = (x + 2h, 0)$.

We also define *last descent* (*ascent*) as the rightmost contiguous sequence of southeast (northeast) steps of the Dyck path. We conventionally number its points from right (left) to left (right); clearly, the last point of last descent always coincides with the last point of last ascent (see Figure 1). Moreover, if we denote by $h(s_i)$ the *height* of the point s_i (i.e. its ordinate) and by *non-decreasing point* the extremity s_{i+1} of a northeast step (s_i, s_{i+1}), then we can define the *area of a path* as the sum of its non-decreasing points' heights and *maxima area path* $P^n_{\max}$ the pyramid p_n that contains, in geometric sense, all paths of the size n. Finally, we call a path P "active" if we obtain another Dyck path when the first and the last step of P are taken off. This is equivalent to say that P does not have valleys with height $h = 0$.

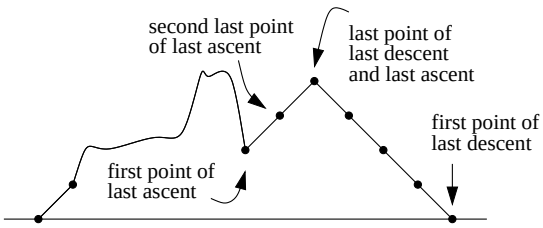

Figure 1: numeration of points of Dyck path's last descendent and ascent.

Given a class of combinatorial objects $\mathcal{C}$ and a parameter $\gamma : \mathcal{C} \longrightarrow \mathbf{N}^+$ such that $\mathcal{C}_n = \{x \in \mathcal{C} : \gamma(x) = n\}$ is a finite set for all n, we define a *generating tree* for this class. We assume there is only one element of minimal size in $\mathcal{C}$ and we describe the recursive construction of this set by using a rooted tree in which each node corresponds to an object. In particular, the vertices on the nth level represent the elements of $\mathcal{C}_n$, the root of the tree is the smallest element and the branch, leading to the node, encodes the choices made in the construction of the object. Starting from these properties and choosing the combinatorial class $\mathcal{D}$ of Dyck paths, we introduce another kind of generating tree which describe, fixed the size n, the recursive construction of $\mathcal{D}_n$. We denote it with $\mathcal{D}_n$*-tree* which clearly has a finite number of levels and contains objects with the same size, regardless of the level. The structure of a generating tree can be elegantly describe by means of the notion of *succession rule*. Moreover, our algorithm is based on the ECO method which is a general method to enumerate combinatorial objects. The basic idea of this one is the definition of a recursive construction for $\mathcal{C}$ by means of an operator ϕ which performs a "local expansion" on the objects (for more details see [2]).

3 Dyck paths

We define an operator which constructs $\mathcal{D}_n$; we study this operator for $n \geq 3$, being the cases $n = 1$ and $n = 2$ trivial.

θ operator:

1. Consider $P^n_{\max}$ as the first path.
2. Take off the first and the last step of $P^n_{\max}$ and insert a peak in every point of the last descent of the Dyck path obtained by removing the first and the last step. Every insertion generates a new Dyck path.
3. For each new generated path repeat the following actions while active paths are generated:
   - 3.1 Take off the first and the last step of the path.
   - 3.2 Insert a peak in every point of the last descent of the obtained Dyck path. Every insertion generates a new path.

In Figure 2 we give an example of θ operator's action.

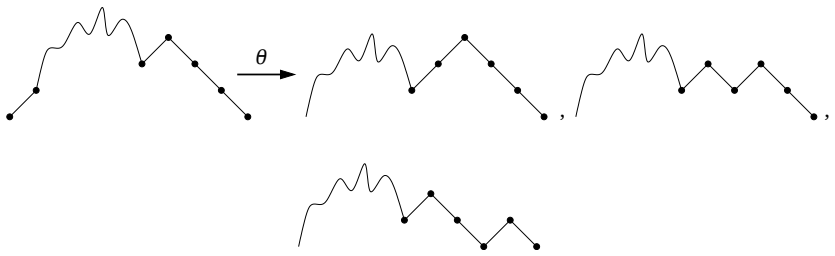

Figure 2: action of θ.

We prove that θ satisfies the following conditions:

PROPOSITION 3.1

1. $\forall\ X_1, X_2 \in \theta(P^n_{\max})$, *then* $X_1 \neq X_2$;

2. $\forall\ X_1,\ X_2 \in \mathcal{D}_n$ *and* $X_1 \neq X_2$, *then* $\theta(X_1) \cap \theta(X_2) = \emptyset$.

PROPOSITION 3.2 $\forall\ Y \in \mathcal{D}_n\ \exists$ *a finite sequence* $X_0, X_1, \ldots, X_k$ *with* $k \in \mathbb{N}$ *and* $X_k = Y$ *such that :*

- $X_0 = P^n_{\max}$;
- $X_{i+1} \in \theta(X_i) \qquad 0 \leq i \leq k-1.$

Proof Proposition 3.1. We prove only point 2. Consider $X_1,\ X_2 \in \mathcal{D}_n$, $X_1 \neq X_2$ and decompose both X_1 and X_2 in two parts as shown in Figure 3.

If $b_1 \neq b_2$, then they remain distinct after the application of θ since it operates just on these parts. On the other hand, if $b_1 = b_2$, then $a_1 \neq a_2$ and after the application of θ the parts a_1 and a_2 remain different. Then $\theta(X_1) \cap \theta(X_2) = \emptyset$ in both cases.

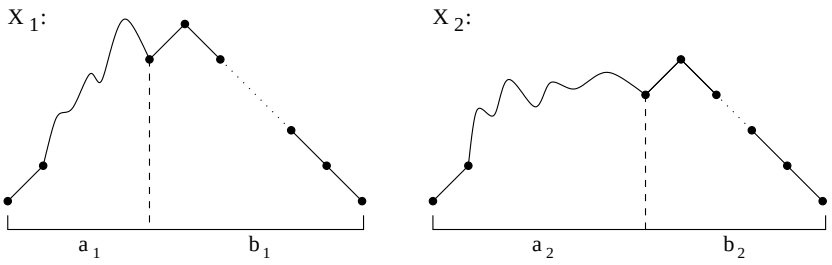

Figure 3: decomposition of a path.

Proof Proposition 3.2. Let us consider a general path Y and apply the inverse operator θ^{-1}; clearly, θ^{-1} takes off the rightmost peak of Y and inserts a northeast step and a southeast step at the beginning and at the end, respectively. Two cases are possible:

1. If last ascent of Y has only one step, then in the obtained path $\theta^{-1}(Y)$ the peaks' number is reduced by one.

2. If last ascent of Y has at least two steps, then the number of last ascent's steps in $\theta^{-1}(Y)$ is reduced by one.

It is easily seen that after k iterations the path $\theta^{-1}(Y)$ has only one peak and coincides with $P^n_{\max}$.

The construction of $\mathcal{D}_n$-tree:

The construction performed by θ can be described by a rooted tree as follows:

1. The root is $P^n_{\max}$ and it is at level zero;

2. if $X \in \mathcal{D}_n$-tree is at level $k \geq 0$, then $Y \in \theta(X)$ is a son of X and it is at level $k+1$.

In Figure 4 $\mathcal{D}_4$-tree is illustrated.

THEOREM 3.1 *$\mathcal{D}_n = \mathcal{D}_n$-tree*

Proof: Let $X \in \mathcal{D}_n$-tree; it is clear that X is a Dyck path. Moreover, Proposition 3.1 assures there are not two copies of the same path in $\mathcal{D}_n$-tree, then $|\mathcal{D}_n\text{-tree}| \leq |\mathcal{D}_n|$ and $\mathcal{D}_n\text{-tree} \subseteq \mathcal{D}_n$.

Vice versa, given $Y \in \mathcal{D}_n$, Proposition 3.2 assures that it is always possible to find a finite sequence of Dyck paths joining $P^n_{\max}$ path to Y; so $Y \in \mathcal{D}_n$-tree since $P^n_{\max}$ is in $\mathcal{D}_n$-tree, then $\mathcal{D}_n \subseteq \mathcal{D}_n$-tree.

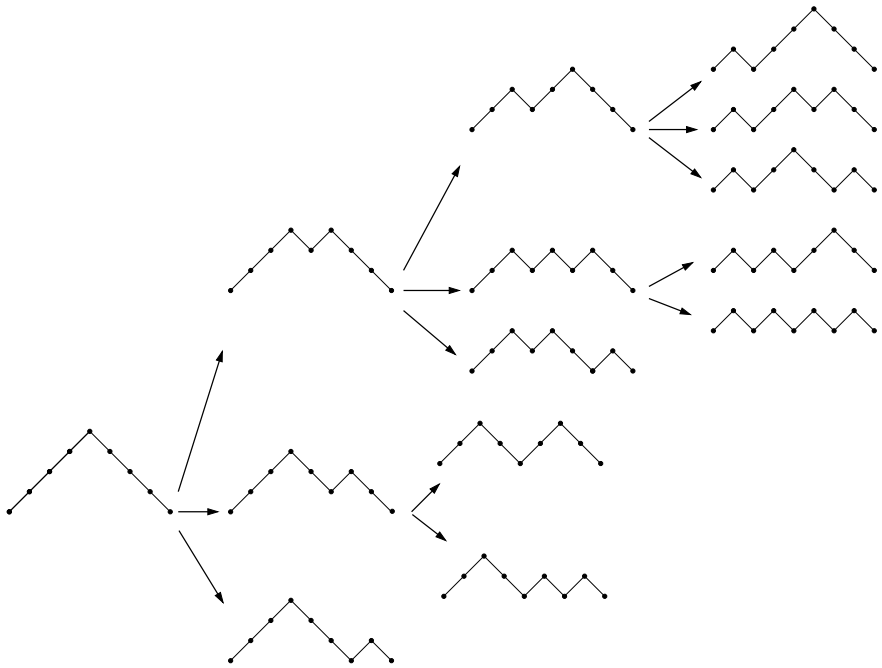

Figure 4: $\mathcal{D}_4$-tree.

3.1 Succession rule

The above construction of $\mathcal{D}_n$-tree can be coded in a succession rule. Given a path P, we have $\theta(P) \neq \emptyset$ if and only if it is active, i.e. if it has not valleys with height $h = 0$. Moreover, from the definition of θ it is clear that the cardinality of the set of sons of a path P is equal to the number of steps in the last descent of P. We label each path with an information which says the number of its sons and the height of its lowest valley. The following notation to connect the label of a parent P, having the height of its lowest valley equal to i, with the labels of its k sons, is used:

$$(k, i) \hookrightarrow (c_1)(c_2)\dots(c_k).$$

Moreover, each of these k paths has the last descent with length s, with $s = 1, 2, \dots, k$. Let $\overline{P}$ be the path obtained from P by removing the first and the last step (this is a part of the action of θ). After the insertion of a peak in one of the last descent's point of $\overline{P}$, the number of valleys is increased by one, with the exception of that path Y obtained by inserting the peak in the last point of $\overline{P}$'s last descent. Actually, in this case, the path Y has the same number of valleys of its father P. It is clear that the height j of the lowest valley of a son of P depends on the insertion of the peak. Indeed, if θ inserts the peak in the t-th point of the $\overline{P}$'s last descent with $1 \leq t \leq i-1$, then $j = t-1$, i.e. the lowest valley is generated by the insertion of the peak. On the other hand, if

$i \leq t \leq k$, then $j = i - 1$, i.e. the lowest valley is the same of $\overline{P}$. In Figure 5 we give an example of θ's action on a path with label $(3, 2)$.

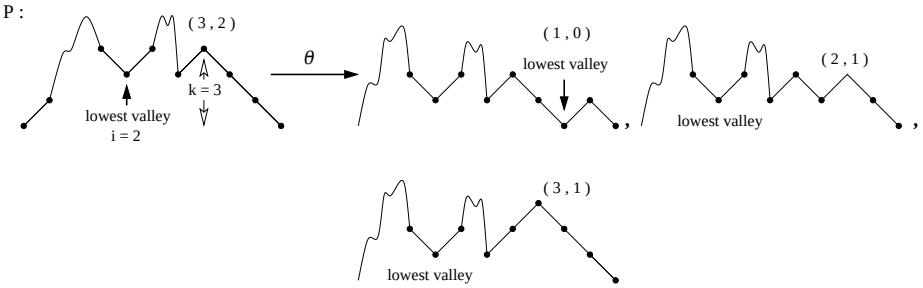

Figure 5: action of θ on a path with label $(3, 2)$.

The above description can be encoded in the following production:

$$(k, i) \hookrightarrow (1, 0)(2, 1) \dots (i, i-1)(i+1, i-1) \dots (k, i-1).$$

We notice that the root of $\mathcal{D}_n$-tree does not have valleys, then the second index of its label should be empty; nevertheless, we agree to label the root by $(n-1, n-1)$. Finally, the following succession rule is obtained:

$$\begin{cases} (n-1, n-1) \\ (k, i) \hookrightarrow (1, 0)(2, 1) \dots (i, i-1)(i+1, i-1) \dots (k, i-1). \end{cases}$$

It is clear that labels with $i = 0$ correspond to paths with at least a valley with height $h = 0$ and they do not generate any other path by θ operator. In Figure 6 we give an example of generating tree for $n = 5$.

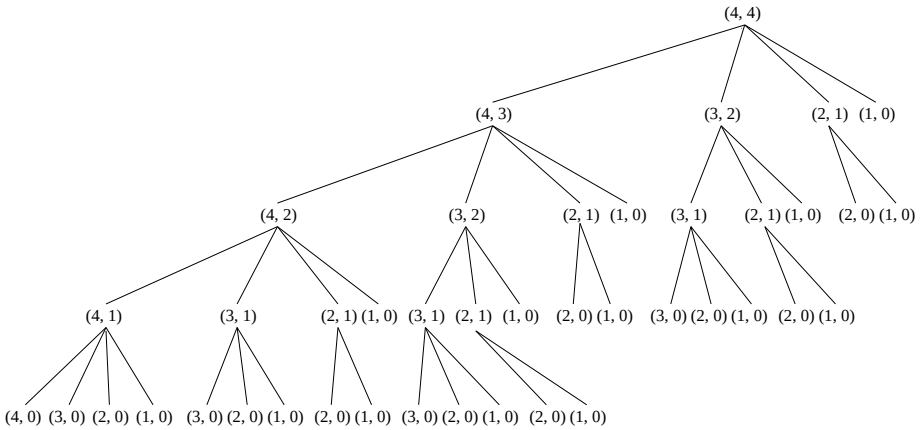

Figure 6: generating tree for $n = 5$.

3.2 The generating algorithm

The aim of this section is to find a method which sequentially lists the Dyck paths of $\mathcal{D}_n$-tree so that each one is generated only by the last generated path. In other words, we are looking for a procedure to visit all the nodes of the tree. Afterwards, we present an algorithm which implements this kind of visit.

It is helpful to order sons of a path X, according to the decreasing lengths of their last descent so that the last one ends in p_1. In particular, the last $P^n_{\max}$'s son is composed by p_{n-1} followed by p_1. We name "firstborn" of a path P the son with the longest last descent (In Figure 7 we give an example of a path's "firstborn").

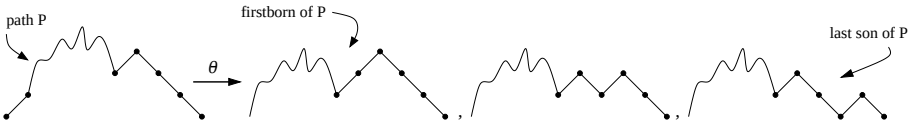

Figure 7: "firstborn" of a path P.

Clearly the "firstborn" of $P^n_{\max}$ can be generated simply overturning its peak. Then we generate all "firstborn" paths on the longest branch of $\mathcal{D}_n$-tree applying $(n-2)$ times the following operation *op1*:

op1: Take off the first and the last path's step, then insert a peak in the last point of the last descent (see Figure 8).

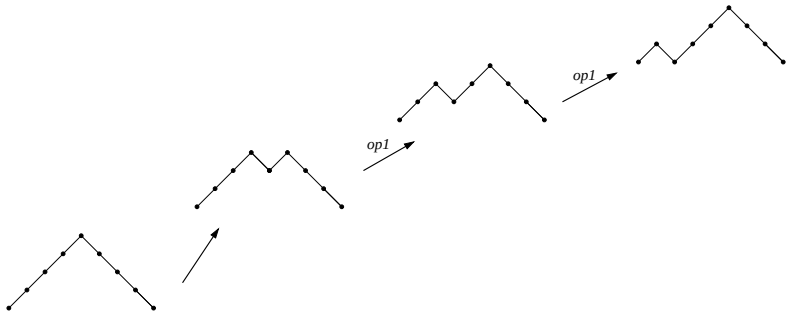

Figure 8: action of *op1*.

When *op1* is no more applicable, i.e. when we arrive at a leaf, we generate leaf's brothers in such a way that they appear according to the decreasing lengths of their last descent. It is easy to see that the following operation *op2* on the last generated path can be performed:

op2: Overturn the rightmost peak in the path (see Figure 9).

In order to explain that *op2* realizes our purpose, we observe that if $Y_i \in \theta(X)$ with $1 \leq i \leq k-1$ (where $k = |\theta(X)|$), then $op2(Y_i) = Y_{i+1} \in \theta(X)$.

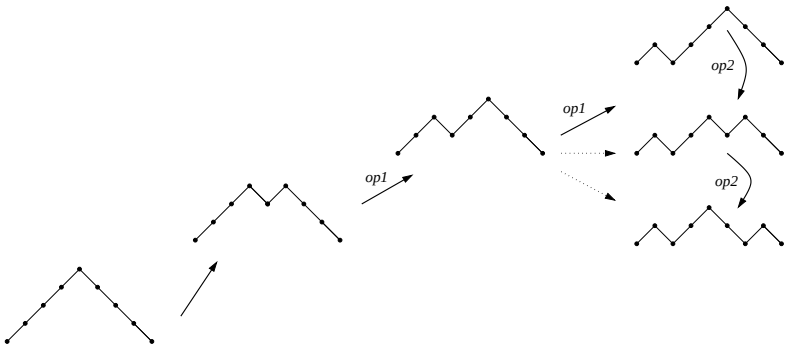

Figure 9: action of *op2*.

Indeed, Y_{i+1} is generated from X by means of θ taking off the first and the last step and inserting a peak in the $(k-i+2)th$ point of X's last descent: this is equivalent to the turnover of the rightmost peak in Y_i.

When *op2* can not be applied anymore, we should visit the $\mathcal{D}_n$-tree's preceding level, in other words we should generate the immediately next brother of X, if it exists. We use the following operation *op3* to generate the "uncle" of the last obtained path:

op3: Take off the rightmost p_1, then insert a northeast step at the beginning of the path and a southeast step in the second-last point of the last ascent (see Figure 10).

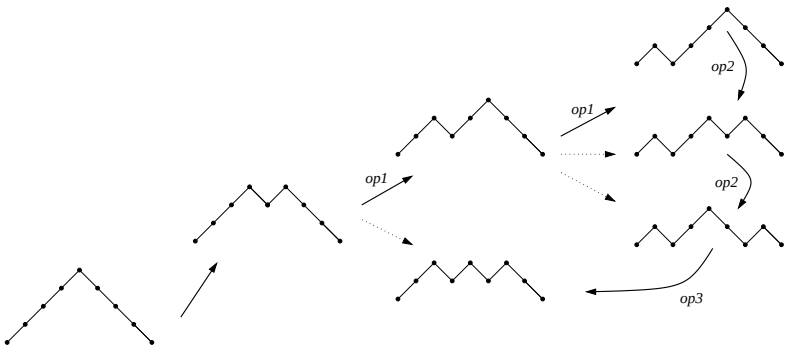

Figure 10: action of *op3*.

Operation *op3*, in plain words, allows to obtain the "uncle" of a path ending with pyramid p_1. This is a crucial point since in this way the visit of a new subtree can start by performing again *op1* and *op2*.

If we denote P as the last son of a path P_i and P_{i+1} as the brother of P_i, we have to prove that $op3(P) = P_{i+1}$. This is easily seen by observing that P_{i+1} is simply obtained from P_i by overturning its rightmost peak (Figure 11)

and that *op3* applied on a path P (ending with a pyramid p_1) is a shortcut for going back in the tree to the father of P (which is P_i) and then overturning its rightmost peak (Figure 12).

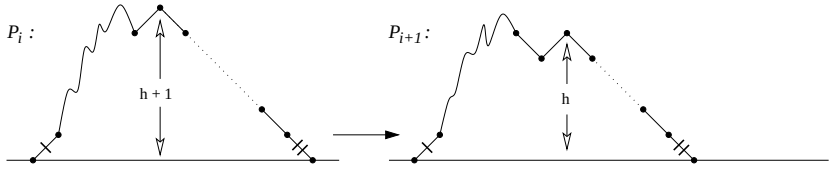

Figure 11: P_{i+1} from P_i.

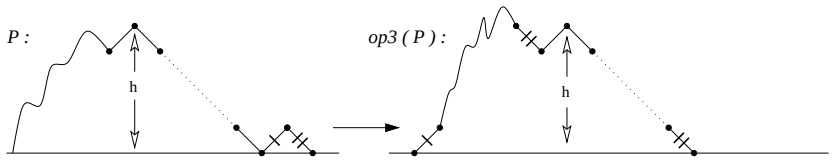

Figure 12: $op3(P) = P_{i+1}$.

The visit of $\mathcal{D}_n$-tree can now be completely conducted thanks to *op1*, *op2* and *op3*. In Figure 13 the visit of $\mathcal{D}_4$-tree is depicted. The dashed lines represents the edges which derive from the action of θ operator, the straight lines are the ones to follow in order to visit the tree, according to the three operations we described above. Finally, the visit of the tree can be summarized in **Algorithm 1**.

Algorithm 1

start with $P^n_{\max}$;
generate the firstborn son of $P^n_{\max}$ *overturning its peak*;
$P :=$ *firstborn son of* $P^n_{\max}$;
while $P \neq$ *the last son of* $P^n_{\max}$ **do**
  if it is possible **then**
    $P' := op1(P)$
  else if it is possible **then**
    $P' := op2(P)$
  else
    $P' := op3(P)$
  end if;
  $P := P'$;
end while

Remark: From Figure 13 we note that the same operation can be consecutively applied more than one time. In particular, *op3* can be applied at most two consecutive times. Indeed, we can have only two possibilities:

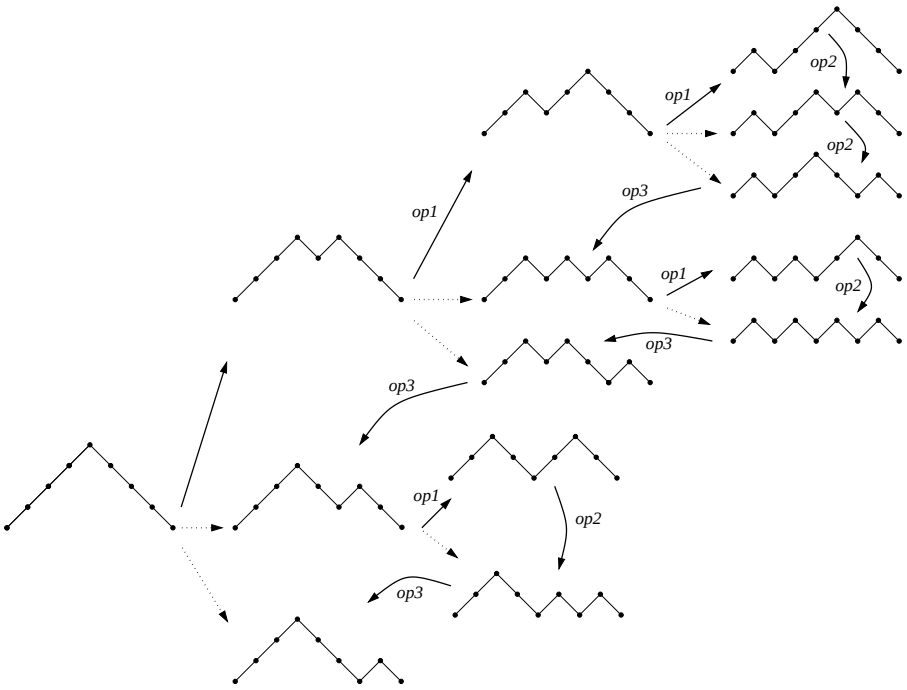

Figure 13: visit of $\mathcal{D}_4$-tree.

a) The path ends in p_1 which is preceded by a peak with height $h \geq 2$. Then, *op3* works only one time because its application (see Figure 14) generates a path that has the last peak with height $h \geq 2$.

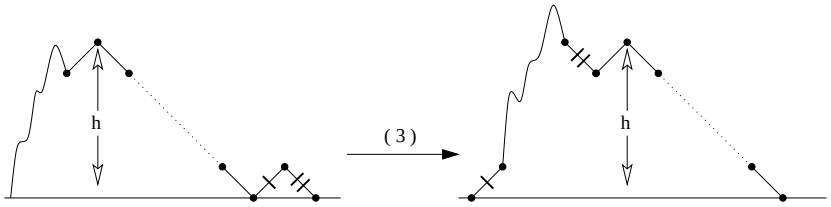

Figure 14: *op3* is performed one time.

b) The path ends in at least two p_1. Then, the application of *op3* generates a path that ends again in p_1; this is case **a)** and the application of *op3* is possible only another time (see Figure 15).

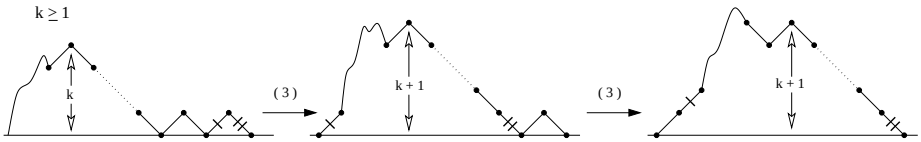

Figure 15: *op3* is performed two times.

3.3 Analysis of the algorithm

Our aim is to realize an algorithm with a constant number of mean operations, while each object in $\mathcal{D}_n$ is generated. It is suitable to associate to each path a binary word by coding with 1 a northeast step and with 0 a southeast. It is clear that the three operations are characterized by a constant number of actions which exchange steps in the path. Indeed, we represent the word by a circular array where the last position is followed by the first one; we introduce a pointer to the first position of the array which always corresponds to the first step of the path (see Figure 16). Operation *op1* is equivalent to exchange the first bit

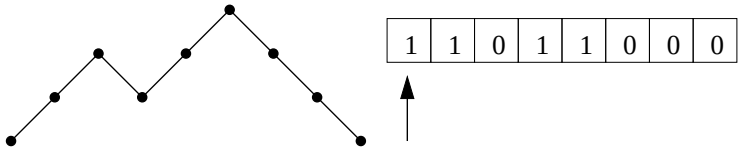

Figure 16: circular array.

1 of the path with the first bit 0 of its last descent and then to move forward the pointer one position (the action of *op1* on the array is illustrated in Figure 17). Operation *op2* is equivalent to exchange the bits of the last sequence 10 in

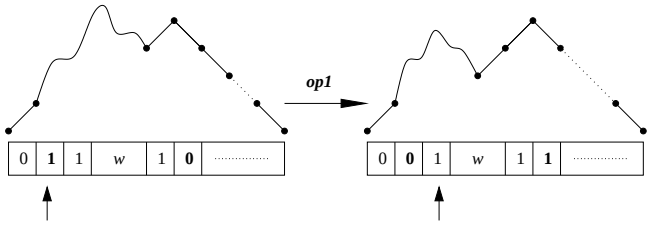

Figure 17: action of *op1* on the array.

the array, while the pointer doesn't move (see Figure 18).

Finally, *op3* is equivalent to exchange the bits of the last and second-last pairs 10 and then to move backward the pointer one position (see Figure 19). Clearly three operations require a constant number of actions independently of the length of the paths and **Algorithm 1** is a constant amortized time (CAT) algorithm.

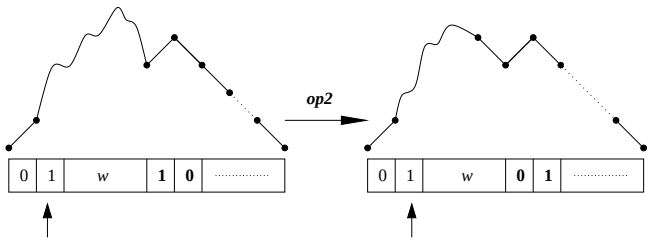

Figure 18: action of *op2* on the array.

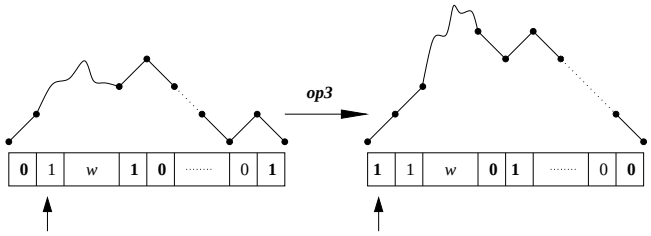

Figure 19: action of *op3* on the array.

4 Conclusions

The main features of the procedure we presented in this work lie in the fact that it uses directly the combinatorial objects (Dyck paths) and it generates all the paths $\in \mathcal{D}_n$, with fixed size n, without using the objects with smaller size. Our approach presents two different strategies which are closely connected. The former can be described by a rooted tree and the latter uses three operations which can be suitably arranged in order to visit all nodes of the tree. Moreover, this visit can be coded in **Algorithm 1** which is a constant amortized time algorithm.

Our research has proved that the basic idea of this algorithm allows to obtain similar results for other classes of well-known paths, like Grand Dyck ($\mathcal{G}_n$) and Motzkin ($\mathcal{M}_n$) paths. Actually, it is possible to obtain all paths of $\mathcal{G}_n$ or $\mathcal{M}_n$ using operations very similar to *op1*, *op2* and *op3*.

Moreover, it is reasonable to think that this method could be applied to other kinds of paths or to other combinatorial classes. Some interesting results should be reached by studying the generation of objects which are in bijections with those ones here considered, as some classes of polyominoes or pattern avoiding permutations enumerated by Catalan, Motzkin or Grand Dyck numbers (for definitions see for example [2]).

References

[1] S. Bacchelli, E. Barcucci, E. Grazzini and E. Pergola, *Exhaustive*

generation of combinatorial objects by ECO, Acta Informatica, **40** (2004), 585–602.

[2] E. Barcucci, A. Del Lungo, E. Pergola and R. Pinzani, *ECO: A Methodology for the Enumeration of Combinatorial Objects*, Journal of Difference Equations and Applications **5** (1999), 435–490.

[3] F. Bergeron, G. Labelle and P. Leroux, *Théorie des espèces et combinatorie des structures arborescentes*, Publications du LACIM 20, Université du Québec à Montréal (1994).

[4] P.J. Chase, *Combination generation and graylex ordering*, Congressus Numeratium, **69** (1989), 215–242.

[5] A. Del Lungo, A. Frosini and S. Rinaldi, *ECO method and the exhaustive generation of convex polyominoes*, In: C.S. Calude, M.J. Dinneen, V. Vajnovszki (eds.) DMTCS 2003, Lecture Notes in Computer Sciences, **2731** (2003), 129–140.

[6] S. Dulucq, *Some combinatorial and algorithmic problems in biology*, In: F. Rechenmann (eds.) Septièmes Entretiens Jacques Cartier (1994).

[7] I. Dutour et J.M. Fédou, *Grammaires d'objets*, rapport Labri, 383-94 Université Bordeaux I (1994).

[8] P. Eades and B. McKay, *An algorithm for generating subsets of fixed size with a strong minimal change property*, Information Processing Letters, **19** (1984), 131–133.

[9] M. Gardner, *Curious properties of the Gray code and how it can be used to solve puzzles*, Scientific American, **227** (2) (1972), 106–109.

[10] F. Gray, *Pulse code communication*, U.S. Patent 2632058 (March 1953).

[11] S.M. Johnson, *Generation of permutations by adjacent traspositions*, Mathematics of Computation, **17** (1963), 282–285.

[12] J.T. Joichi, D.E. White and S.G. Williamson, *Combinatorial Gray codes, SIAM Journal on Computing*, **95** (1) (1980), 130–141.

[13] C.N. Liu and D.T. Tang, *Algorithm 452, enumerating M out N objects*, Comm. ACM, **16** (1973), 485.

[14] J.M. Lucas, D. Roelants van Baronaigien and F. Ruskey, *On rotations and the generation of binary trees*, Journal of Algorithms, **15** (3) (1993), 343–366.

[15] J.E. Ludman, *Gray code generation for MPSK signals*, IEEE Transactions on Communications, **COM-29**, (1981), 1519–1522.

[16] D. Richards, *Data compression and Gray-code sorting*, Information Processing Letters, **22** (4) (1986), 210–215.

[17] J. Robinson and M. Cohn, *Counting sequences*, IEEE Transactions on computers, **C-30** (1981), 17–23.

[18] F. Ruskey, *Adjacent interchange generation of combinations*, Journal of Algorithms **9** (1988), 162–180.

[19] F. Ruskey and J. Sawada, *An efficient algorithm for generating necklaces with fixed density*, SIAM J. Computing **29** (2) (1999), 671–684.

[20] C. Savage, *A survey of combinatorial Gray codes*, SIAM Rev., **39** (4) (1997), 605–629.

[21] H.F. Trotter, *PERM (Algorithm 115)*, Comm. ACM, **5** (8) (1962), 434–435.

[22] V. Vajnovszki, *Generating a Gray code for P-sequences*, The Journal of Mathematical Modelling and Algorithms, **1** (1) (2002), 31–41.

[23] V. Vajnovszki, *Le codages des arbres binaires*, Computer Science Journal of Moldova, **3** (2) (1995), 194–209 (Mathematical Reviews 1485353).

[24] V. Vajnovszki, *Gray visiting Motzkins*, Acta Informatica, **38** (2003), 793–811.

[25] J. West, *Generating trees and the Catalan and Schröder numbers*, Discrete Math. **146** (1995), 247–262.

[26] J. West, *Generating trees and forbidden subsequences*, Discrete Math., **157** (1996), 363–374.