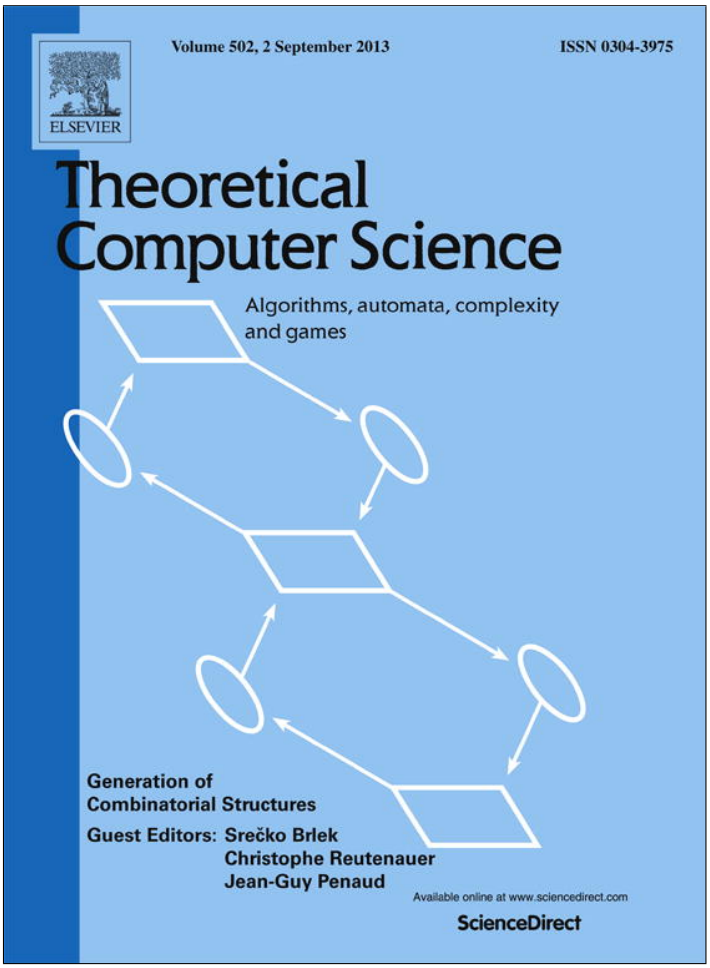


Theoretical Computer Science 502 (2013) 153–164

Contents lists available at ScienceDirect

Theoretical Computer Science

journal homepage: www.elsevier.com/locate/tcs

On the exhaustive generation of plane partitions☆

R. Mantaci [a], P. Massazza [b,*]

[a] LIAFA, CNRS UMR 7089, Université Paris Diderot - Paris 7, Case 7014, 75205 Paris Cedex 13, France
[b] Università degli Studi dell'Insubria, Dipartimento di Scienze Teoriche e Applicate, Via Mazzini 5, 21100 Varese, Italy

ARTICLE INFO

Keywords:
Integer partitions
Exhaustive generation
CAT algorithms

ABSTRACT

We present a CAT (Constant Amortized Time) algorithm for generating all plane partitions of an integer n, that is, all integer matrices with non-increasing rows and columns having sum n.

1. Introduction

Integer partitions of a positive integer n are well known and widely studied combinatorial objects. Defined as non increasing sequences of integers (called parts of the partition) whose sum is n, they can be found in several contexts and problems of bijective combinatorics.

Plane partitions are their natural two-dimensional generalization. A plane partition of an integer n is indeed a matrix $a = (a_{i,j})$ with non-increasing rows and columns ($a_{i,j} \geq a_{i,j+1}$ and $a_{i,j} \geq a_{i+1,j}$) and such that $\sum_{i,j} a_{i,j} = n$. When necessary, we will use the term *linear* partition to denote ordinary partitions, in order to distinguish them from plane partitions.

The number of linear partitions (and *a fortiori* of plane partition) grows exponentially with respect to n. It is hence imperative to have efficient algorithms if one needs to generate them all; more precisely, algorithms which run in constant amortized time (CAT).

Algorithms for the exhaustive generation of linear partitions are known since the 18th-century [1]. There have been deep investigations of the subject, providing several efficient algorithms (see, for instance, Section 7.2.1.4 of [2] and [3,4]). Quite surprisingly, the same problem for plane partitions seems to be much harder. Indeed, no CAT algorithm for generating plane partitions seems to be known (for instance, the problem is missing in [2,4]), although efficient algorithms for generating particular classes of integer matrices exist (a CAT algorithm for generating Young Tableaux is given in [4]).

On the other hand, together with some other researchers, the authors have been recently interested in the exhaustive generation [5] of the accessible configurations of a discrete dynamical system called the Ice Pile Model ($\mathrm{IPM}_k(n)$), describing the evolution of a granular environment (such as heaps of sand grains or snow flakes). Here, n is the number of "grains" composing the system and k a parameter expressing the ability of the grains to slide on each other. In this system, grains are stacked in adjacent columns (initially, they are all in the first column) and, under certain conditions, a grain on top of a column can be moved to the top of another column on the right.

Such configurations can be represented by linear partitions of the integer n. The space of all accessible configurations for a fixed n gets larger when k increases and, when k is sufficiently large, it includes all linear partitions of n. Therefore, the algorithm presented in [5] provides an alternative CAT method to generate all linear partitions of an integer n.

Ice piles can also be defined adding one further dimension [6]. In this case, the configurations of the system are represented by plane partitions. When the authors started looking for a CAT algorithm for the exhaustive generation of

☆ Partially supported by Project M.I.U.R. PRIN 2007–2009: Mathematical aspects and forthcoming applications of automata and formal languages.
* Corresponding author. Tel.: +39 0332218932; fax: +39 0332218919.
E-mail addresses: mantaci@liafa.fr (R. Mantaci), paolo.massazza@uninsubria.it (P. Massazza).

0304-3975/$ – see front matter © 2012 Elsevier B.V. All rights reserved.
doi:10.1016/j.tcs.2012.11.013

bidimensional ice piles, it became apparent that it would have been useful to start from a CAT algorithm for generating plane partitions. This paper does exactly that.

Thus, we design an algorithm for generating plane partitions by properly interleaving several CAT processes (one for each row) which generate linear partitions satisfying suitable conditions. Indeed, each row defines a linear partition which "covers" the partition identified by the row below (columns are non-increasing from top to bottom). This naturally leads to a binary relation between linear partitions that we have to consider when generating all possible values for a row. Hence, in Section 2 we deal with covering partitions and the associated generating problem, showing that a CAT algorithm for covering partitions exists.

It is straightforward to see that a plane partition can be classified according to the linear partition of n whose parts are obtained by summing all elements on each row. We call this the *horizontal projection* of the plane partition. Therefore, the Massazza–Radicioni CAT algorithm [5] can be used to generate all possible horizontal projections. The problem then boils down to generating all plane partitions having a fixed linear partition π as horizontal projection. Section 3 illustrates the solution of that subproblem.

A generation tree associated with a given horizontal projection is defined, where the number of levels is equal to the number of rows and each node corresponds to a CAT process which generates covering partitions. So, a recursive algorithm which traverses the tree can solve the problem. We provide the pseudocode together with a description of the data structure used to represent plane partitions.

Lastly, the complexity of the algorithm is analysed in Section 4, where it is shown that the proposed algorithm is CAT.

2. Preliminaries

2.1. Linear partitions

A linear partition of n is a non-increasing sequence of nonnegative integers with sum n. We indicate by $\mathrm{LP}(n)$ the set of the linear partitions of n and by $\mathrm{SLP}(n)$ the subset consisting of strictly decreasing sequences. For any $x, p \in \mathbb{N}$, with $p > 0$, we denote by $x^{[p]}$ the sequence $(\underbrace{x, \ldots, x}_{p})$ and by $\cdot$ the *catenation product* of sequences. The *length* and the *weight* of $s = (s_1, \ldots, s_l) \in \mathrm{LP}(n)$ are $l(s) = l$ and $w(s) = n$, respectively. Note that s can be written as $s = s_1^{[m_1]} \cdot s_{j_2}^{[m_2]} \cdot \ldots \cdot s_{j_k}^{[m_k]}$ with $s_1 > s_{j_2} > \cdots > s_{j_k}$, that is, s is univocally determined by a pair $(v, \mathbf{m})$ with $v \in \mathrm{SLP}(n_1)$, $n_1 \leq n$ and $\mathbf{m} \in \mathbb{N}^{l(v)}$. The following notations are useful when dealing with sequences,

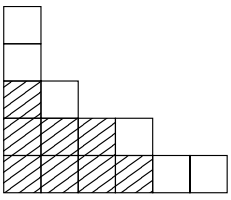
- $s_{<i} = (s_1, \ldots, s_{i-1})$;
- $s_{\leq i} = (s_1, \ldots, s_i)$;
- $s_{>i} = (s_{i+1}, \ldots, s_l)$;
- $s_{\geq i} = (s_i, \ldots, s_l)$.

The set $\mathrm{LP}(n)$ can be ordered with respect to the *negative lexicographic* order, nlex for short, defined as follows.

Definition 1. Let $s, t \in \mathrm{LP}(n)$. Then $s <_{\mathrm{nlex}} t$ if and only if there is i such that $s_{<i} = t_{<i}$ and $s_i > t_i$.

We consider a binary relation between linear partitions, called *covering*.

Definition 2. Let $s \in \mathrm{LP}(n)$ and $t \in \mathrm{LP}(m)$. We say that s *covers* t, denoted as $s \rhd t$, if and only if for all $j \geq 1$ one has $s_j \geq t_j$.



$s = (5, 3, 2, 2, 1, 1)$ covers $t = (3, 2, 2, 1)$

Obviously, if $s \rhd t$ then $n \geq m$. For any $t \in \mathrm{LP}(m)$, the set of the linear partitions of n covering t is $\mathrm{LP}(n|t) = \{s \in \mathrm{LP}(n) | s \rhd t\}$. We can think of $\mathrm{LP}(n|t)$ as the set of reachable states of a simple discrete dynamical system which can be used to describe how ice grains move on a hillside. The initial state of the system consists of one column of ice grains at the top of the hill. Then the system evolves and the grains move to the first position to the right with a lower potential. More formally, we define a partial function *Move* with parameters t, s and i, denoting the profile of the hillside, the profile of the ice pile and the position of the grain to move (see Fig. 1), respectively.

Definition 3. Given $t \in \mathrm{LP}(m)$, $s \in \mathrm{LP}(n|t)$ and $i \in \mathbb{N}$, let k be the smallest integer greater than i such that $s_i - s_k \geq 2$ and $s_i - s_h = 1$ for all h with $i < h < k$ (if such an integer does not exist k is undefined, $k = \perp$). Then

$$\mathrm{Move}(s, t, i) = \begin{cases} s_{<i} \cdot (s_i - 1, s_{i+1}, \ldots, s_{k-1}, s_k + 1) \cdot s_{>k} & \text{if } k \neq \perp \wedge s_i > t_i \\ \perp & \text{otherwise.} \end{cases}$$

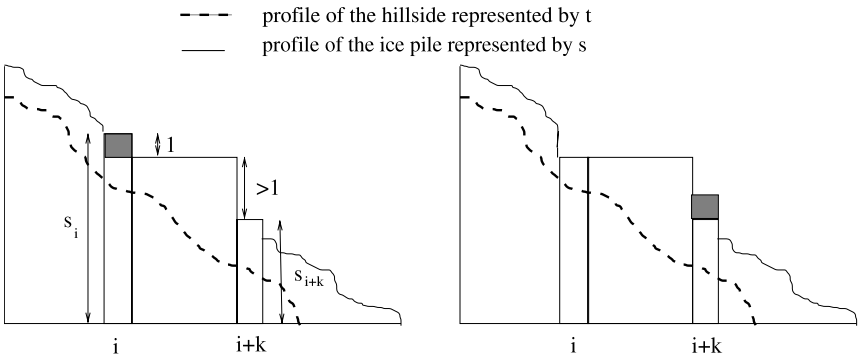


Fig. 1. Graphical representation of a move.

Note. The definition of Move, as well as its name itself, are directly inspired by the Ice Pile Model mentioned in the introduction and in which grains move from one column of the pile to the other. In that context, it is possible to provide a graphical description of a Move and the way it affects the partition s (the parts of s correspond to the height of the columns of the pile).

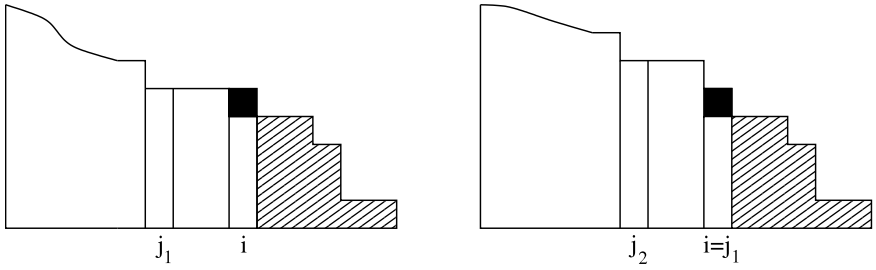
Whenever t is clear from the context, we write $s \stackrel{i}{\Rightarrow} v$ if $v = \text{Move}(s, t, i)$. For any $s \in \text{LP}(n|t)$, the set of linear partitions obtained from s is

$$G(s|t) = \{v \in \text{LP}(n|t) \mid \exists i_1, i_2 \ldots, i_r \in \mathbb{N} \text{ s.t. } s \stackrel{i_1}{\Rightarrow} w \stackrel{i_2}{\Rightarrow} \cdots \stackrel{i_r}{\Rightarrow} v\}.$$

Given $t \in \text{LP}(m)$, it is immediate to see that $\min_{<_{\text{nlex}}}(\text{LP}(n|t)) = (t_1 + n - m) \cdot t_{>1}$. Moreover, the following lemma states that $\text{LP}(n|t)$ is the set of states which can be reached from $\min_{<_{\text{nlex}}}(\text{LP}(n|t))$.

Lemma 1. *Let $t \in LP(m)$. Then, for any $n \geq m$ one has $LP(n|t) = G(s|t)$ where $s = \min_{<_{nlex}}(LP(n|t))$.*

Proof. Suppose $G(s|t) \subsetneq \text{LP}(n|t)$ and let $v = \min_{<_{\text{nlex}}}(\text{LP}(n|t) \setminus G(s|t))$. Consider the largest integer i such $v_i > t_i$. If $i = 1$ one has $v_{>1} = s_{>1}$ and then $v = s$. Otherwise, let $j_1 = \min\{k \leq i | v_k = v_i\}$ and $j_2 = \min\{k < j_1 | v_k = v_{j_1 - 1}\}$.



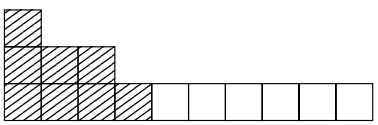
Consider the linear partition $w = v_{<p} \cdot (v_p + 1, v_{p+1}, \ldots, v_{i-1}, v_i - 1) \cdot v_{>i}$ where either $p = j_1$ (if $j_1 < i$) or $p = j_2$ (otherwise) and notice that $w \in \text{LP}(n|t)$ and $w <_{\text{nlex}} v$. This implies $w \in G(s|t)$. Lastly, the contradiction $v \in G(s|t)$ follows from $w \stackrel{p}{\Rightarrow} v$. □

The *set of moves* of $s \in \text{LP}(n|t)$ is defined as the set of integers $\text{M}(s|t) = \{i | \text{Move}(s, t, i) \neq \perp\}$. In $\text{LP}(n|t)$ there is only one linear partition v such that $M(v|t) = \emptyset$, called the *fixed point* of $\text{LP}(n|t)$. This partition obviously belongs to $\text{G}(s|t)$ for any $s \in \text{LP}(n|t)$ and is characterized as follows.

Lemma 2. *Let $t \in LP(m)$ and $v \in LP(n|t)$. If $M(v|t) = \emptyset$ then $v = t \cdot 1^{[n-m]}$.*

Proof. Obviously, if $\{i | i \leq l(t) \wedge v_i > t_i > 0\} \neq \emptyset$ then $M(v|t) \neq \emptyset$. Therefore, one has $v = t \cdot z$ with $z \in \text{LP}(n - m)$ and $M(z|0^{[n-m]}) = \emptyset$. This holds if and only if $z = 1^{[n-m]}$. □

The following picture shows an example of a fixed point.



the fixed point of LP(14|(3, 2, 2, 1))

Lemma 2 implies that the tail of $s \in \text{LP}(n|t)$ beyond the rightmost possible move, is equal to that of t, followed by some 1s, as shown in Fig. 2. Thus, one has:

Corollary 1. *Let $t \in LP(m)$, $s \in LP(n|t)$ and $i = \max(M(s|t))$. Then, there is $c \geq 0$ with $c \leq n - m$, such that $s_{>i} = t_{>i} \cdot 1^{[c]}$.*

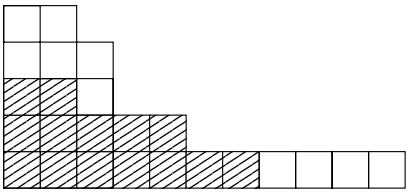


Fig. 2. The tail beyond the rightmost move.

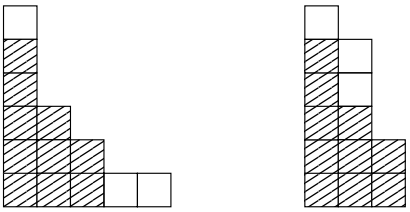


Fig. 3. $(6, 3, 2, 1, 1) \in \mathrm{LP}(13|(5, 3, 2))$ and its grand ancestor.

Among the elements of $\mathrm{LP}(n|t)$, there are some partitions which play a special role and are defined as follows.

Definition 4. Let $t \in \mathrm{LP}(m)$, $s \in \mathrm{LP}(n|t)$ and $i = \max(\mathrm{M}(s|t))$. The *grand ancestor* of s is the linear partition

$$A_t^n(s) = \min_{<_{\mathrm{nlex}}}\{v \in \mathrm{LP}(n|t) | i \in \mathrm{M}(v|t) \wedge v_{\leq i} = s_{\leq i}\}.$$

The characterization of grand ancestors is given in

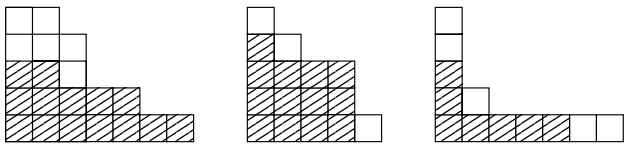
Lemma 3. *Let $t \in LP(m)$, $s \in LP(n|t)$ and $i = \max(M(s|t))$. Then, there are p, q with $p \geq 0$ and either $q = 0$ or $t_{i+p+1} \leq q < s_i - 1$ such that $A_t^n(s)$ is the linear partition*

$$v = s_{\leq i} \cdot (s_i - 1)^{[p]} \cdot q \cdot t_{>i+p+1}.$$

Proof. Choose p as the largest integer such that $\sum_{j=1}^{p}(s_i - 1 - t_{i+j}) \leq w(s_{>i}) - w(t_{>i})$, and let $q = w(s_{>i}) - p(s_i - 1) - w(t_{>i+p+1})$. Then, note that $v \in \mathrm{LP}(n|t)$, $i \in \mathrm{M}(v|t)$ and $v_{\leq i} = s_{\leq i}$. Thus we have only to prove that v is the smallest linear partition with such properties. In fact, suppose there is $z \in \mathrm{LP}(n|t)$ such that $z <_{\mathrm{nlex}} v$, $z_{\leq i} = s_{\leq i}$ and $i \in \mathrm{M}(z|t)$. Let j be the smallest integer such that $z_j > v_j$ (obviously $j > i + p$) and see that $\sum_{e>j} z_e < \sum_{e>j} v_e = \sum_{e>j} t_e$. This implies $z \not\triangleright t$. □

Informally, the grand ancestor of s is obtained by replacing the tail beyond the rightmost move, $s_{>i}$, with the smallest linear partition of height $s_i - 1$ and weight $w(s_{>i})$ which covers $t_{>i}$ (see Fig. 3).

Clearly, one has $A_t^n(s) \leq_{\mathrm{nlex}} s$. In particular it is immediate to see that $s = A_t^n(s)$ if and only if for $i = \max(\mathrm{M}(s|t))$ one has either $s_{>i} = t_{>i}$ or $s_{>i} = t_{>i} \cdot 1$ and $t_{>i} = (s_i - 1)^{[a]}$, with $a > 0$, or $s_i = 2$, $s_{>i} = 1^{[c]}$ and $t_{>i} = 1^{[b]}$, with $b > 0$ and $c > b + 1$. These three cases are illustrated below.



Given $s \in \mathrm{LP}(n|t)$ we denote by s' the successor of s in $\mathrm{LP}(n|t)$ (w.r.t. $<_{\mathrm{nlex}}$). Moreover, we indicate by $T(n, t)$ the tree with nodes in $\mathrm{LP}(n|t)$ and edges identified by the function defined below, which provides the parent of a node.

Definition 5. Let $f : \mathrm{LP}(n|t) \mapsto \mathrm{LP}(n|t) \cup \{\perp\}$ be the function defined as

$$f(s) = \begin{cases} \perp & \text{if } s = \min_{<_{\mathrm{nlex}}}(\mathrm{LP}(n|t)) \\ A_t^n(v) \text{ where } v' = s & \text{otherwise.} \end{cases}$$

Thus, the root of $T(n, t)$ is $\min_{<_{\mathrm{nlex}}}(\mathrm{LP}(n|t)) = (t_1 + n - m) \cdot t_{>1}$. A node of $T(n, t)$ is a grand ancestor if and only if it is an internal node. In particular, s is the parent of s' whenever $s = A_t^n(s)$. Moreover, Lemma 3 implies that the degree of a node in $T(n, t)$ is at most 3. In fact, if there are $i, p > 0$ such that $s = s_{\leq i} \cdot (s_i - 1)^{[p]} \cdot q \cdot t_{>i+p+1}$, with $t_{i+p} < s_i - 1$ and $t_{i+p+1} < q$, then s turns out to be the grand ancestor of the three linear partitions $s_{\leq i} \cdot t_{>i} \cdot 1^{[a]}$, $s_{\leq i+p} \cdot t_{>i+p} \cdot 1^{[b]}$ and $s_{\leq i+p+1} \cdot t_{>i+p+1} \cdot 1^{[c]}$, for suitable $a, b, c > 0$.

Fig. 4 illustrates the tree $T(13, t)$ for $t = (5, 3, 2)$. All the edges denote legal moves, with full edges representing the parent–child relationship (expressed by the function f).

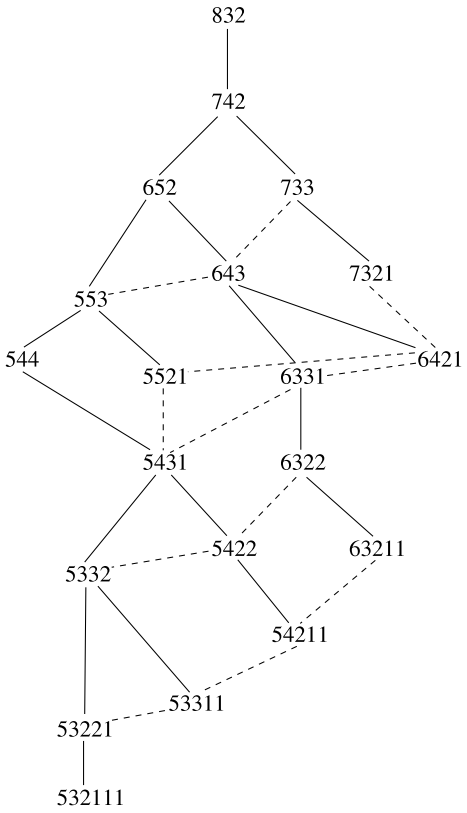


Fig. 4. The tree $T(13, t)$ for $t = (5, 3, 2)$.

2.2. Exhaustive generation of covering partitions

In this section we show how the ordered sequence of elements in $\mathrm{LP}(n|t)$ can be generated by an approach which is similar to the one used in [5] to generate particular linear partitions known as ice piles. In fact, the following lemma states that the successor of s is obtained by moving a grain in the grand ancestor of s.

Lemma 4. *Let $t \in LP(m)$, $s \in LP(n|t)$ and $i = \max(M(s|t))$. Then, one has*

$$s' = Move(A_t^n(s), i).$$

Proof. Let $v = \mathrm{Move}(A_t^n(s), i)$ and suppose $s' <_{\mathrm{nlex}} v$. Clearly, one has $s'_{<i} = v_{<i} = s_{<i}$. If $s'_i = s_i$ then we necessarily have $s'_j < s_j$ for a suitable $j > i$: this leads either to the contradiction $s' \not\triangleright t$ or to $s' \notin \mathrm{LP}(n)$, since Corollary 1 states that $s_{>i} = t_{>i} \cdot 1^{[c]}$ for a suitable $c \geq 0$.
Thus, we have $s'_i = v_i = s_i - 1$ and we consider the smallest $j > i$ such that $s'_j > v_j$, together with the first $k > j$ such that $s'_k < v_k$. By recalling Lemma 3, we have

$$v = s_{<i} \cdot (s_i - 1)^{[p+1]} \cdot (q + 1) \cdot t_{>i+p+1}.$$

Thus, one necessarily has $j > i + p$ and $k > i + p + 1$, getting the contradiction $s' \not\triangleright t$, since $s'_k < t_k$. □

By Definition 5 and Lemma 4, it follows that every node $s \in T(n, t)$ is obtained by moving a grain in its parent. Moreover, Lemma 4 directly leads to Algorithm 1, which iteratively generates the ordered sequence (w.r.t. $<_{\mathrm{nlex}}$) of elements in $\mathrm{LP}(n|t)$.

Algorithm 1 Exhaustive generation of $\mathrm{LP}(n|t)$.

```
1: PROCEDURE LINPARTGEN(n, t)
2: v :=MIN(n, t); {v = min_{<nlex} LP(n|t)}
3: while M(v|t) ≠ ∅ do
4:    i :=MAX(M(v|t));
5:    GRANDANC(v, i, t); {v = A_t^n(v)}
6:    MOVE(v, i, t); {v = Move(v, i, t)}
7:    UPDATEMOVES(M(v|t));
8: end while
```

Example 1. The ordered sequence generated by LinPartGen(13, (5, 3, 2)) (see Fig. 4) is

832, 742, 733, 7321, 652, 643, 6421, 6331, 6322, 63211, 553,
5521, 544, 5431, 5422, 54211, 5332, 53311, 53221, 532111.

Note that this corresponds to a preorder traverse of $T(13, (5, 3, 2))$ if we consider it as an ordered tree and set $v < z$ for any two nodes v, z with the same parent, $s = f(v) = f(z)$, such that $v = \text{Move}(s, i)$ and $z = \text{Move}(s, j)$ with $i > j$.

With respect to the time and space complexity, we have the following result.

Theorem 1. *LinPartGen(n, t) runs in time $O(\sharp LP(n|t))$ using $O(\sqrt{n})$ space.*

Proof. We provide here an outline (see [5] for full details). First, let us consider the space complexity and note that the algorithm needs space for t and for the latest generated element in $\text{LP}(n|t)$. These are linear partitions which can be easily represented by two linked lists of length $O(\sqrt{n})$, (see Section 3.1 of this paper). Moreover, the set $\text{M}(v|t)$ can be represented as an ordered stack of $O(\sqrt{n})$ links to nodes in the list of v (with the link to the node corresponding to the rightmost move in v at the top). Hence the overall space is $O(\sqrt{n})$.

Then, the time complexity can be determined by noting that

- the number of iterations is $\sharp\text{LP}(n|t) - 1$;
- the symmetric difference $\text{M}(s'|t) \diamond \text{M}(s|t)$ is a set with $O(1)$ elements;
- Max, Move and UpdateMoves can be easily implemented such that they always run in time $O(1)$;
- the construction of the grand ancestor made by GrandAnc(v, i, t) runs in $O(d)$ where d is the distance in $T(n, t)$ between v and $A_t^n(v)$;
- if v, w are leaves of $T(n, t)$ then the two paths p_v, p_w, from v to $A_t^n(v)$ and from w to $A_t^n(w)$, respectively, do not have common edges.

So, the generation of $\text{LP}(n|t)$ by means of Algorithm 1 requires time

$$\sum_{i=1}^{\sharp\text{LP}(n|t)-1} O(1) + \sum_{v \text{ is a leaf}} \text{length}(p_v) = O(\sharp\text{LP}(n|t)) + O(\sharp\text{LP}(n|t)) = O(\sharp\text{LP}(n|t)). \quad \square$$

3. The main problem

A plane partition of n is a matrix a of nonnegative integers that are non-increasing from left to right and from top to bottom and such that their sum is exactly n,

$$a_{i,j} \geq a_{i+1,j}, \quad a_{i,j} \geq a_{i,j+1}, \qquad \sum_{i,j} a_{i,j} = n.$$

We denote by $a^{(i)}$ the i-th row of a and by $\text{PP}(n)$ the set of the plane partitions of an integer n. The *horizontal projection* of $a \in \text{PP}(n)$ is defined as the sequence of integers $\text{hp}(a) = (s_1, \ldots, s_l)$ where $s_i = \sum_j a_{i,j}$. It is immediate to verify that

- $\text{hp}(a) \in \text{LP}(n)$;
- if $s_i = s_j$ then $a^{(i)} = a^{(j)}$;
- $a^{(1)} \rhd a^{(2)} \rhd \cdots \rhd a^{(l)}$.

Given $s \in \text{LP}(n)$, the set of plane partitions with horizontal projection s is $\text{PP}_s(n) = \{a \in \text{PP}(n) | \text{hp}(a) = s\}$. Clearly, one has $\text{PP}(n) = \bigcup_{s \in \text{LP}(n)} \text{PP}_s(n)$.

We are interested in defining a total ordering on $\text{PP}(n)$ which naturally extends the negative lexicographic on $\text{LP}(n)$. Thus, we have:

Definition 6. Let $a, b \in \text{PP}(n)$. We let $a < b$ if and only if either $\text{hp}(a) <_{\text{nlex}} \text{hp}(b)$ or $\text{hp}(a) = \text{hp}(b)$ and there is k such that $a^{(k)} <_{\text{nlex}} b^{(k)}$ and $a^{(i)} = b^{(i)}$ for $i > k$.

Then, we consider the following problem:

Problem: Plane Partition Generation (PPG)
Input: an integer n;
Output: the ordered sequence of plane partitions in $\text{PP}(n)$.

Our approach is that of producing the sequence

$$[n], [n-1, 1], \ldots, [1, \ldots, 1], \begin{bmatrix} n-1 \\ 1 \end{bmatrix}, \ldots, \begin{bmatrix} 2 \\ 1 \\ \vdots \\ 1 \end{bmatrix}, \begin{bmatrix} 1 & 1 \\ 1 & 0 \\ \vdots & \vdots \\ 1 & 0 \end{bmatrix}, \begin{bmatrix} 1 \\ \vdots \\ 1 \end{bmatrix}$$

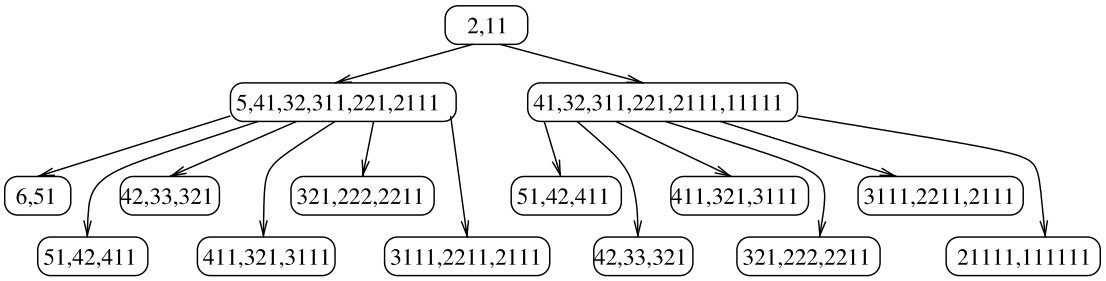


Fig. 5. The generation tree associated with (6, 5, 2).

by a two-steps process:

1. generate the ordered sequence (w.r.t.$<_{\text{nlex}}$) of all $s \in \text{LP}(n)$;
2. for each s, generate the ordered sequence of elements in $\text{PP}_s(n)$.

Given $s = (s_1, \ldots, s_l) \in \text{LP}(n)$, we recall that for $a \in \text{PP}_s(n)$ one has $a^{(1)} \rhd a^{(2)} \cdots \rhd a^{(l)}$ and $a^{(i)} = a^{(j)}$ if and only if $s_i = s_j$. Thus, let $\bar{n} = \sum_{i|s_i > s_{i-1}} s_i \leq n$ and consider the set $\text{SLP}(\bar{n})$ of strictly decreasing sequences of weight $\bar{n}$. Then, we associate with s a sequence $\bar{s} \in \text{SLP}(\bar{n})$ and a multiplicity vector $\mathbf{m} \in \mathbb{N}^{l(\bar{s})}$ such that if a value, say s_k, appears r times in s and is preceded by p different values, then $\bar{s}_{p+1} = s_k$ and $m_{p+1} = r$. In other words, we represent s by the pair $(\bar{s}, \mathbf{m})$ and we reduce the problem of generating $\text{PP}_s(n)$ to the problem of generating all the pairs $(a, \mathbf{m})$ with $a \in \text{PP}_{\bar{s}}(\bar{n})$ (we interpret $\mathbf{m}_i$ as the multiplicity of row $a^{(r)}$, with $r = 1 + \sum_{j<i} \mathbf{m}_j$). This leads to Procedure GeneratePP shown below.

Algorithm 2 Generation of plane partitions

```
1: Procedure GeneratePP(n);
2: s := (n); m := [1];
3: GenerateHProj(n, s, m);
4: while M(s|0^[n]) ≠ ∅ do
5:    s :=Next(s, 0^[n]); {s ← s'}
6:    (n̄, s̄, m) :=StrictDec(s); {s̄ ∈ SLP(n̄)}
7:    GenerateHProj(n̄, s̄, m);
8: end while
```

So, without loss of generality, we suppose that the horizontal projection s is strictly decreasing and we show that the generation of $\text{PP}_s(n)$ (i.e. Procedure GenerateHProj) follows from a recursive scheme which is illustrated by means of the *generation tree* associated with s, defined as follows.

Definition 7. Let $s = (s_1, \ldots, s_l) \in \text{SLP}(n)$. The *generation tree* associated with s is an ordered tree of height $l-1$ such that:

1. the root (level 0) contains the ordered sequence of elements in $\text{LP}(s_l)$ and has $\sharp\text{LP}(s_l)$ children;
2. for all i, with $0 < i < l$, if v is a node at level i with j brothers at its left, then it contains the ordered sequence of the linear partitions in $\text{LP}(s_{l-i}|t)$, where t is the $(j+1)$th element of the sequence contained in the parent of v;
3. if a node is at level i, with $i < l-1$, and contains a sequence of p elements then it has p children.

We define the size $|v|$ of a node v in a generation tree as the length of the sequence contained in v. Fig. 5 illustrates the generation tree associated with the horizontal projection (6, 5, 2). As it is shown hereinafter in Lemma 6, the complexity of Procedure GenerateHProj, used to generate $\text{PP}_s(n)$, depends on the following two properties of generation trees.

Proposition 1. *In a generation tree, any node v which is not the root satisfies the condition $|v| \geq 2$.*

Proof. By definition, each linear partition of the sequence contained in v belongs to $\text{LP}(s_{l-i}|t)$, where t is a suitable linear partition of s_{l-i+1} contained in the parent of v. Hence, since $s_{l-i} > s_{l-i+1}$, one has $|v| = \sharp\text{LP}(s_{l-i}|t) \geq 2$. □

Corollary 2. *Let V_i denote the set of nodes at level i in a generation tree. Then, for any $i > 0$ one has*

$$\sum_{v \in V_i} |v| \geq \sum_{k=0}^{i-1} \sum_{v \in V_k} |v|.$$

Proof. We argue by induction. If $i = 1$, by property 1 in Definition 7 one has

$$\sum_{v \in V_1} |v| \geq \sharp V_1 = \sharp\text{LP}(s_l) = \sum_{k=0}^{1-1} \sum_{v \in V_k} |v|.$$

Thus, let $i > 1$ and suppose that the statement holds for i. By Proposition 1 and property 3 in Definition 7 we have

$$\sum_{v\in V_{i+1}} |v| \geq 2\sharp V_{i+1} = 2\sum_{v\in V_i} |v|.$$

Lastly, by induction hypothesis we get

$$2\sum_{v\in V_i} |v| \geq \sum_{v\in V_i} |v| + \sum_{k=0}^{i-1}\sum_{v\in V_k} |v| = \sum_{k=0}^{i}\sum_{v\in V_k} |v|. \quad \square$$

Note that there is a one to one correspondence between $\mathrm{PP}_s(n)$ and the elements which appear in the sequences contained in the leaves of a generation tree. So, we can easily solve PPG by traversing the generation tree using a method which resembles the preorder visit. The only difference is that a node is visited each time we return from a subtree (p times if it has p children), with the jth visit which sets the corresponding row of the plane partition to the jth element of the sequence associated with the node.

Example 2. With respect to Fig. 5, the visit of the tree outputs the sequence

$$\begin{matrix} 6 \\ 5 \\ 2 \end{matrix}\ ,\ \begin{matrix} 5\,1 \\ 5 \\ 2 \end{matrix}\ ,\ \begin{matrix} 5\,1 \\ 4\,1 \\ 2 \end{matrix}\ ,\ \begin{matrix} 4\,2 \\ 4\,1 \\ 2 \end{matrix}\ ,\ \begin{matrix} 4\,1\,1 \\ 4\,1 \\ 2 \end{matrix}\ ,\ \begin{matrix} 4\,2 \\ 3\,2 \\ 2 \end{matrix}\ ,\ \ldots,\ \begin{matrix} 1\,1\,1\,1\,1\,1 \\ 1\,1\,1\,1\,1 \\ 1\,1 \end{matrix}$$

3.1. The data structures

In order to manipulate a plane partition, we define a data structure consisting of an array of rows, where each row is implemented as a doubly-linked list. Thus, the ith row $a^{(i)} = (a_{i,1}, \ldots, a_{i,j_i})$ is represented by a doubly-linked list having as many nodes as different values in $a^{(i)}$, that is, $l_i = \sharp\{e | a_{i,e} > a_{i,e+1}\}$ nodes. Each node of the ith row contains two integers: a value in $a^{(i)}$ and the index of the rightmost column where it appears in row i. Hence, the sequence of nodes representing $a^{(i)}$

$$(v_{i,1}, k_{i,1}), (v_{i,2}, k_{i,2}), \ldots, (v_{i,l_i}, k_{i,l_i}),$$

satisfies the following relations

- $v_{i,1} > v_{i,2} \cdots > v_{i,l_i}$;
- $v_{i,1} = a_{i,1} = a_{i,k_{i,1}}$;
- for all $e > 1$, $v_{i,e} = a_{i,k_{i,e-1}+1} = a_{i,k_{i,e}}$.

According to this representation, the linear partition (6, 6, 6, 5, 5, 3, 2, 2, 2, 2, 1, 1) is represented by the list of nodes ((6, 3), (5, 5), (3, 6), (2, 10), (1, 12)).
Moreover, each node of the list representing $a^{(i)}$ is also connected to a node of the list representing $a^{(i+1)}$, as follows: $(v_{i,p}, k_{i,p})$ has a link to $(v_{i+1,q}, k_{i+1,q})$ such that $k_{i+1,q} \geq k_{i,p}$ and $k_{i+1,q-1} < k_{i,p}$. In other terms, $(v_{i,p}, k_{i,p})$ has a link connecting it to the node of the $(i+1)$th row having the value $a_{i+1,k_{i,p}}$ as first entry. This is useful to know in time $O(1)$ the value $a_{i+1,j}$ below a value $a_{i,j}$ such that $a_{i,j} > a_{i,j+1}$.

Indeed, each node of the ith row corresponds to a position of the linear partition $a^{(i)}$ where a move may be applied. The only condition to verify to ensure that such a move can indeed be applied is that the observed value is greater than the value immediately below in $a^{(i+1)}$ (in the same column). Fig. 6 illustrates this representation for a non trivial plane partition.

We will implement a plane partition as an array R of links (one for each row). An array of stacks St is also used to represent the set of all possible moves applicable to a given plane partition at a given time. Each stack St[i] (possibly empty) contains the set of the moves for $a^{(i)}$, with the rightmost move always at the top. Each move is stacked as a link to the node (of the list representing a row) corresponding to the column where the move will be applied. Note that when $a^{(i)}$ needs to be initialized as the smallest (w.r.t. to $<_{\mathrm{nlex}}$) linear partition that covers $a^{(i+1)}$, its corresponding stack (St[i]) has to contain exactly one entry, the link to the first node of the ith list. Indeed, at this stage, only a move in column 1 is possible for the ith row.

We will show later how our procedures modify such structures and what is the complexity of such modifications.

3.2. The algorithm

According to Algorithm 2, given a horizontal projection $s \in \mathrm{LP}(n)$ identified by $(\bar{s}, \mathbf{m})$ ($\bar{s} \in \mathrm{SLP}(\bar{n})$, $\bar{n} \leq n$ and $\mathbf{m} \in \mathbb{N}^{l(\bar{s})}$), the visit of the generation tree for $\mathrm{PP}_s(n)$ is done by calling Procedure GenerateHProj($\bar{n}, \bar{s}, \mathbf{m}$) shown below. Note that a plane partition $a \in \mathrm{PP}_s(n)$ will be represented by the pair $(\bar{a}, \mathbf{m})$, where $\bar{a} \in \mathrm{PP}_{\bar{s}}(\bar{n})$ has $l(\bar{s})$ different rows.

The procedure initializes all the data structures used for generating plane partitions, that is, the array R of links to rows, the array M of multiplicities (of rows) and the array St of stacks. A row $l+1$ with value 0 in the first column is also added for technical reasons. Then, the root of the generation tree is created by calling InitRow(s, l), which sets up the last row of the plane partition. Finally, the (recursive) visit is started with the call GenerateRow(l, s).

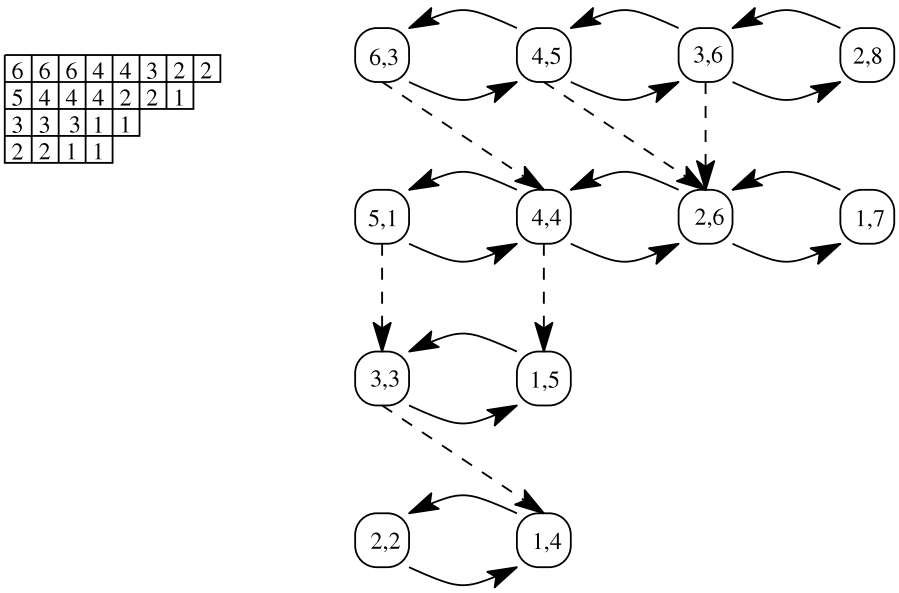


Fig. 6. Representation of a plane partition.

Algorithm 3 Generation of plane partitions with given horizontal projection.

```
1: PROCEDURE GENERATEHPROJ(n, s, m)
2: l :=LENGTH(s);
3: for i := 1 to l do
4:    St[i]:= EMPTYSTACK();
5:    R[i] := NULL;
6:    M[i] := m_i;
7: end for
8: R[l + 1] :=CREATENODE();
9: R[l + 1] →item:= (0, 1);
10: R[l + 1] →next:=NULL;
11: R[l + 1] →pred:=NULL;
12: R[l + 1] →below:=NULL;
13: M[l + 1] := 1;
14: INITROW(l, s);
15: GENERATEROW(l, s);
```

For any i, with $1 \le i \le l(\bar{s})$, the process of initializing the ith row of $\bar{a} \in \mathrm{PP}_{\bar{s}}(\bar{n})$, provided that rows $i+1, i+2, \ldots, l(\bar{s})$ are already properly represented as doubly-linked lists, consists of creating the list representation of the smallest linear partition in $\mathrm{LP}(\bar{s}_i|\bar{a}^{(i+1)})$, that is, $(\bar{a}_{i,1} + \bar{s}_i - \bar{s}_{i+1}, \bar{a}_{i,2}, \ldots, \bar{a}_{i,j_i})$ (see Lemma 1), and assigning the address of the first node to $R[i]$. Moreover, the *below* fields of all the nodes in row i are set to point to the associated nodes in row $i+1$ (as explained in Section 3.1). This is what INITROW$(i, \bar{s})$ does, together with the initialization of the stack of moves of row i. As noted in the previous section, only the move associated with the first node needs to be stacked at this stage.

GENERATEROW$(i, \bar{s})$ assumes that rows $i+1, \ldots, l(\bar{s})$ have a value (i.e. for all j with $i \le j \le l$, $R[j]$ points to the first node of the list representing $\bar{a}^{(j)}$) and that $\bar{a}^{(i)}$ is equal to $\min_{<_{\mathrm{nlex}}}(\mathrm{LP}(\bar{s}_i|\bar{a}^{(i+1)}))$. Then, for each $v \in \mathrm{LP}(\bar{s}_i|\bar{a}^{(i+1)})$, it recursively generates all possible values for the rows $i-1, i-2, \ldots, 1$. Thus, if $i = 1$ (that is, a leaf is being visited) it simply generates the ordered sequence of elements in $\mathrm{LP}(s_1|\bar{a}^{(2)})$. Otherwise (an internal node is being visited), the visit of the subtrees has to be interleaved with the process which generates the linear partitions in $\mathrm{LP}(\bar{s}_i|\bar{a}^{(i+1)})$ (the possible values for the ith row).

4. Complexity

Our algorithm consists of several functions and procedures, which are independently analysed in order to compute their complexity.

Procedure STRICTDEC(s) (used in Algorithm 2) computes the linear partition $\bar{s}$ associated with s (by eliminating repeated entries) and the multiplicity vector $\mathbf{m}$ in time $O(l_s)$, where l_s is the length of the linked list representing s (note that $l_s = l(\bar{s})$). As a matter of fact, INITROW(i, s) (see Algorithm 4) makes a copy of the list representing $a^{(i+1)}$, while possibly adding a new node. Then, it follows that it runs in time $O(l_{i+1})$ where l_{i+1} is the length of the list for $a^{(i+1)}$. For two partitions s and t such that $s \triangleright t$, the boolean function ISBOTTOM(s, t) returns *true* if no move is possible on s. Thus, ISBOTTOM$(a^{(i)}, a^{(i+1)})$ simply tests in time $O(1)$ whether the stack St$[i]$ is empty. Lastly, it is immediate to see that the most important function is NEXT (used

Algorithm 4 Initialization of a row.

```
1: PROCEDURE INITROW(i, s)
2: R[i] :=COPY(R[i + 1]);
3: d := s_i − s_{i+1};
4: (x, y) := R[i] → item;
5: if y = 1 then
6:    R[i] → item = (x + d, y);
7:    t_1 := R[i];
8: else
9:    HEADINSERT(R[i], (x + d, 1));
10:   R[i] → below := R[i + 1];
11:   t_1 := R[i] → next;
12: end if
13: t_2 := R[i + 1];
14: while t_1 ≠ NULL do
15:   t_1 → below := t_2;
16:   t_1 := t_1 → next;
17:   t_2 := t_2 → next;
18: end while
19: PUSH(St[i], R[i]);
```

Algorithm 5 Recursive generation of rows.

```
1: PROCEDURE GENERATEROW(i, s)
2: if i = 1 then
3:    while not ISBOTTOM(a^(1), a^(2)) do
4:       a^(1) := NEXT(a^(1), a^(2));
5:    end while
6: else
7:    while not ISBOTTOM(a^(i), a^(i+1)) do
8:       INITROW(i − 1, s);
9:       GENERATEROW(i − 1, s);
10:      a^(i) := NEXT(a^(i), a^(i+1));
11:   end while
12:   GENERATEROW(i − 1, s);
13: end if
```

in algorithms 2 and 5), that is, the iterator which is used to pass through the ordered sequence of covering linear partitions; the analysis of its complexity is provided in the following lemma.

Lemma 5. *The amortized cost of* NEXT *is* $O(1)$.

Proof. First, note that a call NEXT(s, t) needs to

1. identify the position of the next move m to apply to the partition s;
2. compute the grand ancestor of s;
3. update the current partition, by applying the move m to the grand ancestor of s;
4. update the stack of moves of the partition after the move.

These steps correspond exactly to the four instructions at lines 4–7 of Algorithm 1. Hence, by Theorem 1, we know that $\sharp\mathrm{LP}(n|t) - 1$ iterations of the instruction $s :=$NEXT(s, t), (starting from $s = \min_{<_{\mathrm{nlex}}}(\mathrm{LP}(n|t))$) pass through the ordered sequence of linear partitions in $\mathrm{LP}(n|t)$ in time $O(\sharp\mathrm{LP}(n|t))$. □

For the sake of completeness, it is worthwhile to go into details of steps 1–4 cited above, especially to understand the role played by the data structures of Section 3.1. In particular, our calls to NEXT always have the form NEXT$(a^{(i)}, a^{(i+1)})$, therefore step 1 can be performed simply by consulting the top of the stack St$[i]$.

Step 2 is the only step which possibly needs more than a constant time. Recall that for any $e > 0$, immediately before the eth call NEXT$(a^{(i)}, a^{(i+1)})$ the row $a^{(i)}$ is the eth element of $\mathrm{LP}(s_i|a^{(i+1)})$. Since $a^{(i)}$ and its grand ancestor $A^{s_i}_{a^{(i+1)}}(a^{(i)})$ have the same prefix up to the position $j = \max(\mathrm{M}(a^{(i)}|a^{(i+1)}))$, only the suffix of the grand ancestor needs to be reconstructed. As a direct consequence of Corollary 1 and Lemma 3, this reconstruction can be done in a time proportional to $C = \sharp\{k > j | a^{(i+1)}_k > a^{(i+1)}_{k+1}\}$, that is, the number of nodes representing $a^{(i+1)}_{>j}$. Notice that $w(a^{(i)}_{>j}) - w(a^{(i+1)}_{>j}) \geq 1$ and then Corollary 1 implicitly states that C is smaller than the distance between $a^{(i)}$ and its grand ancestor in $T(s_i, a^{(i+1)})$ (see the proof of Theorem 1).

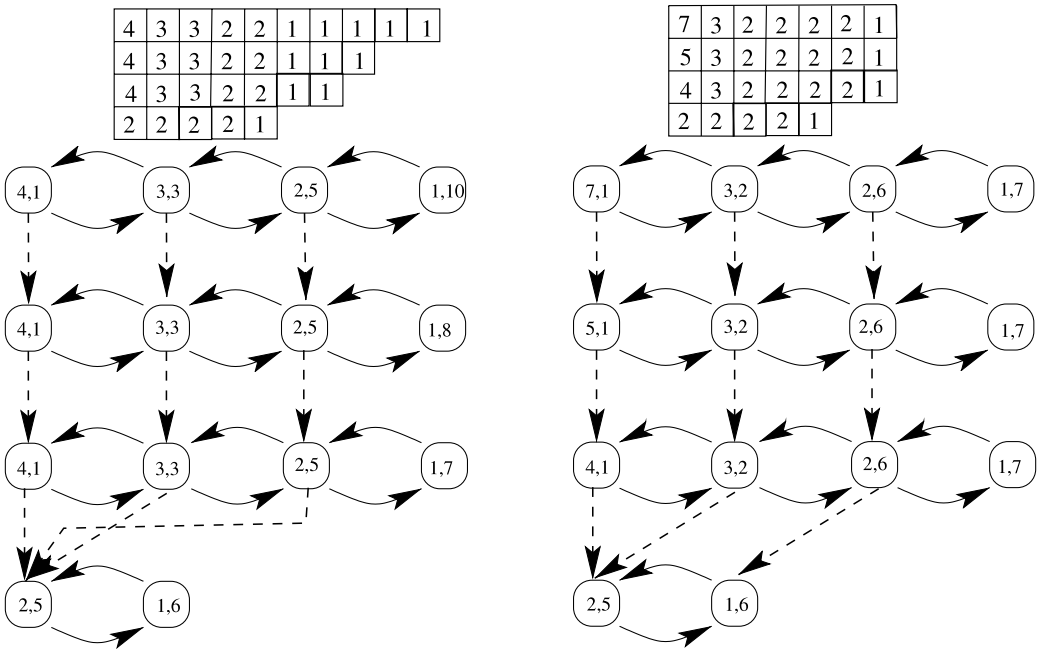


Fig. 7. The move is applied to the third column of the third row. No new node.

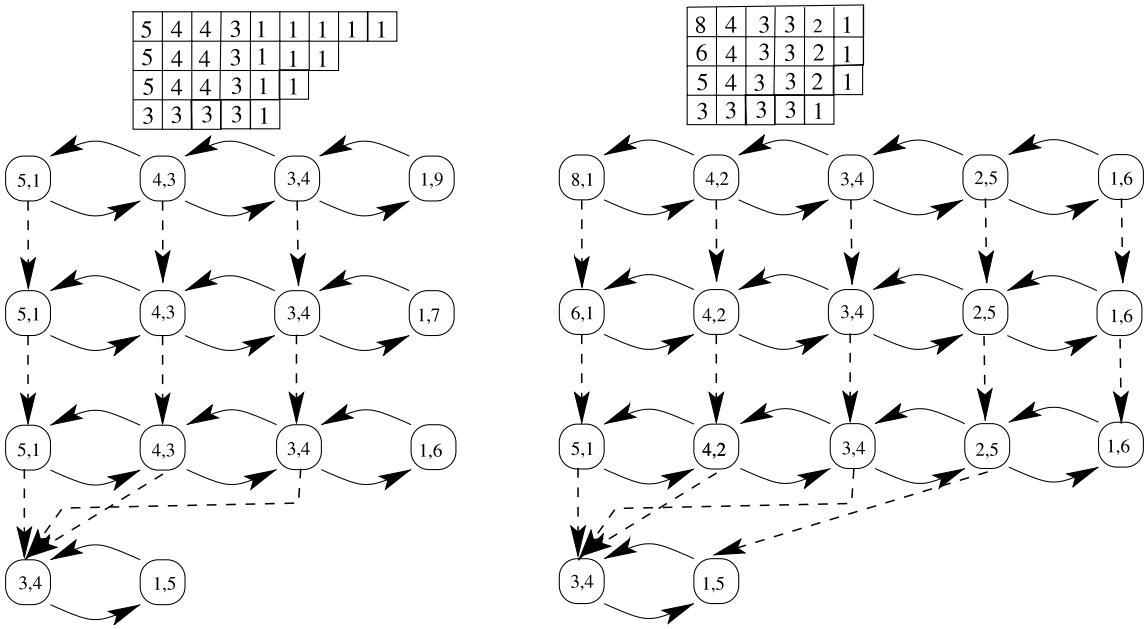


Fig. 8. The move is applied to the third column of the third row. One new node.

Step 3. A move at position j of a linear partition represented as a doubly-linked list, concerns only a small number of nodes in the neighbourhood of the node having its second entry equal to j, and hence can be performed in constant time (note that this would not be true if we had used other data structures such as an array of integers to represent the partition). In pictures 7 and 8, we have illustrated this in two particular cases, one in which no new node needs to be created and another where the creation of a node is necessary (when the column whose height is increased by the move is the only one with that height after the move). These pictures also illustrate the process of initializing the rows above the one where the move is performed: this leads to the successor of the current plane partition.

Note that this presentation is not exhaustive of all possible cases, since, for instance, some nodes (at most two, those corresponding to the two columns whose height is modified by the move) may have to be deleted from the list.

Lastly, step 4, is performed in constant time. In fact, after the rightmost move at j (a pop has been executed on the stack St[i] to obtain the link to the node associated with j) the remaining entries in St[i] are still valid, with the top indicating a move in position $k \leq j - 1$ of the current partition. By considering the profile of the grand ancestor (see Lemma 3), it is immediate to see that at most three new moves need to be pushed onto the stack: one possibly associated with a position h, with $k < h \leq i$, and two possibly associated with the suffix starting at $i + 1$.

Now, let us consider the procedure GenerateHProj(n, s, m) which is used to generate all plane partitions with a given horizontal projection in constant amortized time, as stated in the following:

Lemma 6. GenerateHProj*(n, s, m) runs in time $O(\sharp PP_s(n))$.*

Proof. The block of instructions from line 1 to line 14 is executed in time $O(l(s))$. Thus, we need only determine the cost of GenerateRow(l, s). This call consists of the (implicit) visit of the generation tree associated with s, where each node corresponds to one recursive call to GenerateRow. Since each element of $\mathrm{PP}_s(n)$ corresponds exactly to one element in a leaf (and vice versa), in order to prove that GenerateHProj(n, s, m) runs in time $O(\sharp \mathrm{PP}_s(n))$ it is sufficient to prove that:

1. each sequence in a node is CAT generated;
2. the leaves contain a number of elements which is larger than the number of elements in internal nodes.

Thus, statement (1) follows from Lemma 5, while Corollary 2 implies (2). □

As a consequence, one has that the set of all plane partitions of n can be generated in constant amortized time.

Theorem 2. *Procedure* GeneratePP *is CAT.*

Proof. Let $C(n)$ and $D(n, s)$ be the running times of GeneratePP(n) (see Algorithm 2) and GenerateHProj(n, s, m), respectively. First, by Lemma 5, the generation of $\mathrm{LP}(n) = \mathrm{LP}(n|0^{[n]})$ by iterating $\sharp\mathrm{LP}(n) - 1$ times the instruction $s := \mathrm{Next}(s, 0^{[n]})$ (line 5 of Algorithm 2) requires time $O(\sharp\mathrm{LP}(n))$. Therefore, one has $C(n) = O(\sharp\mathrm{LP}(n)) + \sum_{s\in\mathrm{LP}(n)} D(n, s)$. Lastly, since $\sharp\mathrm{PP}(n) = \sum_{s\in\mathrm{LP}(n)} \sharp\mathrm{PP}_s(n)$, by Lemma 6 we obtain

$$C(n) = O(\sharp\mathrm{LP}(n)) + \sum_{s\in\mathrm{LP}(n)} O(\sharp\mathrm{PP}_s(n)) = O(\sharp\mathrm{LP}(n)) + O(\sharp\mathrm{PP}(n)) = O(\sharp\mathrm{PP}(n)). \quad \square$$

5. Conclusions

We have shown that the problem of the exhaustive generation of all plane partitions of an integer n can be solved by a CAT algorithm which works (in some sense) by operating a reduction from two-dimensional objects (plane partitions) to one-dimensional ones (covering partitions). In particular, we point out that the method of generating covering partitions by extending the approach in [5] seems to have interesting applications in contexts where the objects to be generated can be expressed in terms of a finite number of linear partitions together with a finite number of binary relations on them. For instance, a CAT algorithm for generating parallelogram polyominoes of size n is shown in [7], where a slightly different binary relation between linear partitions leads to a representation of a parallelogram polyomino by means of two suitable partitions.

Acknowledgments

The authors wish to thank the anonymous referees for their accurate suggestions.

References

[1] C.F. Hindenburg, Infinitinomii dignitatum Exponentis Indeterminati, Göttingen (1779) 73–91.
[2] Donald E. Knuth, Art of Computer Programming, Volume 4A, The: Combinatorial Algorithms, Part 1, Addison-Wesley, 2011.
[3] A. Zoghbi, I. Stojmenović, Fast algorithms for generating integer partitions, Internat. J. Computer Math. 70 (1998) 319–332.
[4] F. Ruskey, Combinatorial Generation, draft, University of Victoria, 2003.
[5] P. Massazza, R. Radicioni, A CAT algorithm for the exhautive generation of ice piles, RAIRO Informatique théorique 44 (2010) 525–543.
[6] E. Duchi, R. Mantaci, H.D. Phan, D. Rossin, Bidimensional sand pile and ice pile models, PU.M.A. 17 (1–2) (2007) 71–96.
[7] R. Mantaci, P. Massazza, From linear partitions to parallelogram polyominoes, in: Proc. of DLT 2011, in: LNCS, vol. 6795, Springer, 2011, pp. 350–361.