

# Efficient Generation of Rectangulations via Permutation Languages

Arturo Merino ✉   
TU Berlin, Germany

Torsten Mütze ✉   
University of Warwick, Coventry, United Kingdom  
Charles University, Prague, Czech Republic

---

## Abstract

A generic rectangulation is a partition of a rectangle into finitely many interior-disjoint rectangles, such that no four rectangles meet in a point. In this work we present a versatile algorithmic framework for exhaustively generating a large variety of different classes of generic rectangulations. Our algorithms work under very mild assumptions, and apply to a large number of rectangulation classes known from the literature, such as generic rectangulations, diagonal rectangulations, 1-sided/area-universal, block-aligned rectangulations, and their guillotine variants. They also apply to classes of rectangulations that are characterized by avoiding certain patterns, and in this work we initiate a systematic investigation of pattern avoidance in rectangulations. Our generation algorithms are efficient, in some cases even loopless or constant amortized time, i.e., each new rectangulation is generated in constant time in the worst case or on average, respectively. Moreover, the Gray codes we obtain are cyclic, and sometimes provably optimal, in the sense that they correspond to a Hamilton cycle on the skeleton of an underlying polytope. These results are obtained by encoding rectangulations as permutations, and by applying our recently developed permutation language framework.

**2012 ACM Subject Classification** Theory of computation → Design and analysis of algorithms; Mathematics of computing → Discrete mathematics

**Keywords and phrases** Exhaustive generation, Gray code, flip graph, polytope, generic rectangulation, diagonal rectangulation, cartogram, floorplan, permutation pattern

**Digital Object Identifier** 10.4230/LIPIcs.SoCG.2021.54

**Related Version** *Full Version:* [arXiv:2103.09333](https://arxiv.org/abs/2103.09333) [23]

**Funding** This work was supported by German Science Foundation grant 413902284.

*Arturo Merino:* Supported by ANID Becas Chile 2019-72200522.

*Torsten Mütze:* Supported by Czech Science Foundation grant GA 19-08554S.

## 1 Introduction

Partitioning a geometric shape into smaller shapes is a fundamental theme in discrete and combinatorial geometry. In this paper we consider *rectangulations*, i.e., partitions of a rectangle into finitely many interior-disjoint rectangles. Such partitions have an abundance of practical applications, which motivates their combinatorial and algorithmic study. For example, rectangulations are an appealing way to represent geographic information as a cartogram. This is a map where each country is represented as a rectangle, the adjacencies between rectangles correspond to those between countries, and the areas of the rectangles are determined by some geographic variable, such as population size [34]. If the rectangulation is *area-universal* [12], then such an adjacency-preserving cartogram can be drawn for any assignment of area values to the rectangles. Another important use of rectangulations is as floorplans in VLSI design and architectural design. These problems often involve additional

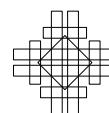


© Arturo Merino and Torsten Mütze;  
licensed under Creative Commons License CC-BY 4.0  
37th International Symposium on Computational Geometry (SoCG 2021).

Editors: Kevin Buchin and Éric Colin de Verdière; Article No. 54; pp. 54:1–54:18

Leibniz International Proceedings in Informatics

LIPICs Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany



constraints on top of adjacency, such as extra space for wires [27] or proportion limits for the rooms [24]. An important notion in this context are *slicing* floorplans [27], i.e., floorplans that can be subdivided into rectangles by a sequence of vertical or horizontal guillotine cuts.

Rectangulations have rich combinatorial properties, and a task that has received a lot of attention is counting, i.e., determining the number of rectangulations of a particular type with  $n$  rectangles, either exactly as a function of  $n$  [37] or asymptotically as  $n$  grows [32]. This led to several beautiful bijections of rectangulations with pattern-avoiding permutations [1, 4, 29] or with twin binary trees [37]. The focus of this paper is on another fundamental algorithmic task, which is more fine-grained than counting, namely exhaustive generation, meaning that every rectangulation from a given class must be produced exactly once. While such generation algorithms are known for many other discrete objects such as permutations, combinations, subsets, trees etc. and covered in standard textbooks such as Knuth's [20], much less is known about the generation of geometric objects such as rectangulations.


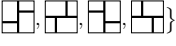

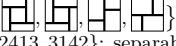

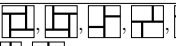
The ultimate goal for a generation algorithm is to produce each new object in time  $\mathcal{O}(1)$ , which requires that consecutively generated objects differ only by a "small local change". Such a minimum change listing of combinatorial objects is often called a *Gray code* [31]. If the time bound  $\mathcal{O}(1)$  for producing the next object holds in every step, then the algorithm is called *loopless* [11], and if it holds on average it is called *constant amortized time (CAT)* [30]. The Gray code problem entails the definition of a *flip graph*, which has as nodes all the combinatorial objects to be generated, and an edge between any two objects that differ in the specified small way. Clearly, computing a Gray code ordering of the objects is equivalent to traversing a Hamilton path or cycle in the corresponding flip graph. It turns out that some interesting flip graphs arising from rectangulations can be equipped with a natural lattice structure [21, 22], analogous to the Tamari lattice on triangulations, and realized as polytopes in high-dimensional space [28], analogous to the associahedron. This ties in the Gray code problem with deep methods and results from lattice and polytope theory.

## 1.1 Our results

The main contribution of this paper is a versatile algorithmic framework for generating a large variety of different classes of generic rectangulations, i.e., rectangulations with the property that no four rectangles meet in a point. In particular, we obtain efficient generation algorithms for several interesting classes known from the literature, in some cases loopless or CAT algorithms; see Table 1. The initialization time and memory requirement for all these algorithms is linear in the number of rectangles. The classes of rectangulations shown in the table arise from generic rectangulations by imposing structural constraints, such as the guillotine property or forbidden configurations, or by equivalence relations, and they will be defined in Section 2.2. We implemented the algorithms generating the classes of rectangulations from the table in C++, and we made the code available for download and experimentation on the Combinatorial Object Server [10].

The classes of rectangulations that our algorithms can generate are not limited to the examples shown in Table 1, but can be described by the following *closure property*; see Figure 1. Given an infinite class of rectangulations  $\mathcal{C}$ , we require that if a rectangulation  $R$  is contained in  $\mathcal{C}$ , then the rectangulation obtained from  $R$  by deleting the bottom-right rectangle is also in  $\mathcal{C}$ , and the two rectangulations obtained from  $R$  by inserting a new rectangle at the bottom or right, respectively, are also in  $\mathcal{C}$ . If  $\mathcal{C}$  satisfies this property, then our algorithms allow generating the set  $\mathcal{C}_n \subseteq \mathcal{C}$  of all rectangulations from  $\mathcal{C}$  with exactly  $n$  rectangles, for every  $n \geq 1$ , by so-called *jumps*, a minimum change operation that generalizes simple flips, T-flips, and wall slides studied in [8, 29]. Moreover, if the class  $\mathcal{C}$  is symmetric,

■ **Table 1** Classes of rectangulations that can be generated by our algorithms. The second column gives a description of the class in terms of forbidden rectangulation patterns (n/a means not applicable), and one or more bijectively equivalent classes of pattern-avoiding permutations. Underlined and overlined permutation patterns are so-called vincular and barred patterns; see [15] and the papers referenced in the table. The last column specifies the obtained runtime bound for generating each rectangulation, where  $n$  is the number of rectangles. These are all worst case bounds that apply in every step (in particular, LL=loopless), with the exception of the  $\mathcal{O}(1)$  bound for generic rectangulations, which holds on average (CAT=constant amortized time). For more extensive counting results on pattern-avoiding rectangulations, see [23].

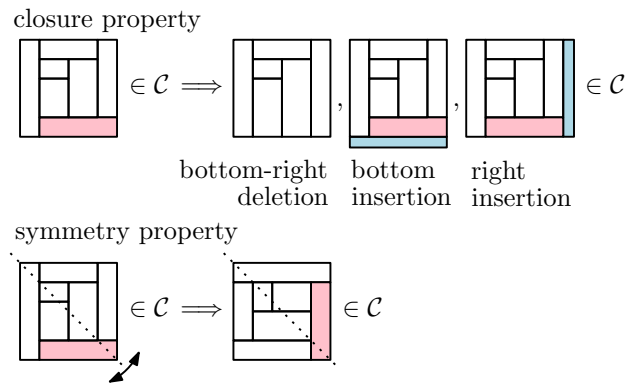
Class	Forbidden patterns	Counts/OEIS [26]	Refs.	Runtime	
generic	$\emptyset$ {35124, 35142, 24513, 42513}: 2-clumped permutations	1, 2, 6, 24, 116, 642, 3938, 26194, ...	[22, 29]	$\mathcal{O}(1)$ CAT	
diagonal =mosaic floorpl. /R-equivalence	 {2413, 3142}: Baxter {2413, 3412}: twisted Baxter {2143, 3142}	1, 2, 6, 22, 92, 422, 2074, 10754, ... A001181 (Baxter numbers)	[1, 8] [21, 37]	$\mathcal{O}(1)$ LL	
1-sided =area-universal		1, 2, 6, 20, 72, 274, 1088, 4470, ...	[12]	$\mathcal{O}(n)$	
block-aligned /S-equivalence	n/a {2143, 3412}	1, 1, 2, 6, 22, 88, 374, 1668, 7744, ... A214358	[4]	$\mathcal{O}(1)$ LL	
guilloché	generic		1, 2, 6, 24, 114, 606, 3494, 21434, ...	$\mathcal{O}(n)$	
	diagonal =slicing fl.pl. /R-equiv.	 {2413, 3142}: separable	1, 2, 6, 22, 90, 394, 1806, 8558, ... A006318 (Schröder numbers)	[1, 37] [4, 5]	
	1-sided	 {2413, 3142, 21354, 45312}	1, 2, 6, 20, 70, 254, 948, 3618, ... A078482	[5]	$\mathcal{O}(n)$
		 {2413, 3142, 2143, 3412}	1, 2, 6, 20, 68, 232, 792, 2704, ... A006012	[5]	$\mathcal{O}(n^2)$
	block-aligned /S-equiv.	n/a {2413, 3142, 2143, 3412}	1, 1, 2, 6, 20, 70, 254, 948, 3618, ... A078482	[4]	$\mathcal{O}(n)$

i.e., if  $R$  is in  $\mathcal{C}$  then the rectangulation obtained from  $R$  by reflection at the diagonal from top-left to bottom-right is also in  $\mathcal{C}$ , then the jump Gray code for  $\mathcal{C}_n$  is cyclic, i.e., the last rectangulation differs from the first one only by a jump. In other words, we not only obtain a Hamilton path in the corresponding flip graph, but a Hamilton cycle. In fact, all the classes of rectangulations listed in Table 1 satisfy the aforementioned closure and symmetry properties, so in all those cases we obtain cyclic jump Gray codes.

Generic rectangulations and diagonal rectangulations, shown in the first two rows of Table 1, have an underlying lattice and polytope structure [21, 22, 28], and in those two cases our Gray codes form a Hamilton cycle on the skeleton of this polytope, i.e., jumps are provably optimal minimum change operations.

It turns out that many interesting classes of rectangulations can be characterized by pattern avoidance; see the second column in Table 1. Under very mild conditions on the patterns, these classes satisfy the aforementioned closure property, and can hence be generated by our framework. In this work we initiate a systematic investigation of pattern avoidance in rectangulations, and we obtain the first counting results for many known and new classes; see the third column in Table 1 and the more extensive tables in [23].

Our generation framework for rectangulations consists of two main algorithms. The first is a simple greedy algorithm that generates a jump Gray code ordering for any set of rectangulations  $\mathcal{C}_n \subseteq \mathcal{C}$  for which  $\mathcal{C}$  satisfies the aforementioned closure property; see



■ **Figure 1** Closure property and symmetry property.

Algorithm  $J^\square$  and Theorem 5 in Section 3. The second is a memoryless version of the first algorithm, which computes the same ordering of rectangulations; see Algorithm  $M^\square$  and Theorem 8 in Section 5. This algorithm can be fine-tuned to derive efficient algorithms for several known rectangulation classes such as the ones listed in Table 1, by providing corresponding jump oracles for the class  $\mathcal{C}$ .

To prove Theorems 5 and 8, we encode rectangulations by permutations as described by Reading [29], and we then apply our framework for exhaustively generating permutation languages presented in [14, 15, 17]. The minimum change operations on permutations used in that framework translate to jumps on rectangulations. Generating different classes of rectangulations efficiently is thus another major new application of our permutation language framework, and in this paper we flesh out the details of this application.

## 1.2 Related work

There has been some prior work on generating a few special classes of rectangulations, all based on Avis and Fukuda’s reverse search method [6]. Specifically, Nakano [25] described a CAT generation algorithm for generic rectangulations, which does not produce a Gray code, however. This algorithm has been adapted by Takagi and Nakano [33] to generate generic rectangulations with bounds on the number of rectangles that do not touch the outer face. Yoshii, Chigira, Yamanaka and Nakano [38] gave a Gray code for diagonal rectangulations based on a generating tree that is different from ours, resulting in a loopless algorithm. Their Gray code changes at most 3 edges of the rectangulation in each step, whereas our algorithm changes only 1 edge in each step for diagonal rectangulations and generic rectangulations. Consequently, none of the listings produced by these earlier algorithms corresponds to a walk along the skeleton of the underlying polytope.

There has been a lot of work on combinatorial properties of rectangulations. Yao, Chen, Cheng and Graham [37] showed that diagonal rectangulations are counted by the Baxter numbers and that guillotine diagonal rectangulations are counted by the Schröder numbers, using a bijection between diagonal rectangulations and twin binary trees. Ackerman, Barequet and Pinter [1] presented another bijection between diagonal rectangulations and Baxter permutations, which also yields a bijection between guillotine diagonal rectangulations and separable permutations. Shen and Chu [32] provided asymptotic estimates for these two rectangulation classes. Moreover, He [16] presented an optimal encoding of diagonal rectangulations with  $n$  rectangles using only  $3n - 3$  bits, which is optimal.

The term “generic rectangulation” was coined by Reading [29], who established a bijection between generic rectangulations and 2-clumped permutations, proving that these permutations are representatives of equivalence classes of a lattice congruence of the weak order on the symmetric group. Earlier, generic rectangulations had been studied under the name “rectangular drawings” by Amano, Nakano and Yamanaka [3] and by Inoue, Takahashi and Fujimaki [13, 19], who established recursion formulas and asymptotic bounds for their number. More general classes of rectangular partitions were analyzed by Conant and Michaels [9].

Ackerman, Barequet and Pinter [2] considered the setting where we are given a set of  $n$  points in general position in a rectangle, and the goal is to partition the rectangle into smaller rectangles by  $n$  walls, such that each point from the set lies on a distinct wall. They showed that for every set of points that forms a separable permutation in the plane, the number of possible rectangulations is the  $(n + 1)$ st Baxter number, and for every point set the number of possible guillotine rectangulations is the  $n$ th Schröder number. They also presented a counting and generation procedure based on simple flips and T-flips using reverse search, which was later improved by Yamanaka, Rahman and Nakano [36].

### 1.3 Outline of this paper

In Section 2 we provide basic definitions and concepts that will be used throughout the paper. In Section 3 we present a greedy algorithm for generating a set of rectangulations by jumps, and we provide a sufficient condition for the algorithm to succeed. In Section 4 we show that the algorithm applies to a large number of rectangulation classes that are characterized by pattern avoidance. In Section 5 we demonstrate how to make our generation algorithm memoryless and efficient. The implementation details of these algorithms and the proofs of Theorems 5 and 8 are omitted due to space constraints; they can be found in [23].

## 2 Preliminaries

### 2.1 Generic rectangulations

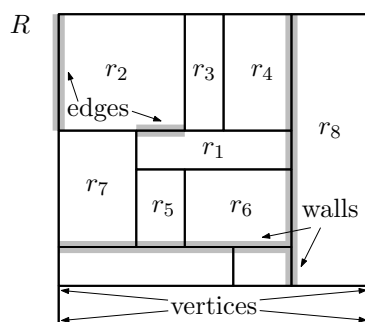
A *generic rectangulation*, or rectangulation for short, is a partition of a rectangle into finitely many interior-disjoint axis-aligned rectangles, such that no four rectangles of the partition have a point in common; see Figure 2. Given rectangles  $r$  and  $s$ , we say that  $r$  is *left* of  $s$ , and  $s$  is *right* of  $r$ , if the right side of  $r$  intersects the left side of  $s$  (necessarily in a line segment, rather than a single point). Similarly, we say that  $r$  is *below*  $s$ , and  $s$  is *above*  $r$ , if the top side of  $r$  intersects the bottom side of  $s$ . We consider generic rectangulations up to equivalence that preserves the left/right and below/above relations between rectangles, and we write  $\mathcal{R}_n$  for the set of all rectangulations with  $n$  rectangles.

We refer to every rectangle corner in a rectangulation as a *vertex*, to every minimal line segment between two vertices as an *edge*, and to every maximal line segment between two vertices that are not corners of the rectangulation as a *wall*. The *type* of a vertex that is not a corner describes the shape of the T-joint at this vertex, and it is one of  $\top$ ,  $\vdash$ ,  $\perp$ , or  $\dashv$ .

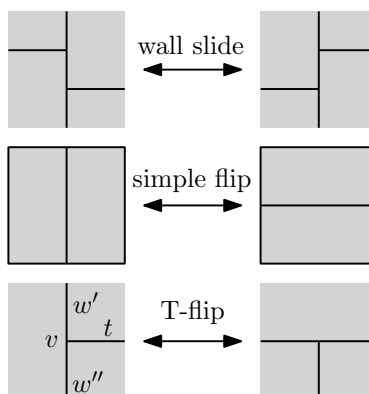
### 2.2 Flip operations and classes of rectangulations

Our Gray codes use three types of local change operations on rectangulations; see Figure 3.

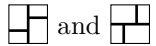
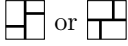
A *wall slide* swaps the order of two neighboring vertices of types  $\vdash$  and  $\dashv$  along a vertical wall, or of types  $\top$  and  $\perp$  along a horizontal wall. A *simple flip* swaps the orientation of a wall that separates two rectangles. For a vertex  $v$  that belongs to three rectangles, consider the wall  $w$  that goes through  $v$  and the wall  $t$  that ends at  $v$ , and let  $w'$  and  $w''$  be the two halves of  $w$  meeting in  $v$ . A *T-flip* swaps the orientation of  $w'$  or  $w''$  so that it merges with  $t$ .



■ **Figure 2** Generic rectangulation  $R$  with 11 rectangles. The rectangle  $r_1$  is below  $r_2$ ,  $r_3$  and  $r_4$ , above  $r_5$  and  $r_6$ , right of  $r_7$  and left of  $r_8$ .





■ **Figure 3** Local change operations on rectangulations.

We now define various interesting subclasses of generic rectangulations that have been studied in the literature and that appear in Table 1. A *diagonal* rectangulation is one in which every rectangle intersects the *main diagonal* that goes from the top-left to the bottom-right corner of the rectangulation. We write  $\mathcal{D}_n \subseteq \mathcal{R}_n$  for the set of all diagonal rectangulations with  $n$  rectangles. Diagonal rectangulations are characterized by avoiding the wall patterns  [8]. Consider the equivalence relation on  $\mathcal{R}_n$  obtained from wall slides, sometimes referred to as *R-equivalence* [4]. The equivalence classes are referred to as *mosaic floorplans*, and every equivalence class contains exactly one diagonal rectangulation, obtained by repeatedly destroying occurrences of  by wall slides [8]. Consequently, in a diagonal rectangulation, along every vertical wall, all  $\vdash$ -vertices are below all  $\dashv$ -vertices, and along every horizontal wall, all  $\perp$ -vertices are to the left of all  $\top$ -vertices.

In a *1-sided* rectangulation, every wall is the side of at least one rectangle. This notion was introduced by Eppstein, Mumford, Speckmann, and Verbeek [12], who used it to characterize *area-universal* rectangulations, i.e., for any assignment of areas to the rectangles, the rectangulation can be drawn so that each rectangle has the prescribed area.

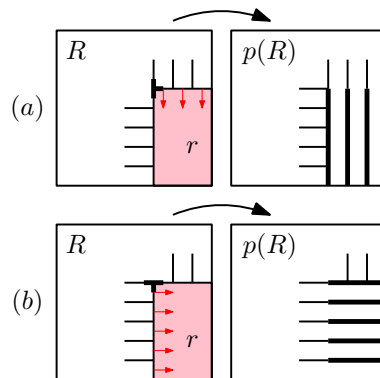
Asinowski et al. [4] also considered the equivalence relation on  $\mathcal{R}_n$  obtained from wall slides and simple flips, and they called it *S-equivalence*. By definition, S-equivalence is a coarser relation than R-equivalence, i.e., the equivalence classes are obtained by identifying mosaic floorplans that differ in simple flips. In [23] we introduce *block-aligned* rectangulations, which are a subset of diagonal rectangulations with the property that every equivalence class of S-equivalence contains exactly one block-aligned rectangulation.

A rectangulation is *guillotine*, if each of its rectangles can be cut out from the entire rectangulation by a sequence of straight vertical or horizontal cuts. Guillotine rectangulations are characterized by avoiding the windmill patterns  and . Various special classes of guillotine diagonal rectangulations, characterized by the avoidance of certain wall configurations, were introduced by Asinowski and Mansour [5] (see Section 4 for precise definitions). Mosaic floorplans that are guillotine are also known as *slicing* floorplans.

### 2.3 Deletion of rectangles

We now describe two operations on a generic rectangulation  $R$  with  $n$  rectangles, namely deleting a rectangle and inserting a rectangle. The resulting rectangulations have  $n - 1$  or  $n + 1$  rectangles, respectively, and they will be denoted by  $p(R)$  and  $c_i(R)$ , notations that refer to the parent and children of  $R$ , in a tree structure that will be discussed shortly. The deletion and insertion operations were introduced in [18] and heavily used e.g. in [1] and [25].

The idea of deletion is to contract the rectangle in the bottom-right corner of the rectangulation. Formally, given a rectangulation  $R \in \mathcal{R}_n$ ,  $n \geq 2$ , we consider the rectangle  $r$  in the bottom-right corner, and we consider the top-left vertex of  $r$ . If this vertex has type  $\vdash$ , then we collapse  $r$  by sliding its top side, which forms a wall, downwards until it merges with the bottom side of  $r$ ; see Figure 4 (a). Similarly, if this vertex has type  $\top$ , then we collapse  $r$  by sliding its left side, which forms a wall, to the right until it merges with the right side of  $r$ ; see Figure 4 (b). We denote the resulting rectangulation with  $n - 1$  rectangles by  $p(R) \in \mathcal{R}_{n-1}$ , and we say that  $p(R)$  is obtained from  $R$  by *deletion*.



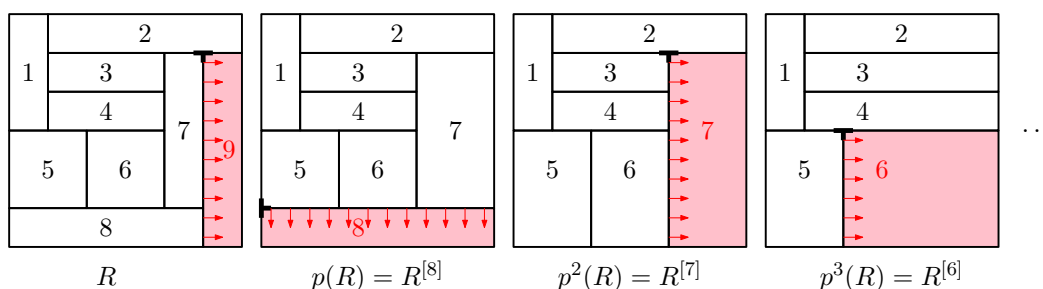
■ **Figure 4** Deletion operation.

Moreover, we denote the  $n$  rectangles of  $R$  by  $r_n, r_{n-1}, \dots, r_1$  in the order in which they are deleted when applying the deletion operation exhaustively; see Figure 5. Clearly, if  $r_i$  is deleted and its top-left vertex has type  $\vdash$ , then the rightmost rectangle above  $r_i$  is  $r_{i-1}$ . Similarly, if the top-left vertex has type  $\top$ , then the lowest rectangle to the left of  $r_i$  is  $r_{i-1}$ .

For any  $R \in \mathcal{R}_n$  and  $i = 1, \dots, n$  we define  $R^{[i]} := p^{n-i}(R)$ , i.e., this is the sub-rectangulation of  $R$  formed by the first  $i$  rectangles; see Figure 5.

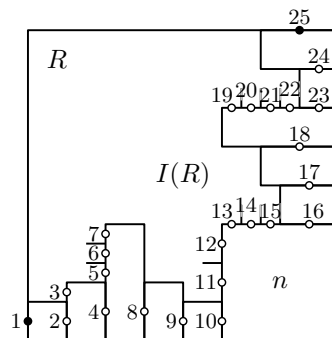
### 2.4 Insertion of rectangles

The idea of insertion is to add a new rectangle into the bottom-right corner of the rectangulation. Given a rectangulation  $R \in \mathcal{R}_n$ , we first define a set of points in  $R$  that can become the top-left corner of the newly added rectangle; see Figure 6.



■ **Figure 5** A rectangulation and the indexing of its rectangles given by repeated deletion.

For any rectangle  $r$  in  $R \in \mathcal{R}_n$ ,  $n \geq 1$ , that touches the bottom boundary of  $R$ , we consider all edges forming the left side of  $r$ , and from every such edge we select one interior point, and we refer to it as a *vertical insertion point*. Similarly, for any rectangle  $r$  in  $R$  that touches the right boundary of  $R$ , we consider the set of all edges forming the top side of  $r$ , and from every such edge we select one interior point, and we refer to it as a *horizontal insertion point*. Combinatorially it does not make a difference which interior point is selected.



■ **Figure 6** Linear ordering of insertion points. First and last insertion point are filled.

We order the insertion points linearly, by sorting all vertical insertion points lexicographically by their  $(x, y)$ -coordinates, followed by all horizontal insertion points sorted lexicographically by their  $(y, x)$ -coordinates; see Figure 6. We write  $I(R) = (q_1, q_2, \dots, q_\nu)$  for the sequence of all insertion points ordered in this linear order. In particular,  $\nu = \nu(R)$  denotes the number of insertion points.

► **Lemma 1.** *For any rectangulation  $R \in \mathcal{R}_n$  we have  $\nu(R) \leq n + 1$ .*

The proof of Lemma 1 is straightforward; see [23]. The upper bound in Lemma 1 is attained if every rectangle touches the bottom or right boundary of  $R$ .

Given  $R \in \mathcal{R}_n$  and the sequence of insertion points  $I(R) = (q_1, \dots, q_\nu)$ , for each  $i = 1, \dots, \nu$  we define a rectangulation  $c_i(R) \in \mathcal{R}_{n+1}$  as follows: If  $q_i$  is a vertical insertion point, then  $c_i(R)$  is obtained from  $R$  by inserting a new rectangle  $r_{n+1}$  in the bottom-right corner such that  $r_{n+1}$  has above it exactly all rectangles which in  $R$  lie to the right of  $q_i$  and touch the bottom boundary of  $R$ , and such that  $r_{n+1}$  has to its left exactly all rectangles which in  $R$  touch the vertical wall through  $q_i$  below  $q_i$ ; see Figure 7 (a). Similarly, if  $q_i$  is a horizontal insertion point, then  $c_i(R)$  is obtained from  $R$  by inserting a new rectangle  $r_{n+1}$  in the bottom-right corner such that  $r_{n+1}$  has to its left exactly all rectangles which in  $R$



lie below  $q_i$  and touch the right boundary of  $R$ , and such that  $r_{n+1}$  has above it exactly all rectangles which in  $R$  touch the horizontal wall through  $q_i$  to the right of  $q_i$ ; see Figure 7 (b). We say that  $c_i(R)$  is obtained from  $R$  by *insertion*.

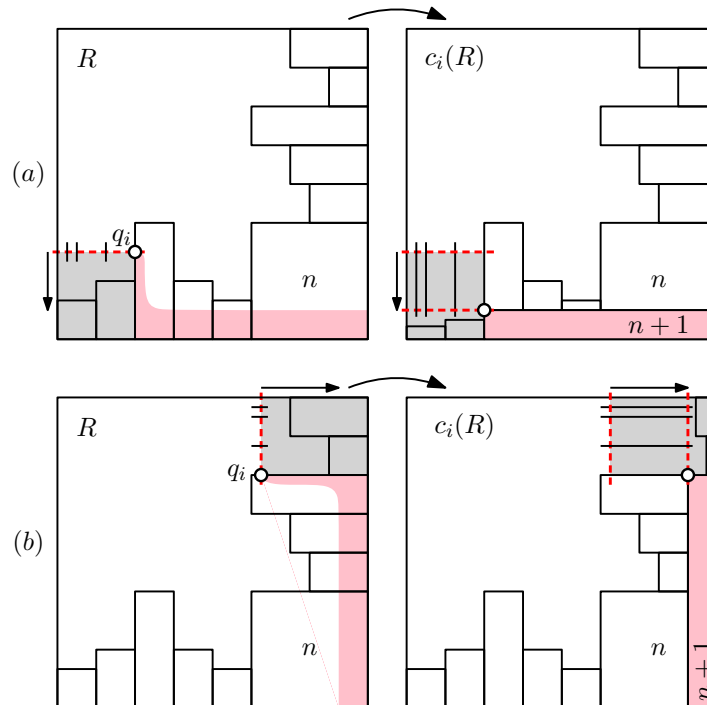


Figure 7 Insertion operation.

By these definitions, the operations of deletion and insertion are inverse to each other, which we record in the following lemma.

► **Lemma 2.** For any rectangulation  $R \in \mathcal{R}_n$  and any two distinct insertion points  $q_i$  and  $q_j$  from  $I(R)$ , the rectangulations  $c_i(R) \in \mathcal{R}_{n+1}$  and  $c_j(R) \in \mathcal{R}_{n+1}$  are distinct, and we have  $R = p(c_i(R)) = p(c_j(R))$ .

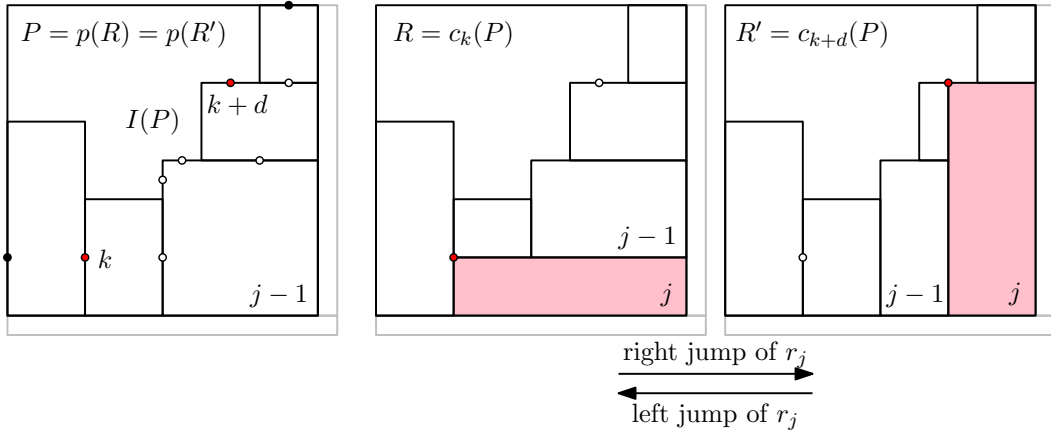
The first and last insertion point, highlighted in Figure 6, play a special role. We say that  $R$  is *bottom-based* if  $R$  has a rectangle whose bottom side is the entire bottom boundary of  $R$ , and  $R$  is *right-based* if  $R$  has a rectangle whose right side is the entire right boundary of  $R$ . Note that  $\square$  is both bottom-based and right-based, and if  $n \geq 2$ , then  $R \in \mathcal{R}_n$  is bottom-based if and only if  $R = c_1(p(R))$  and right-based if and only if  $R = c_{\nu(p(R))}(p(R))$ .

### 3 The basic algorithm

We now present the basic algorithm that we use to generate a set of rectangulations  $\mathcal{C}_n \subseteq \mathcal{R}_n$ .

#### 3.1 Jumps in rectangulations

We first introduce a local change operation that generalizes the three kinds of flips introduced in Section 2.2 (recall Figure 3) and that will be applied when moving from one rectangulation in  $\mathcal{C}_n$  to the next in the algorithm. A *jump* changes the insertion point for exactly one rectangle of the rectangulation. Formally, for a rectangulation  $R \in \mathcal{R}_n$ , we say that  $R' \in \mathcal{R}_n$  differs from  $R$  by a *right jump of rectangle  $r_j$  by  $d$  steps*, denoted  $R' = \vec{J}(R, j, d)$ , where  $2 \leq j \leq n$  and  $d > 0$ , if one of the following conditions holds; see Figure 8:



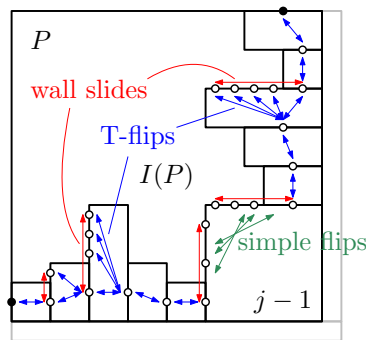
■ **Figure 8** Definition of jumps.

- $j = n$ , and  $p(R) = p(R') =: P \in \mathcal{R}_{n-1}$ ,  $R = c_k(P)$  and  $R' = c_{k+d}(P)$  for some  $k > 0$ ;
- $j < n$ , and  $R$  and  $R'$  are either both bottom-based or both right-based, and  $p(R')$  differs from  $p(R)$  in a right jump of rectangle  $r_j$  by  $d$  steps.

In words, the first condition asserts that the first  $n - 1$  rectangles in  $R$  and  $R'$  form the same rectangulation  $P \in \mathcal{R}_{n-1}$ , and  $R$  and  $R'$  are obtained by insertion from  $P$  using the  $k$ th and  $(k + d)$ th insertion point, respectively. The second condition asserts that  $R$  and  $R'$  agree in the rectangle  $r_n$ , which either forms the bottom boundary or the right boundary of those rectangulations, and  $p(R')$  differs from  $p(R)$  in a right jump with the same parameters.

A right jump as before is called *minimal* w.r.t. to a set of rectangulations  $\mathcal{C}_n \subseteq \mathcal{R}_n$ , if in the first condition above there is no index  $\ell$  with  $k < \ell < k + d$  such that  $c_\ell(P) \in \mathcal{C}_n$ .

A (*minimal*) *left jump*, denoted  $R' = \overleftarrow{J}(R, j, d)$ , is defined analogously by replacing  $c_{k+d}$  by  $c_{k-d}$  and  $k < \ell < k + d$  by  $k > \ell > k - d$  in the definitions above. Clearly, if  $R'$  differs from  $R$  by a right jump of rectangle  $r_j$  by  $d$  steps, then  $R$  differs from  $R'$  by a left jump of rectangle  $r_j$  by  $d$  steps, and vice versa, i.e., we have  $R' = \overleftarrow{J}(R, j, d)$  if and only if  $R = \overleftarrow{J}(R', j, d)$ . We sometimes simply say that  $R$  and  $R'$  differ in a jump, without specifying the direction. We state the following simple observations for further reference; see Figure 9.



■ **Figure 9** Jumps generalize wall slides, simple flips and T-flips.

► **Lemma 3.** Consider two rectangulations  $R, R' \in \mathcal{R}_n$  that differ in a jump of rectangle  $r_j$ , define  $P := R^{[j-1]} = R'^{[j-1]} \in \mathcal{R}_{j-1}$ , and let  $q_k$  and  $q_\ell$  be the insertion points in  $I(P)$  such that  $R^{[j]} = c_k(P)$  and  $R'^{[j]} = c_\ell(P)$ .

- (a) If  $q_k$  and  $q_\ell$  are consecutive (w.r.t.  $I(P)$ ) on a common wall of  $P$ , then  $R$  and  $R'$  differ in a wall slide.
- (b) If  $q_k$  lies on the last vertical wall and  $q_\ell$  on the first horizontal wall of  $P$  (w.r.t.  $I(P)$ ), then  $R$  and  $R'$  differ in a simple flip.
- (c) If  $q_k$  lies on a vertical wall and  $q_\ell$  is the first insertion point on the next vertical wall of  $P$  (w.r.t.  $I(P)$ ), or if  $q_k$  lies on a horizontal wall and  $q_\ell$  is the last insertion point on the previous horizontal wall, then  $R$  and  $R'$  differ in a  $T$ -flip.

### 3.2 Generating rectangulations by minimal jumps

Consider the following algorithm that attempts to greedily generate a set of rectangulations  $\mathcal{C}_n \subseteq \mathcal{R}_n$  using minimal jumps.

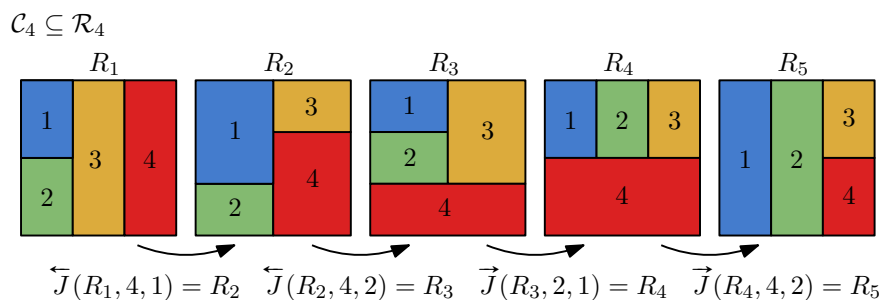
■ **Algorithm  $J^\square$**  (Greedy minimal jumps).

This algorithm attempts to greedily generate a set of rectangulations  $\mathcal{C}_n \subseteq \mathcal{R}_n$  using minimal jumps starting from an initial rectangulation  $R_0 \in \mathcal{C}_n$ .

- J1. [Initialize] Visit the initial rectangulation  $R_0$ .
- J2. [Jump] Generate an unvisited rectangulation from  $\mathcal{C}_n$  by performing a minimal jump of the rectangle with maximum index in the most recently visited rectangulation. If no such jump exists, or the jump direction is ambiguous, then terminate. Otherwise visit this rectangulation and repeat J2.

To illustrate how Algorithm  $J^\square$  works, consider the set of five rectangulations  $\mathcal{C}_4 = \{R_1, \dots, R_5\} \subseteq \mathcal{R}_4$  shown in Figure 10. If initialized with  $R_0 := R_1$ , then the algorithm performs a left jump of rectangle 4 by one step (a right jump of rectangle 4 is impossible) to reach  $R_2$ , i.e., we have  $R_2 = \overleftarrow{J}(R_1, 4, 1)$ . In  $R_2$ , there are two options, either a right jump of rectangle 4 by one step, leading back to  $R_1$ , which has been visited before, or a left jump of rectangle 4 by two steps, leading to  $R_3$ , so we visit  $R_3 = \overleftarrow{J}(R_2, 4, 2)$ . In  $R_3$ , the jumps involving rectangle 4 lead to rectangulations that were visited before ( $R_1$  and  $R_2$ ). Moreover, a jump of rectangle 3 does not lead to a rectangulation in  $\mathcal{C}_4$ . However, a right jump of rectangle 2 by one step leads to  $R_4$  (a left jump of rectangle 2 is impossible), so we visit  $R_4 = \overrightarrow{J}(R_3, 2, 1)$ . Finally, in  $R_4$  a right jump of rectangle 4 by two steps leads to  $R_5 = \overrightarrow{J}(R_4, 4, 2)$  (a left jump of rectangle 4 is impossible). In this example, Algorithm  $J^\square$  successfully visits every rectangulation from  $\mathcal{C}_4$  exactly once.

On the other hand, suppose we instead initialize the algorithm with  $R_0 := R_3$ . The algorithm will then visit  $R_2 := \overleftarrow{J}(R_3, 4, 2)$  followed by  $R_1 := \overleftarrow{J}(R_2, 4, 1)$ , and then terminates without success, as from  $R_1$  no jump leads to an unvisited rectangulation from  $\mathcal{C}_4$ . Lastly,



■ **Figure 10** Example execution of Algorithm  $J^\square$ .

suppose we initialize Algorithm  $J^\square$  with  $R_0 := R_2$ . As before, in  $R_2$ , there are two possibilities, either a right jump or a left jump of rectangle 4, both leading to an unvisited rectangulation from  $\mathcal{C}_4$ . Both are minimal jumps in opposite directions, and as the jump direction is ambiguous, the algorithm terminates immediately without success.

► **Remark 4.** *We do not recommend using Algorithm  $J^\square$  in the stated form to generate a set of rectangulations efficiently!* This is because the algorithm requires to maintain the list of all previously visited rectangulations (possibly exponentially many), and to look up this list in each step to check whether a rectangulation obtained by a jump from the current one has been visited before. For us, Algorithm  $J^\square$  is merely a tool to define a Gray code ordering of the rectangulations in the given set  $\mathcal{C}_n$  in way that is easy to remember (cf. [35]). In fact, in Section 5 we will present a modified algorithm that dispenses with the costly lookup operations, and that computes the very same sequence of rectangulations.

### 3.3 A guarantee for success

By definition, Algorithm  $J^\square$  visits every rectangulation from a given set  $\mathcal{C}_n \subseteq \mathcal{R}_n$  at most once, but it may terminate before having visited all. We now provide a sufficient condition guaranteeing that Algorithm  $J^\square$  visits every rectangulation from  $\mathcal{C}_n$  exactly once.

A set of generic rectangulations  $\mathcal{C}_n \subseteq \mathcal{R}_n$  is called *zigzag*, if either  $n = 1$  and  $\mathcal{C}_1 = \{\square\}$ , or if  $n \geq 2$  and  $\mathcal{C}_{n-1} := \{p(R) \mid R \in \mathcal{C}_n\}$  is zigzag and for every  $R \in \mathcal{C}_{n-1}$  we have  $c_1(R) \in \mathcal{C}_n$  and  $c_{\nu(R)}(R) \in \mathcal{C}_n$ . In words, a zigzag set  $\mathcal{C}_n$  is closed under repeatedly deleting bottom-right rectangles and replacing them by rectangles inserted either below or to the right of the remaining ones; recall Figure 1. The name “zigzag” does not refer to the shape of a rectangulation and will become clear momentarily. We also say that  $\mathcal{C}_n$  is *symmetric*, if reflection at the main diagonal is an involution of  $\mathcal{C}_n$ , i.e., if  $R \in \mathcal{C}_n$ , then the rectangulation obtained from  $R$  by reflection at the main diagonal is also in  $\mathcal{C}_n$ . We write  $\begin{bmatrix} n \\ \dots \\ \dots \end{bmatrix}$  for the rectangulation that consists of  $n$  vertically stacked rectangles.

► **Theorem 5.** *Given any zigzag set of rectangulations  $\mathcal{C}_n$  and initial rectangulation  $R_0 = \begin{bmatrix} n \\ \dots \\ \dots \end{bmatrix}$ , Algorithm  $J^\square$  visits every rectangulation from  $\mathcal{C}_n$  exactly once. Moreover, if  $\mathcal{C}_n$  is symmetric, then the ordering of rectangulations generated by Algorithm  $J^\square$  is cyclic, i.e., the first and last rectangulation differ in a minimal jump.*

Note that the rectangulation  $R_0 = \begin{bmatrix} n \\ \dots \\ \dots \end{bmatrix}$  is contained in every zigzag set by definition, so this is a valid initialization for Algorithm  $J^\square$ . We write  $J^\square(\mathcal{C}_n)$  for the sequence of rectangulations generated by Algorithm  $J^\square$  for a zigzag set  $\mathcal{C}_n$  when initialized with  $R_0 = \begin{bmatrix} n \\ \dots \\ \dots \end{bmatrix}$ .

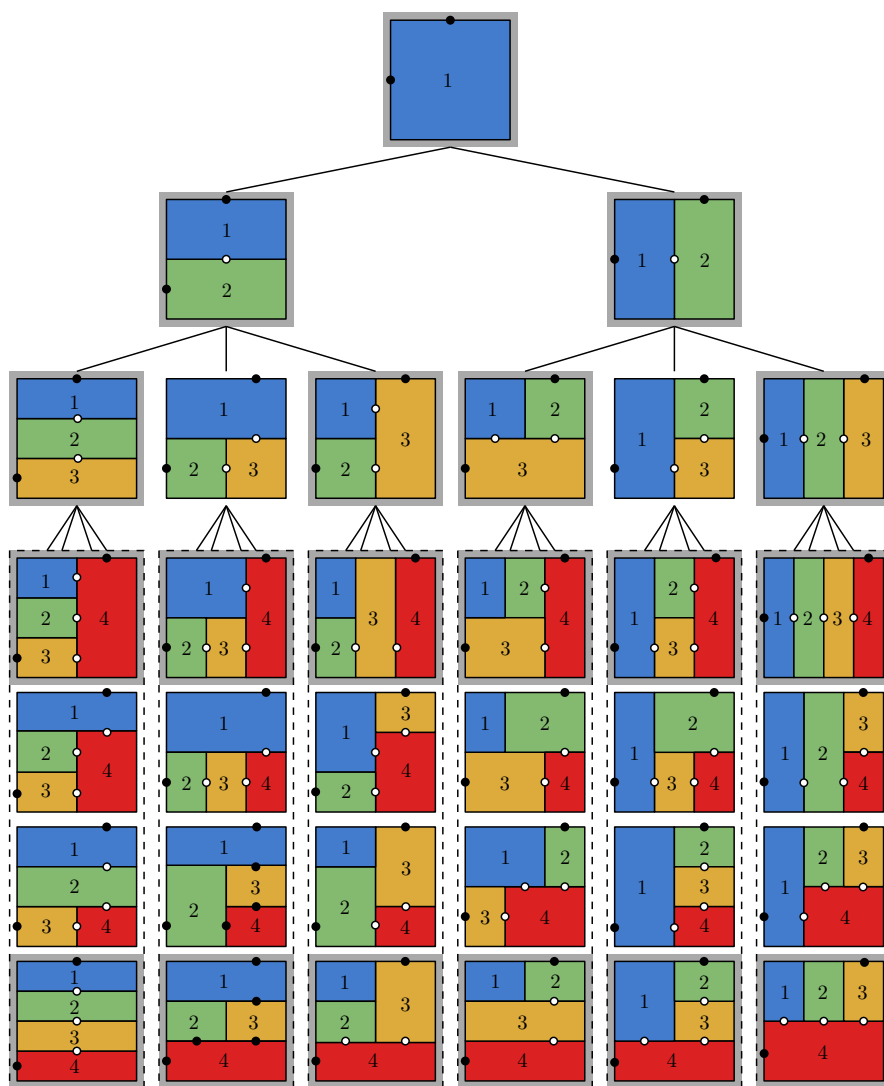
It is easy to see that the number of distinct zigzag sets of generic rectangulations is at least  $2^{|\mathcal{R}_n|(1-o(1))} \geq 2^{\Omega(11.56^n)}$  (the latter estimate uses the best known lower bound on  $|\mathcal{R}_n|$  from [3]), i.e., at least double-exponential in  $n$ . In other words, Algorithm  $J^\square$  exhaustively generates a given set of generic rectangulations in a vast number of cases. Moreover, many natural classes of rectangulations are in fact zigzag. In particular, *all* the classes introduced in Section 2.2 and shown in Table 1 satisfy the aforementioned closure property. Moreover, all of these classes are symmetric, so for each of them we obtain cyclic jump orderings.

### 3.4 Tree of rectangulations

The notion of zigzag sets and the operation of Algorithm  $J^\square$  can be interpreted combinatorially in the so-called *tree of rectangulations*, which is an infinite rooted tree, defined recursively as follows; see Figure 11: The root of the tree is a single rectangle  $\square \in \mathcal{R}_1$ . For any node

$R \in \mathcal{R}_n$ ,  $n \geq 1$ , of the tree we consider all insertion points of the rectangulation  $R$ , and the set of children of  $R$  in the tree is  $\{c_i(R) \in \mathcal{R}_{n+1} \mid i = 1, \dots, \nu(R)\}$ . Conversely, the parent of each  $R \in \mathcal{R}_n$ ,  $n \geq 2$ , is  $p(R) \in \mathcal{R}_{n-1}$ . In words, insertion leads to the children of a node, and deletion leads to the parent of a node. By Lemma 2, each generic rectangulation appears exactly once in the tree, and the set of nodes in distance  $n$  from the root of the tree is precisely the set  $\mathcal{R}_{n+1}$  of generic rectangulations with  $n + 1$  rectangles. We emphasize that this tree is *unordered*, i.e., there is no specified ordering among the children of a node.

By Lemma 1, a node  $R \in \mathcal{R}_n$  in the tree has at most  $n + 1$  children, i.e., we have  $|\mathcal{R}_n| \leq n!$ . As we see from Figure 11, this inequality is tight up to  $n = 4$ , but starting from  $n = 4$ , there are nodes  $R \in \mathcal{R}_n$  with strictly less than  $n + 1$  children, i.e., we have  $|\mathcal{R}_5| < 5!$ . In fact, it was shown in [3] that  $|\mathcal{R}_n| = \mathcal{O}(28.3^n)$ .



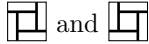
■ **Figure 11** Tree of generic rectangulations up to depth 3 with insertion points highlighted, where first and last insertion point are filled. The rectangulations in the dashed boxes at the bottom level  $\mathcal{R}_4$  are stacked on top of each other due to space constraints, but they are children of a common parent node. Bottom- or right-based rectangulations, corresponding to insertion at the first or last insertion point, are marked by gray boxes.

A subset  $\mathcal{C}_n \subseteq \mathcal{R}_n$  of nodes in depth  $n - 1$  of this tree is zigzag, if and only if it arises from the full tree of rectangulations by pruning some subtrees whose roots are neither bottom-based nor right-based rectangulations. In Figure 11, all bottom-based or right-based rectangulations are highlighted by gray boxes, and can therefore not be pruned, while all other nodes can possibly be pruned. If no nodes are pruned, then we have  $\mathcal{C}_n = \mathcal{R}_n$ , and if all possible nodes are pruned, then  $\mathcal{C}_n$  is the set  $\mathcal{B}_n$  of  $2^{n-1}$  rectangulations obtained by repeatedly stacking a new rectangle either below or to the right of the previous ones, i.e.,  $\mathcal{B}_n = \{c_1(R), c_{\nu(R)}(R) \mid R \in \mathcal{B}_{n-1}\}$  for  $n \geq 2$  and  $\mathcal{B}_1 = \{\square\}$ . Moreover, we have  $\mathcal{B}_n \subseteq \mathcal{C}_n \subseteq \mathcal{R}_n$  for any zigzag set  $\mathcal{C}_n$ .

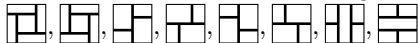
The operation of Algorithm  $J^\square$  for a zigzag set  $\mathcal{C}_n$  as input can be interpreted as follows: Given the pruned tree corresponding to  $\mathcal{C}_n$ , we consider the set of nodes on all previous levels of the tree, i.e., the sets  $\mathcal{C}_{i-1} := \{p(R) \mid R \in \mathcal{C}_i\}$  for  $i = n, n - 1, \dots, 2$ , which are all zigzag sets by definition. Moreover, we consider the orderings  $J^\square(\mathcal{C}_i)$ ,  $i = 1, \dots, n$ , defined by Algorithm  $J^\square$  for each of these sets. These sequences turn the unordered tree corresponding to  $\mathcal{C}_n$  into an ordered tree, where the children  $c_i(R)$  of each node  $R$  from left to right appear alternatingly in increasing order  $i = 1, \dots, \nu(R)$  or in decreasing order  $i = \nu(R), \nu(R) - 1, \dots, 1$ . Consequently, in the sequence  $J^\square(\mathcal{C}_i)$ ,  $i \geq 2$ , which forms the left-to-right sequence of all nodes in depth  $i - 1$  of this ordered tree, the rectangle  $r_i$  alternatingly jumps left and right between the first and last insertion point, which motivates the name “zigzag” set; see also the animations provided in [10].

#### 4 Pattern-avoiding rectangulations

We now show that Algorithm  $J^\square$  applies to a large number of rectangulation classes that are defined by pattern avoidance, under some very mild conditions; recall Table 1.

A *rectangulation pattern* is a configuration of walls with prescribed directions and incidences. For example, the windmill patterns  describe four walls such that when considering the walls in clockwise or counterclockwise order, respectively, the end vertex of one wall lies in the interior of the next wall. We can also think of a pattern as the rectangulation formed by the given walls and incidences. For example, we can think of the windmill patterns as rectangulations with 5 rectangles. We say that a rectangulation  $R$  *contains* the pattern  $P$ , if  $R$  contains a subset of walls with the directions and incidences specified by  $P$ . Otherwise we say that  $R$  *avoids*  $P$ . For any set of rectangulation patterns  $\mathcal{P}$  and for any set of rectangulations  $\mathcal{C}$ , we write  $\mathcal{C}(\mathcal{P})$  for the rectangulations from  $\mathcal{C}$  that avoid each pattern from  $\mathcal{P}$ . For example, diagonal rectangulations are given by  $\mathcal{D}_n = \mathcal{R}_n(\{\text{diag patterns}\})$ .

We say that a rectangulation pattern  $P$  is *tame*, if for any rectangulation  $R$  that avoids  $P$ , we also have that  $c_1(R)$  and  $c_{\nu(R)}(R)$  avoid  $P$ . In words, inserting a new rectangle below  $R$  or to the right of  $R$  must not create the pattern  $P$ . These definitions yield the next lemma.

► **Lemma 6.** *If a rectangulation pattern is neither bottom-based nor right-based, then it is tame. In particular, each of the patterns  is tame.*

The following powerful theorem allows to obtain many new zigzag sets of rectangulations from a given zigzag set  $\mathcal{C}_n \subseteq \mathcal{R}_n$  by forbidding one or more tame patterns. All of these zigzag sets can then be generated by our Algorithm  $J^\square$ .

► **Theorem 7.** *Let  $\mathcal{C}_n \subseteq \mathcal{R}_n$  be a zigzag set of rectangulations, and let  $\mathcal{P}$  be a set of tame rectangulation patterns. Then  $\mathcal{C}_n(\mathcal{P})$  is a zigzag set of rectangulations. Moreover, if  $\mathcal{P}$  is symmetric, then  $\mathcal{C}_n(\mathcal{P})$  is symmetric.*

Recall that  $\mathcal{P}$  is symmetric if for each pattern  $P \in \mathcal{P}$ , we have that the pattern obtained from  $P$  by reflection at the main diagonal is also in  $\mathcal{P}$ . The significance of the second part of the theorem is that if  $\mathcal{C}_n(\mathcal{P})$  is symmetric, then the ordering of rectangulations of  $\mathcal{C}_n(\mathcal{P})$  generated by Algorithm  $J^\square$  is cyclic by Theorem 5. See [23] for a proof of Theorem 7.

## 5 Efficient computation

Recall from Remark 4 that Algorithm  $J^\square$  in its stated form is unsuitable for efficient implementation. We now discuss how to make the algorithm efficient, so as to achieve the time bounds claimed in Table 1 for several interesting classes of rectangulations.

### 5.1 Memoryless algorithm

Consider Algorithm  $M^\square$  below, which takes as input a zigzag set of rectangulations  $\mathcal{C}_n \subseteq \mathcal{R}_n$  and generates them exhaustively by minimal jumps in the same order as Algorithm  $J^\square$ , i.e., in the order  $J^\square(\mathcal{C}_n)$ . After initialization in line M1, the algorithm loops over lines M2–M5, visiting the current rectangulation  $R$  at the beginning of each iteration (line M2), until it terminates (line M3). The key idea is to track explicitly which rectangle jumps in each step, and the direction of the jump. With this information, the jump is determined by the condition that it must be minimal w.r.t.  $\mathcal{C}_n$ , i.e., starting from the current insertion point of the given rectangle, we choose the first insertion point (w.r.t. their linear ordering) for that rectangle in the given direction that creates the next rectangulation from  $\mathcal{C}_n$ .

■ **Algorithm  $M^\square$**  (Memoryless minimal jumps).

---

This algorithm generates all rectangulations of a zigzag set  $\mathcal{C}_n \subseteq \mathcal{R}_n$  by minimal jumps in the same order as Algorithm  $J^\square$ . It maintains the current rectangulation in the variable  $R$ , and auxiliary arrays  $o = (o_1, \dots, o_n)$  and  $s = (s_1, \dots, s_n)$ .

- M1.** [Initialize] Set  $R \leftarrow \left[ \begin{smallmatrix} n \\ \dots \end{smallmatrix} \right]$ , and  $o_j \leftarrow \triangleleft$ ,  $s_j \leftarrow j$  for  $j = 1, \dots, n$ .
  - M2.** [Visit] Visit the current rectangulation  $R$ .
  - M3.** [Select rectangle] Set  $j \leftarrow s_n$ , and terminate if  $j = 1$ .
  - M4.** [Jump rectangle] In the current rectangulation  $R$ , perform a jump of rectangle  $r_j$  that is minimal w.r.t.  $\mathcal{C}_n$ , where the jump direction is left if  $o_j = \triangleleft$  and right if  $o_j = \triangleright$ .
  - M5.** [Update  $o$  and  $s$ ] Set  $s_n \leftarrow n$ . If  $o_j = \triangleleft$  and  $R^{[j]}$  is bottom-based set  $o_j \leftarrow \triangleright$ , or if  $o_j = \triangleright$  and  $R^{[j]}$  is right-based set  $o_j \leftarrow \triangleleft$ , and in both cases set  $s_j \leftarrow s_{j-1}$  and  $s_{j-1} \leftarrow j - 1$ . Go back to M2.
- 

Specifically, the jump directions are maintained by an array  $o = (o_1, \dots, o_n)$ , where  $o_j = \triangleleft$  means that rectangle  $r_j$  performs a left jump in the next step, and  $o_j = \triangleright$  means that rectangle  $r_j$  performs a right jump in the next step (line M4). All sub-rectangulations of the initial rectangulation  $\left[ \begin{smallmatrix} n \\ \dots \end{smallmatrix} \right]$  are right-based, so the initial jump directions are  $o_j = \triangleleft$  for  $j = 1, \dots, n$  (line M1). Whenever rectangle  $r_j$  jumps left and reaches the first insertion point, which means that  $R^{[j]}$  is bottom-based, or if it jumps right and reaches the last insertion point, which means that  $R^{[j]}$  is right-based, then the jump direction  $o_j$  is reversed (line M5).

The array  $s = (s_1, \dots, s_n)$  is used to determine which rectangle jumps in each step. Specifically, the last entry  $s_n$  determines the rectangle that jumps in the current iteration (line M3). This array simulates a stack in a loopless fashion, following an idea first used by Bitner, Ehrlich, and Reingold [7]. The stack is initialized by  $(s_1, \dots, s_n) = (1, 2, \dots, n)$  (line M1), with  $s_n$  being the value on the top of the stack. The stack is popped (by the

instruction  $s_j \leftarrow s_{j-1}$  in line M5) when rectangle  $r_j$  reaches its first or last insertion point in this step, meaning that this rectangle is not eligible to jump in the next step, but becomes eligible again after the next step, which is achieved by pushing the value  $j$  on the stack again (by the instructions  $s_n \leftarrow n$  and  $s_{j-1} \leftarrow j - 1$  in line M5). See Table 2 for an example.

► **Theorem 8.** For any zigzag set of rectangulations  $\mathcal{C}_n \subseteq \mathcal{R}_n$ , Algorithm  $M^\square$  visits every rectangulation from  $\mathcal{C}_n$  exactly once, in the order  $J^\square(\mathcal{C}_n)$  defined by Algorithm  $J^\square$ .

To make meaningful statements about the running time of Algorithm  $M^\square$ , we need to specify the data structures used to represent the current rectangulation  $R$ , and the operations on this data structure to perform the operations in lines M4 and M5. Most importantly, we need to develop oracles which efficiently compute the next minimal jump w.r.t.  $\mathcal{C}_n$  for some interesting zigzag sets  $\mathcal{C}_n$ . One should think of  $\mathcal{C}_n$  here as a class of rectangulations specified by some properties or forbidden patterns, such as “diagonal guillotine rectangulations”, and not as large precomputed set of rectangulations. All of these details can be found in [23], and they are part of our C++ implementation provided in [10].

■ **Table 2** Execution of Algorithm  $M^\square$  for the set  $\mathcal{C}_4 = \mathcal{D}_4$  of diagonal rectangulations with 4 rectangles. Empty entries in the  $o$  and  $s$  column are unchanged compared to the previous row.

$J^\square(\mathcal{C}_4)$	jump	$o_1o_2o_3o_4$	$s_1s_2s_3s_4$	$J^\square(\mathcal{C}_4)$	jump	$o_1o_2o_3o_4$	$s_1s_2s_3s_4$
1	$\overleftarrow{J}(R, 4, 1)$	$\triangleleft \triangleleft \triangleleft \triangleleft$	1 2 3 4	12	$\overrightarrow{J}(R, 4, 1)$	$\triangleright$	1 1 4
2	$\overleftarrow{J}(R, 4, 1)$		4	13	$\overrightarrow{J}(R, 4, 1)$		4
3	$\overleftarrow{J}(R, 4, 1)$		4	14	$\overrightarrow{J}(R, 4, 1)$		4
4	$\overleftarrow{J}(R, 3, 1)$	$\triangleright$	3 3	15	$\overrightarrow{J}(R, 3, 1)$	$\triangleleft$	3 3
5	$\overrightarrow{J}(R, 4, 1)$		4	16	$\overleftarrow{J}(R, 4, 1)$		4
6	$\overrightarrow{J}(R, 4, 1)$		4	17	$\overleftarrow{J}(R, 4, 1)$		4
7	$\overrightarrow{J}(R, 4, 1)$		4	18	$\overleftarrow{J}(R, 4, 1)$		4
8	$\overleftarrow{J}(R, 3, 1)$	$\triangleleft$	3 3	19	$\overrightarrow{J}(R, 3, 1)$	$\triangleright$	3 3
9	$\overleftarrow{J}(R, 4, 1)$	$\triangleright$	2 2 4	20	$\overrightarrow{J}(R, 4, 1)$	$\triangleleft$	2 1 4
10	$\overleftarrow{J}(R, 4, 2)$		4	21	$\overrightarrow{J}(R, 4, 2)$		4
11	$\overleftarrow{J}(R, 2, 1)$	$\triangleright$	3 2	22		$\triangleleft$	3 1



## References

- 1 E. Ackerman, G. Barequet, and R. Y. Pinter. A bijection between permutations and floorplans, and its applications. *Discrete Appl. Math.*, 154(12):1674–1684, 2006. doi:10.1016/j.dam.2006.03.018.
- 2 E. Ackerman, G. Barequet, and R. Y. Pinter. On the number of rectangulations of a planar point set. *J. Combin. Theory Ser. A*, 113(6):1072–1091, 2006. doi:10.1016/j.jcta.2005.10.003.
- 3 K. Amano, S. Nakano, and K. Yamanaka. On the number of rectangular drawings: Exact counting and lower and upper bounds, 2007. IPSJ SIG Technical Report 2007-AL-115 (5).
- 4 A. Asinowski, G. Barequet, M. Bousquet-Mélou, T. Mansour, and R. Y. Pinter. Orders induced by segments in floorplans and (2-14-3, 3-41-2)-avoiding permutations. *Electron. J. Combin.*, 20(2):Paper 35, 43, 2013.
- 5 A. Asinowski and T. Mansour. Separable  $d$ -permutations and guillotine partitions. *Ann. Comb.*, 14(1):17–43, 2010. doi:10.1007/s00026-010-0043-8.
- 6 D. Avis and K. Fukuda. Reverse search for enumeration. *Discrete Appl. Math.*, 65(1-3):21–46, 1996. First International Colloquium on Graphs and Optimization (GOI), 1992 (Grimentz). doi:10.1016/0166-218X(95)00026-N.
- 7 J. R. Bitner, G. Ehrlich, and E. M. Reingold. Efficient generation of the binary reflected Gray code and its applications. *Comm. ACM*, 19(9):517–521, 1976. doi:10.1145/360336.360343.
- 8 J. Cardinal, V. Sacristán, and R. I. Silveira. A note on flips in diagonal rectangulations. *Discrete Math. Theor. Comput. Sci.*, 20(2):Paper No. 14, 22, 2018.
- 9 J. Conant and T. Michaels. On the number of tilings of a square by rectangles. *Ann. Comb.*, 18(1):21–34, 2014. doi:10.1007/s00026-013-0209-2.
- 10 The Combinatorial Object Server: <http://www.combos.org/rect>.
- 11 G. Ehrlich. Loopless algorithms for generating permutations, combinations, and other combinatorial configurations. *J. Assoc. Comput. Mach.*, 20:500–513, 1973. doi:10.1145/321765.321781.
- 12 D. Eppstein, E. Mumford, B. Speckmann, and K. Verbeek. Area-universal and constrained rectangular layouts. *SIAM J. Comput.*, 41(3):537–564, 2012. doi:10.1137/110834032.
- 13 R. Fujimaki, Y. Inoue, and T. Takahashi. An asymptotic estimate of the numbers of rectangular drawings or floorplans. In *2009 IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 856–859, 2009. doi:10.1109/ISCAS.2009.5117891.
- 14 E. Hartung, H. P. Hoang, T. Mütze, and A. Williams. Combinatorial generation via permutation languages. In Shuchi Chawla, editor, *Proceedings of the 2020 ACM-SIAM Symposium on Discrete Algorithms, SODA 2020, Salt Lake City, UT, USA, January 5-8, 2020*, pages 1214–1225. SIAM, 2020. doi:10.1137/1.9781611975994.74.
- 15 E. Hartung, H. P. Hoang, T. Mütze, and A. Williams. Combinatorial generation via permutation languages. I. Fundamentals, 2020. To appear in *Trans. Amer. Math. Soc.*; preprint available at [arXiv:1906.06069](https://arxiv.org/abs/1906.06069).
- 16 B. D. He. A simple optimal binary representation of mosaic floorplans and Baxter permutations. *Theoret. Comput. Sci.*, 532:40–50, 2014. doi:10.1016/j.tcs.2013.05.007.
- 17 H. P. Hoang and T. Mütze. Combinatorial generation via permutation languages. II. Lattice congruences, 2020. To appear in *Israel J. Math.*; preprint available at [arXiv:1911.12078](https://arxiv.org/abs/1911.12078).
- 18 X. Hong, G. Huang, Y. Cai, J. Gu, S. Dong, C.-K. Cheng, and J. Gu. Corner block list: An effective and efficient topological representation of non-slicing floorplan. In E. Sentovich, editor, *Proceedings of the 2000 IEEE/ACM International Conference on Computer-Aided Design, 2000, San Jose, California, USA, November 5-9, 2000*, pages 8–12. IEEE Computer Society, 2000. doi:10.1109/ICCAD.2000.896442.
- 19 Y. Inoue, T. Takahashi, and R. Fujimaki. Counting rectangular drawings or floorplans in polynomial time. *IEICE Trans. Fundam. Electron. Commun. Comput. Sci.*, 92-A(4):1115–1120, 2009. doi:10.1587/transfun.E92.A.1115.
- 20 D. E. Knuth. *The Art of Computer Programming. Vol. 4A. Combinatorial algorithms. Part 1*. Addison-Wesley, Upper Saddle River, NJ, 2011.

- 21 S. Law and N. Reading. The Hopf algebra of diagonal rectangulations. *J. Combin. Theory Ser. A*, 119(3):788–824, 2012. doi:10.1016/j.jcta.2011.09.006.
- 22 E. Meehan. The Hopf algebra of generic rectangulations, 2019. arXiv:1903.09874.
- 23 A. Merino and T. Mütze. Combinatorial generation via permutation languages. III. Rectangulations, 2021. Full preprint version of the present article. arXiv:2103.09333.
- 24 W. J. Mitchell, J. P. Steadman, and R. S. Liggett. Synthesis and optimization of small rectangular floor plans. *Environment and Planning B: Planning and Design*, 3(1):37–70, 1976. doi:10.1068/b030037.
- 25 S. Nakano. Enumerating floorplans with  $n$  rooms. In *Algorithms and computation (Christchurch, 2001)*, volume 2223 of *Lecture Notes in Comput. Sci.*, pages 107–115. Springer, Berlin, 2001. doi:10.1007/3-540-45678-3\_10.
- 26 OEIS Foundation Inc. The on-line encyclopedia of integer sequences, 2020. URL: <http://oeis.org>.
- 27 R. H. J. M. Otten. Automatic floorplan design. In J. S. Crabbe, C. E. Radke, and H. Ofek, editors, *Proceedings of the 19th Design Automation Conference, DAC '82, Las Vegas, Nevada, USA, June 14-16, 1982*, pages 261–267. ACM/IEEE, 1982. doi:10.1145/800263.809216.
- 28 V. Pilaud and F. Santos. Quotientopes. *Bull. Lond. Math. Soc.*, 51(3):406–420, 2019. doi:10.1112/blms.12231.
- 29 N. Reading. Generic rectangulations. *European J. Combin.*, 33(4):610–623, 2012. doi:10.1016/j.ejc.2011.11.004.
- 30 F. Ruskey. Combinatorial Gray code. In M.-Y. Kao, editor, *Encyclopedia of Algorithms*, pages 342–347. Springer, 2016.
- 31 C. Savage. A survey of combinatorial Gray codes. *SIAM Rev.*, 39(4):605–629, 1997. doi:10.1137/S0036144595295272.
- 32 Z. C. Shen and C. C. N. Chu. Bounds on the number of slicing, mosaic, and general floorplans. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 22(10):1354–1361, 2003. doi:10.1109/TCAD.2003.818136.
- 33 M. Takagi and S. Nakano. Listing all rectangular drawings with certain properties. *Systems and Computers in Japan*, 35(4):1–8, 2004. doi:10.1002/scj.10563.
- 34 M. van Kreveld and B. Speckmann. On rectangular cartograms. *Comput. Geom.*, 37(3):175–187, 2007. doi:10.1016/j.comgeo.2006.06.002.
- 35 A. Williams. The greedy Gray code algorithm. In *Algorithms and Data Structures - 13th International Symposium, WADS 2013, London, ON, Canada, August 12-14, 2013. Proceedings*, pages 525–536, 2013. doi:10.1007/978-3-642-40104-6\_46.
- 36 K. Yamanaka, M. S. Rahman, and S. Nakano. Enumerating floorplans with columns. *IEICE Trans. Fundam. Electron. Commun. Comput. Sci.*, 101-A(9):1392–1397, 2018. doi:10.1587/transfun.E101.A.1392.
- 37 B. Yao, H. Chen, C.-K. Cheng, and R. L. Graham. Floorplan representations: Complexity and connections. *ACM Trans. Design Autom. Electr. Syst.*, 8(1):55–80, 2003. doi:10.1145/606603.606607.
- 38 S. Yoshii, D. Chigira, K. Yamanaka, and S. Nakano. Constant time generation of rectangular drawings with exactly  $n$  faces. *IEICE Trans. Fundam. Electron. Commun. Comput. Sci.*, 89-A(9):2445–2450, 2006. doi:10.1093/ietfec/e89-a.9.2445.